ADSP-219x/2191 DSP Hardware Reference

Revision 1.1, August 2003

Part Number
82-00390-06

Analog Devices, Inc.
Digital Signal Processor Division
One Technology Way
Norwood, Mass. 02062-9106

CONTENTS

INTRODUCTION

HOST PORT

SERIAL PORTS (SPORTS)

ADSP-2191 DSP I/O REGISTERS

PREFACE

Thank you for purchasing and developing systems using ADSP-219x DSPs from Analog Devices.

Purpose of This Manual

The *ADSP-219x/2191 DSP Hardware Reference* provides architectural information on the ADSP-219x modified Harvard architecture Digital Signal Processor (DSP) core and ADSP-2191 DSP products.

This functional description also describes the ADSP-2191 memory derivatives, the ADSP-2195 and the ADSP-2196. Most of this manual refers to the ADSP-2191 DSP; refer to the chip data sheets for differences.

The architectural descriptions cover functional blocks, buses, and ports, including all the features and processes they support. For programming information, refer to the *ADSP-219x DSP Instruction Set Reference*.

Intended Audience

This manual is intended for system designers and programmers who are familiar with digital signal processing (DSP) concepts. Users should have a working knowledge of microcomputer technology and DSP related mathematics.

Manual Contents

This reference presents instruction information organized by the type of the instruction. Instruction types relate to the machine language opcode for the instruction. On this DSP, the opcodes categorize the instructions by the portions of the DSP architecture that execute the instructions. The following chapters cover the different types of instructions:

- "Introduction" on page 1-1—This chapter describes the DSP .
- "Computational Units" on page 2-1—This chapter describes the arithmetic/logic unit (ALU), multiplier/accumulator (multiplier), and shifter.
- "Program Sequencer" on page 3-1—This chapter describes program flow.
- "Data Address Generators (DAGs)" on page 4-1—This chapter describes the automatic generation of addresses for indirect addressing.
- "Memory" on page 5-1—This chapter describes how to use internal memory.
- "I/O Processor" on page 6-1—This chapter describes Direct Memory Access (DMA) of DSP memory through the external, host, serial, SPI, and UART ports.
- "External Port" on page 7-1—This chapter describes how to configure, connect, and access external memory or memory-mapped peripherals.
- "Host Port" on page 8-1—This chapter describes how to directly access the DSP memory space, boot space, and I/O space.
- "Serial Ports (SPORTs)" on page 9-1—This chapter describes the serial ports (SPORTS) available on the DSP.

- "Serial Peripheral Interface (SPI) Ports" on page 10-1—This chapter describes the use of the DSP's two SPI ports.
- "UART Port" on page 11-1—This chapter describes how to use its UART port.
- "Timer" on page 12-1—This chapter describes how to use the DSP's three 32-bit timers.
- "JTAG Test-Emulation Port" on page 13-1—This chapter describes the use of the DSP's JTAG port.
- "System Design" on page 14-1—This chapter describes basic system interface features of the ADSP-219x DSP family processors.
- "ADSP-219x DSP Core Registers" on page A-1—This chapter describes the DSP core's general-purpose.
- "ADSP-2191 DSP I/O Registers" on page B-1—This chapter describes the DSP's I/O processor registers.
- "Numeric Formats" on page C-1—This chapter describes various aspects of the 16-bit data format and how to implement a block floating-point format in software.

Additional Literature

For more information about Analog Devices DSPs and development products, see the following documents:

- *ADSP-2191 DSP Microcomputer Data Sheet*
- *ADSP-219x DSP Instruction Set Reference*

All the manuals are available in PDF format from the software distribution CD-ROM. You can also access these manuals via VisualDSP++ online Help.

What's New in This Manual

This revision of the *ADSP-219x/2191 DSP Hardware Reference* includes fixes to defects logged on the Analog Devices Web site under documentation errata. In addition, the following changes were made:

- A preface was added
- Several block diagrams and descriptions of core DSP components were updated in Chapter 1 "Introduction", Chapter 2 "Computational Units", Chapter 3 "Program Sequencer", Chapter 4 "Memory", and Chapter 12 "Timer".
- Appendix C "Interrupts" was moved to Chapter 6 together with new information.

Technical or Customer Support

You can reach our DSP Tools Customer Support in the following ways:

- E-mail development tools questions to `dsptools.support@analog.com`
- E-mail processor questions to `dsp.support@analog.com`
- Phone questions to **1800-ANALOGD**
- Visit our World Wide Web site at `http://www.analog.com/dsp`
- Telex questions to **924491, TWX:710/394-6577**
- Cable questions to **ANALOG NORWOODMASS**
- Contact your local Analog Devices sales office or an authorized Analog Devices distributor

Processor Family

The name *ADSP-219x* refers to the family of Analog Devices 16-bit, fixed-point processors. VisualDSP++™ currently supports these processors:

- ADSP-2191
- ADSP-2192-12
- ADSP-2195
- ADSP-2196
- Mixed-signal processors (ADSP-21990, ADSP-21991, and ADSP-21992)

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from printed documents/manuals.

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

DSP Product Information

For information on digital signal processors, visit our Web site at **http://www.analog.com/dsp**. It provides access to technical information and documentation, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products by:

- FAXing questions or requests for information:
  1(781)461-3010 (North America) or
  089/76 903-557 (Europe Headquarters)
- Accessing the FTP site:
  ftp ftp.analog.com or **ftp 137.71.23.21** or
  ftp://ftp.analog.com

Product Related Documents

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

- *VisualDSP++ Getting Started Guide for 16-Bit Processors*
- *VisualDSP++ User's Guide for 16-Bit Processors*
- *VisualDSP++ C/C++ Compiler and Library Manual for ADSP-219x DSPs*
- *VisualDSP++ Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs*
- *VisualDSP++ Linker and Utilities Manual for 16-Bit Processors*
- *VisualDSP++ Loader and Utilities Manual for 16-Bit Processors*
- *VisualDSP++ Kernel (VDK) User's Guide for 16-Bit Processors*
- *VisualDSP++ Component Software Engineering User's Guide for 16-Bit Processors*

Technical Publications Online or on the Web

You can access DSP (**or TigerSHARC processor**) documentation in these ways:

- **Online Access using VisualDSP++ Installation CD-ROM**

  Your VisualDSP++™ software distribution CD-ROM includes all of the listed VisualDSP++ software tool publications.

  After you install VisualDSP++ software on your PC, select the **Help Topics** command on the VisualDSP++ **Help** menu, click the **Reference** book icon, and select **Online Manuals**. From this **Help** topic, you can open any of the manuals, which are either in HTML format or in Adobe Acrobat PDF format.

  If you are not using VisualDSP++, you can manually access these PDF files from the CD-ROM using Adobe Acrobat.

- **Web Access**

  Use the Analog Devices technical publications Web site http://**www.analog.com/industry/dsp/tech_doc/gen_purpose.html** to access DSP publications, including data sheets, hardware reference manuals, instruction set reference manuals, and VisualDSP++ software documentation. You can view, download, or print in PDF format. Some publications are also available in HTML format.

Printed Manuals

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

VisualDSP++ and Tools Manuals

The VisualDSP++ and Tools manuals can be purchased through your local Analog Devices sales office or an authorized Analog Devices distributor. These manuals can be purchased only as a kit.

Hardware Manuals

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

Data Sheets

All data sheets (preliminary and production) can be downloaded from the Analog Devices Web site. As a general rule, only production (not preliminary) data sheets can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**). You can request data sheets using part numbers.

If you want to have a data sheet faxed to you, use the Analog Devices Faxback system at **1-800-446-6212**. Follow the prompts, and you can either get a particular data sheet or a list of the data sheet code numbers faxed to you. If the data sheet you request is not listed on Faxback, check for it on the Web site.

Recommendations for Improving Our Documents

Please send us your comments and recommendation on how to improve our manuals. Contact us at:

- Software/Development Tools manuals
  `dsptools.support@analog.com`
- Data sheets, Hardware and Instruction Reference Set manuals
  `dsp.support@analog.com`

Conventions

The following table identifies and describes text conventions used in this manual.

Note that additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative 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 ellipse; 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.

Example	Description
AX0, SR, PX	Register names appear in UPPERCASE and keyword font
TMR0E, $\overline{\text{RESET}}$	Pin names appear in UPPERCASE and keyword font; active low signals appear with an $\overline{\text{OVERBAR}}$.
DRx, $\overline{\text{MS3-0}}$	Register and pin names in the text may refer to groups of registers or pins. When a lowercase “x” appears in a register name (e.g., DRx), that indicates a set of registers (e.g., DR0, DR1, and DR2). A range also may be shown with a hyphen (e.g., $\overline{\text{MS3-0}}$ indicates $\overline{\text{MS3}}$, $\overline{\text{MS2}}$, $\overline{\text{MS1}}$, and $\overline{\text{MS0}}$).
IF, DO/UNTIL	Assembler instructions (mnemonics) appear in UPPERCASE and in keyword font
ⓘ	This symbol indicates a note that provides supplementary information on a related topic. In the online Help version of this book, the word **Note** appears instead of this symbol.
⊘	This symbol indicates a warning that advises on an inappropriate usage of the product that could lead to undesirable results or product damage. In the online Help version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

This description covers the ADSP-2191 and memory derivatives, the ADSP-2195 and the ADSP-2196. Most of this manual refers to the ADSP-2191 DSP; refer to the chip data sheets for differences.

This chapter provides the following sections:

Overview—Why Fixed-Point DSP?

A digital signal processor's (DSP's) data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Because 16-bit, fixed-point DSP math is required for certain DSP coding algorithms, using a 16-bit, fixed-point DSP can provide all the features needed for certain algorithm and software development efforts. Also, a narrower bus width (16-bit as opposed to 32- or 64-bit wide) leads to reduced power consumption and other design savings. The extent to which this is true depends on the fixed-point processor's architecture.

High-level language programmability, large address spaces, and wide dynamic range allow system development time to be spent on algorithms and signal processing concerns, rather than assembly language coding, code paging, and error handling. The ADSP-2191 DSP is a highly integrated, 16-bit fixed-point DSP that provides many of these design advantages.

ADSP-219x Design Advantages

The ADSP-219x family DSPs are high-performance 16-bit DSPs for communications, instrumentation, industrial/control, voice/speech, medical, military, and other applications. These DSPs provide a DSP core that is compatible with previous ADSP-2100 family DSPs, but provides many additional features. The ADSP-219x core combines with on-chip peripherals to form a complete system-on-a-chip. The off-core peripherals add on-chip SRAM, integrated I/O peripherals, timer, and interrupt controller.

The ADSP-219x architecture balances a high-performance processor core with high performance buses (PM, DM, DMA). In the core, every computational instruction can execute in a single cycle. The buses and instruction cache provide rapid, unimpeded data flow to the core to maintain the execution rate.

Figure 1-1 on page 1-4 shows a detailed block diagram of the processor, illustrating the following architectural features:

- Computation units—multiplier, ALU, shifter, and data register file
- Program sequencer with related instruction cache, interval timer, and Data Address Generators (DAG1 and DAG2)
- Dual-blocked SRAM
- External ports for interfacing to off-chip memory, peripherals, and hosts

- Input/Output (I/O) processor with integrated DMA controllers, serial ports (SPORTs), serial peripheral interface (SPI) ports, and a UART port
- JTAG Test Access Port for board test and emulation

Figure 1-1 on page 1-4 also shows the three on-chip buses of the ADSP-219x: the PM bus, DM bus, and DMA bus. The PM bus provides access to either instructions or data. During a single cycle, these buses let the processor access two data operands (one from PM and one from DM), and access an instruction (from the cache).

The buses connect to the ADSP-219x's external port, which provides the processor's interface to external memory, I/O memory-mapped, and boot memory. The external port performs bus arbitration and supplies control signals to shared, global memory and I/O devices.

Further, the ADSP-219x addresses the five central requirements for DSPs:

- Fast, flexible arithmetic computation units

  Fast, Flexible Arithmetic. The ADSP-219x family DSPs execute all computational instructions in a single cycle. They provide both fast cycle times and a complete set of arithmetic operations.

- Unconstrained data flow to and from the computation units

  Unconstrained Data Flow. The ADSP-219x has a modified Harvard architecture combined with a data register file. In every cycle, the DSP can:

  - Read two values from memory or write one value to memory
  - Complete one computation
  - Write up to three values back to the register file

- Extended precision and dynamic range in the computation units

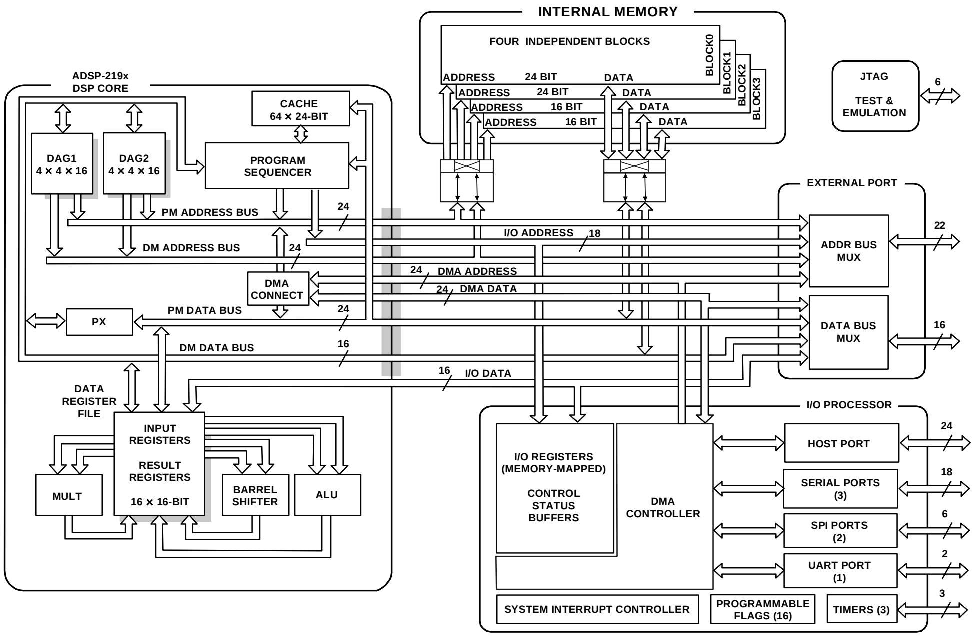


Figure 1-1. ADSP-219x/ADSP-2191 DSP Block Diagram

40-Bit Extended Precision. The DSP handles 16-bit integer and fractional formats (twos-complement and unsigned). The processors carry extended precision through result registers in their computation units, limiting intermediate data truncation errors.

- Dual address generators with circular buffering support

Dual Address Generators. The DSP has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus and bit-reverse operations are supported with memory page constraints on data buffer placement only.

- Efficient program sequencing

  Efficient Program Sequencing. In addition to zero-overhead loops, the DSP supports quick setup and exit for loops. Loops are both nestable (eight levels in hardware) and interruptable. The processors support both delayed and non-delayed branches.

ADSP-219x Architecture

This section provides the following topics:

Overview

An ADSP-219x is a single-chip microcomputer optimized for digital signal processing (DSP) and other high speed numeric processing applications. These DSPs provide a complete system-on-a-chip, integrating a large, high-speed SRAM and I/O peripherals supported by a dedicated DMA bus. The following sections summarize the features of each functional block in the ADSP-219x architecture, which appears in Figure 1-1 on page 1-4.

The ADSP-2191 combines the ADSP-219x family base architecture (three computational units, two data address generators, and a program sequencer) with three serial ports, two SPI-compatible ports, one UART port, a DMA controller, three programmable timers, general-purpose Programmable Flag pins, extensive interrupt capabilities, and on-chip program and data memory blocks.

The ADSP-2191 architecture is code compatible with ADSP-218x family DSPs. Though the architectures are compatible, the ADSP-2191 architecture has a number of enhancements over the ADSP-218x architecture. The enhancements to computational units, data address generators, and program sequencer make the ADSP-2191 more flexible and even easier to program than the ADSP-218x DSPs.

Indirect addressing options provide addressing flexibility—pre-modify with no update, pre- and post-modify by an immediate 8-bit, twos-complement value and base address registers for easier implementation of circular buffering.

The **ADSP-2191** DSP integrates 64K words of on-chip memory configured as 32K words 24-bit SRAM and 32K words of 16-bit SRAM. The **ADSP-2195** DSP features 16K of 24-bit SRAM and 16K words of 16-bit SRAM, whereas the **ADSP-2196** DSP provides 8K words of 24-bit SRAM and 8K words of 16-bit SRAM.

The ADSP-2191's flexible architecture and comprehensive instruction set support multiple operations in parallel. For example, in one processor cycle, the ADSP-2191 can:

- Generate an address for the next instruction fetch
- Fetch the next instruction

- Perform one or two data moves
- Update one or two data address pointers
- Perform a computational operation

These operations take place while the processor continues to:

- Receive and transmit data through two serial ports
- Receive and/or transmit data from a host
- Receive or transmit data through the UART
- Receive or transmit data over two SPI ports
- Access external memory through the external memory interface
- Decrement the timers

DSP Core Architecture

The ADSP-219x instruction set provides flexible data moves and multifunction (one or two data moves with a computation) instructions. Every single-word instruction can be executed in a single processor cycle. The ADSP-219x assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

Figure 1-1 on page 1-4 shows the architecture of the ADSP-219x core. It contains three independent computational units: the ALU, the multiplier/accumulator, and the shifter. The computational units process 16-bit data from the register file and have provisions to support multiprecision computations. The ALU performs a standard set of arithmetic and logic operations; division primitives also are supported. The multiplier performs single-cycle multiply, multiply/add, and multiply/subtract operations. The multiplier has two 40-bit accumulators, which help with overflow. The shifter performs logical and arithmetic shifts, normalization, denormalization, and derive exponent operations. The shifter can efficiently implement numeric format control, including multiword and block floating-point representations.

Register-usage rules influence placement of input and results within the computational units. For most operations, the computational units' data registers act as a data register file, permitting any input or result register to provide input to any unit for a computation. For feedback operations, the computational units let the output (result) of any unit be input to any unit on the next cycle. For conditional or multifunction instructions, there are restrictions limiting which data registers may provide inputs or receive results from each computational unit. For more information, see "Multifunction Computations" on page 2-64.

A powerful program sequencer controls the flow of instruction execution. The sequencer supports conditional jumps, subroutine calls, and low interrupt overhead. With internal loop counters and loop stacks, the ADSP-2191 executes looped code with zero overhead; no explicit jump instructions are required to maintain loops.

Two data address generators (DAGs) provide addresses for simultaneous dual operand fetches (from data memory and program memory). Each DAG maintains and updates four 16-bit address pointers. Whenever the pointer is used to access data (indirect addressing), it is pre- or post-modified by the value of one of four possible modify registers. A length value and base address may be associated with each pointer to implement automatic modulo addressing for circular buffers. Page registers in the DAGs allow circular addressing within 64K word boundaries of each of the 256 memory pages, but these buffers may not cross page boundaries. Secondary registers duplicate all the primary registers in the DAGs; switching between primary and secondary registers provides a fast context switch.

Efficient data transfer in the core is achieved by using internal buses:

- Program Memory Address (PMA) Bus
- Program Memory Data (PMD) Bus

- Data Memory Address (DMA) Bus
- Data Memory Data (DMD) Bus
- DMA Address Bus
- DMA Data Bus

The internal address buses share a single external address bus, allowing memory to be expanded off-chip, and the data buses share a single external data bus. Boot memory space and external I/O memory space also share the external buses.

Program memory can store both instructions and data, permitting the ADSP-219x to fetch two operands in a single cycle, one from program memory and one from data memory. The DSP's dual memory buses also let the ADSP-219x core fetch an operand from data memory and the next instruction from program memory in a single cycle.

DSP Peripherals Architecture

Figure 1-1 on page 1-4 shows the DSP's on-chip peripherals, which include the external memory interface, host port, serial ports, SPI compatible ports, UART port, JTAG test and emulation port, timers, flags, and interrupt controller. Figure 1-2 on page 1-11 illustrates a typical ADSP-2191 system with peripheral connections.

The ADSP-2191 has a 16-bit host port with DMA capability that provides external hosts access to on-chip memory. This parallel port consists of a multiplexed data/address bus and provides a low-service overhead data move capability. Configurable for 8- or 16-bit data bus widths, this port provides a glueless interface to a wide variety of 8- and 16-bit microcontrollers. Two chip-selects provide hosts access to the DSP's entire memory map. The DSP is bootable through this port.

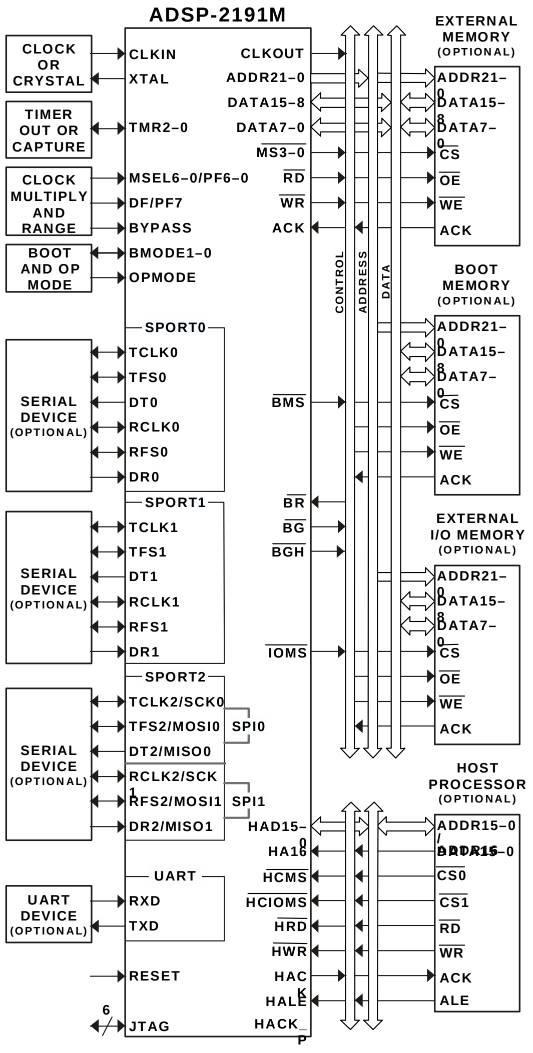


Figure 1-2. ADSP-219x/ADSP-2191 DSP Block Diagram

The ADSP-2191 also has an external memory interface that is shared by the DSP's core, the DMA controller, and DMA capable peripherals, which include the UART port, serial ports, SPI ports, and the host port.

The external port consists of an 8- or 16-bit data bus, a 22-bit address bus, and control signals. The data bus is configurable to provide an 8- or 16-bit interface to external memory. Support for word packing lets the DSP access 16- or 24-bit words from external memory regardless of the external data bus width. When configured for an 8-bit interface, the unused eight lines provide eight programmable, bidirectional general purpose Programmable Flag lines, six of which can be mapped to software condition signals.

The memory DMA controller lets the ADSP-2191 transfer data to and from internal and external memory. On-chip peripherals also can use this port for DMA transfers to and from memory.

The ADSP-2191 can respond to up to 16 interrupt sources at any given time: three internal (stack, emulator kernel, and power-down), two external (emulator and reset), and twelve user-defined (peripherals) interrupt requests. Programmers assign a peripheral to one of the 12 user defined interrupt requests. These assignments determine the priority of each peripheral for interrupt service. Several peripherals can be combined on a single interrupt request line.

There are three serial ports on the ADSP-2191 that provide a complete synchronous, full-duplex serial interface. This interface includes optional companding in hardware and a wide variety of framed or frameless data transmit and receive modes of operation. Each serial port can transmit or receive an internal or external, programmable serial clock and frame syncs. Each serial port supports 128-channel time division multiplexing (TDM).

The ADSP-2191 provides up to sixteen general-purpose I/O pins, which are programmable as either inputs or outputs. Eight of these pins are dedicated general purpose programmable flag pins. The other eight are multifunctional pins, acting as general purpose I/O pins when the DSP connects to an 8-bit external data bus and acting as the upper eight data pins when the DSP connects to a 16-bit external data bus. These programmable flag pins can implement edge- or level-sensitive interrupts. The execution of conditional instructions can be based on some of the programmable flag pins.

Three programmable interval timers generate periodic interrupts. Each timer can be independently set to operate in one of three modes:

- Pulse waveform generation mode
- Pulse width count/capture mode
- External event watchdog mode

Each timer has one bi-directional pin and four registers that implement its mode of operation: a configuration register, a count register, a period register, and a pulsewidth register. A single status register supports all three timers. A bit in the mode status register globally enables or disables all three timers, and a bit in each timer's configuration register enables or disables the corresponding timer independently of the others.

Memory Architecture

The **ADSP-2191** provides 64K words of on-chip memory. This memory is divided into four 16K blocks located on memory page 0 in the DSP's memory map. The **ADSP-2195** features only two 16K blocks, and the **ADSP-2196** has two 8K blocks. In addition to the internal and external memory space, the ADSP-2191 can address two additional and separate memory spaces: I/O space and boot space.

As shown in Figure 1-3 on page 1-14, the DSP's two internal memory blocks populate all of Page 0. The entire DSP memory map consists of 256 pages (pages 0-255), and each page is 64K words long. External memory space consists of four memory banks (banks 3–0) and supports a wide variety of SRAM memory devices. Each bank is selectable using the memory select pins ($\overline{\texttt{MS3-0}}$) and has configurable page boundaries, waitstates, and waitstate modes. The 1K word of on-chip boot-ROM populates the lower 1K addresses of page 255. Other than page 0 and page 255, the remaining 254 pages are addressable off-chip. I/O memory pages differ from external memory pages in that I/O pages are 1K word long, and the external I/O pages have their own select pin ($\overline{\text{IOMS}}$). Pages 0–7 of I/O memory space reside on-chip and contain the configuration registers for the peripherals. Both the DSP core and DMA-capable peripherals can access the DSP's entire memory map.

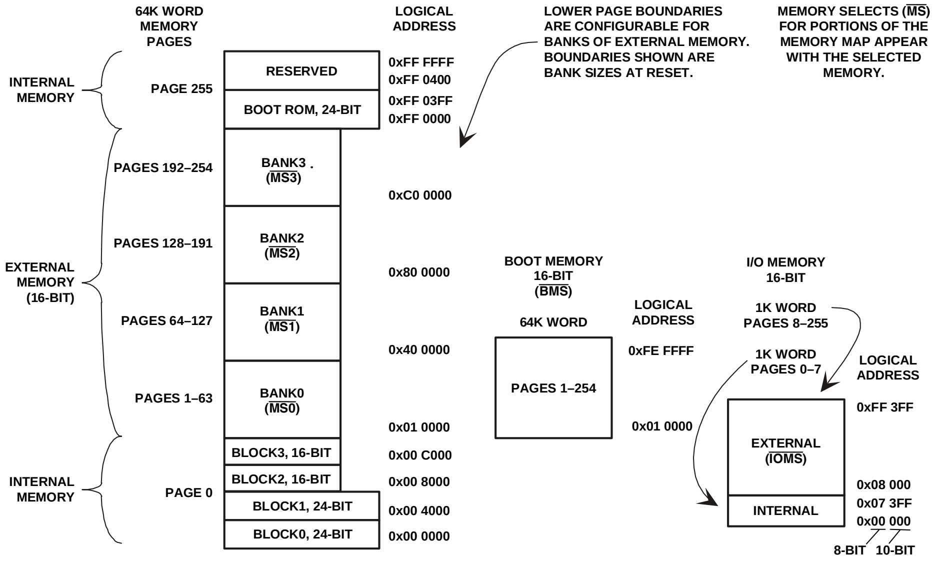


Figure 1-3. ADSP-2191 Internal/External Memory, Boot Memory, and I/O Memory Maps

Internal (On-Chip) Memory

The ADSP-2191's unified program and data memory space consists of 16M locations that are accessible through two 24-bit address buses, the PMA and DMA buses. The DSP uses slightly different mechanisms to generate a 24-bit address for each bus. The DSP has three functions that support access to the full memory map.

The DAGs generate 24-bit addresses for data fetches from the entire DSP memory address range. Because DAG index (address) registers are 16 bits wide and hold the lower 16-bits of the address, each of the DAGs has its own 8-bit page register (DMPGx) to hold the most significant eight address bits. Before a DAG generates an address, the program must set the DAG's DMPGx register to the appropriate memory page.

- The program sequencer generates the addresses for instruction fetches. For relative addressing instructions, the program sequencer bases addresses for relative jumps, calls, and loops on the 24-bit Program Counter (PC) register. For direct addressing instructions (two-word instructions), the instruction provides an immediate 24-bit address value. The PC allows linear addressing of the full 24 bit address range.

- The program sequencer relies on an 8-bit Indirect Jump Page (IJPG) register to supply the most significant eight address bits for indirect jumps and calls that use a 16-bit DAG address register for part of the branch address. Before a cross page jump or call, the program must set the program sequencer's IJPG register to the appropriate memory page.

The ADSP-2191 has 1K word of on-chip ROM that holds boot routines. If peripheral booting is selected, the DSP starts executing instructions from the on-chip boot ROM, which starts the boot process from the selected peripheral. For more information, see "Booting Modes" on page 1-22. The on-chip boot ROM is located on Page 255 in the DSP's memory map.

The ADSP-2191 has internal I/O memory for peripheral control and status registers. For more information, see the I/O memory space discussion on page 1-16.

External (Off-Chip) Memory

Each of the ADSP-2191's off-chip memory spaces has a separate control register, so applications can configure unique access parameters for each space. The access parameters include read and write wait counts, waitstate completion mode, I/O clock divide ratio, write hold time extension, strobe polarity, and data bus width. The core clock and peripheral clock ratios influence the external memory access strobe widths. For more information, see "Clock Signals" on page 1-21. The off-chip memory spaces are:

- External memory space ($\overline{\texttt{MS3-0}}$ pins)
- I/O memory space ($\overline{\texttt{IOMS}}$ pin)
- Boot memory space ($\overline{\texttt{BMS}}$ pin)

All of these off-chip memory spaces are accessible through the external port, which can be configured for 8-bit or 16-bit data widths.

External Memory Space. External memory space consists of four memory banks. These banks can contain a configurable number of 64K word pages. At reset, the page boundaries for external memory have Bank 0 containing pages 1-63, Bank 1 containing pages 64-127, Bank 2 containing pages 128-191, and Bank 3 containing pages 192-254. The $\overline{\texttt{MS3-0}}$ memory bank pins select Bank 3-0, respectively. The external memory interface decodes the eight MSBs of the DSP program address to select one of the four banks. Both the DSP core and DMA-capable peripherals can access the DSP's external memory space.

I/O Memory Space. The ADSP-2191 supports an additional external memory called I/O memory space. This space is designed to support simple connections to peripherals (such as data converters and external registers) or to bus interface ASIC data registers. I/O space supports a total of 256K locations. The first 8K addresses are reserved for on-chip peripherals. The upper 248K addresses are available for external peripheral devices and are selected with the $\overline{\texttt{IOMS}}$ pin. The DSP's instruction set provides instructions for accessing I/O space. These instructions use an 18-bit address that is assembled from an 8-bit I/O Memory Page (IOPG) register and a 10-bit immediate value supplied in the instruction. Both the ADSP-219x core and a host (through the host port) can access I/O memory space.

Boot Memory Space. Boot memory space consists of one off-chip bank with 253 pages. The $\overline{\text{BMS}}$ pin selects boot memory space. Both the DSP core and DMA-capable peripherals can access the DSP's off-chip boot memory space. If the DSP is configured to boot from boot memory space, the DSP starts executing instructions from the on-chip boot ROM, which starts booting the DSP from boot memory. For more information, see "Booting Modes" on page 1-22.

Interrupts

The interrupt controller allows the DSP to respond to 17 interrupts with minimal overhead. The controller implements an interrupt priority scheme, allowing programs assign interrupt priorities to each peripheral. For more information, see "System Interrupt Controller" on page 6-1.

DMA Controller

The ADSP-2191 has a DMA controller that supports automated data transfers with minimal overhead for the DSP core. Cycle stealing DMA transfers can occur between the ADSP-2191's internal memory and any of its DMA capable peripherals. Additionally, DMA transfers also can be accomplished between any of the DMA capable peripherals and external devices connected to the external memory interface. DMA capable peripherals include the host port, serial ports, SPI ports, UART port, and memory-to-memory (memDMA) DMA channel. Each individual DMA capable peripheral has one or more dedicated DMA channels. For a description of each DMA sequence, the DMA controller uses a set of parameters—called a DMA descriptor. When successive DMA sequences are needed, these descriptors can be linked or chained together. When chained, the completion of one DMA sequence auto-initiates and starts the next sequence. DMA sequences do not contend for bus access with the DSP core, instead DMAs "steal" cycles to access memory.

Host Port

The ADSP-2191's host port functions as a slave on the external bus of an external host. The host port interface lets a host read from or write to the DSP's memory space, boot space, or internal I/O space. Examples of hosts include external microcontrollers, microprocessors, or ASICs.

The host port is a multiplexed address and data bus that provides an 8- or 16-bit data path and operates using an asynchronous transmission protocol. To access the DSP's internal memory space, a host steals one cycle per access from the DSP. A host access to the DSP's external memory uses the external port interface and does not stall (or steal cycles from) the DSP's core. Because a host can access internal I/O memory space, a host can control any of the DSP's I/O mapped peripherals.

DSP Serial Ports (SPORTs)

The ADSP-2191 incorporates three complete synchronous serial ports (SPORT0, SPORT1, and SPORT2) for serial and multiprocessor communications. The SPORTs support the following features:

- Bidirectional operation—each SPORT has independent transmit and receive pins.
- Buffered (eight-deep) transmit and receive ports—each port has a data register for transferring data words to and from other DSP components and shift registers for shifting data in and out of the data registers.

- Clocking—each transmit and receive port either can use an external serial clock (≤75 MHz) or generate its own, in frequencies ranging from 1144 Hz to 75 MHz.
- Word length—each SPORT supports serial data words from 3- to 16-bits in length transferred in big endian (MSB) or little endian (LSB) format.
- Framing—each transmit and receive port can run with or without frame sync signals for each data word.
- Companding in hardware—each SPORT can perform A-law or μ-law companding, according to ITU recommendation G.711.
- DMA operations with single-cycle overhead—each SPORT can automatically receive and transmit multiple buffers of memory data, one data word each DSP cycle.
- 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 the H.100 standard.

Serial Peripheral Interface (SPI) Ports

The DSP has two SPI-compatible ports, which enable the DSP to communicate with multiple SPI compatible devices. These ports are multiplexed with SPORT2, so either SPORT2 or the SPI ports are active depending on the state of the `OPMODE` pin or `OPMODE` bit. To change the mode, the pin can be changed during hardware or software reset, or the bit can be changed at runtime.

The SPI interface uses three pins for transferring data: two data pins (master output-slave input, `MOSIx`, and master input-slave output, `MISOx`) and a clock pin (serial clock, `SCKx`). Two SPI chip-select input pins ($\overline{\texttt{SPISSx}}$) let other SPI devices select the DSP, and fourteen SPI chip select output pins (`SPIxSEL7-1`) let the DSP select other SPI devices. The SPI select pins are re-configured programmable flag pins. Using these pins, the SPI ports provide a full duplex, synchronous serial interface, which supports both master and slave modes and multiple master environments.

Each SPI port's baud rate and clock phase/polarities are programmable, and each has an integrated DMA controller, configurable to support both transmit and receive data streams. The SPI's DMA controller can only service uni-directional accesses at any given time.

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

UART Port

The UART port provides a simplified UART interface to another peripheral or host. It performs full duplex, asynchronous transfers of serial data. The UART port supports two modes of operation:

- PIO (programmed I/O)
- DMA (direct memory access)

Programmable Flag (PFx) Pins

The ADSP-2191 has sixteen bi-directional, general-purpose I/O, programmable flag (`PF15-0`) pins. The `PF7-0` pins are dedicated to general-purpose I/O. The `PF15-8` pins serve either as general-purpose I/O pins (if the DSP is connected to an 8-bit external data bus) or serve as `DATA15-8` lines (if the DSP is connected to a 16-bit external data bus). The programmable flag pins have special functions for clock multiplier selection and for SPI port operation.

Low-Power Operation

The ADSP-2191 has four low-power options that significantly reduce the power dissipation when the device operates under standby conditions. To enter any of these modes, the DSP executes an `IDLE` instruction. The ADSP-2191 uses configuration of the bits in the `PLLCTL` register to select between the low-power modes as the DSP executes the `IDLE` instruction. Depending on the mode, an `IDLE` shuts off clocks to different parts of the DSP in the different modes. The low-power modes are:

- Idle
- Powerdown core
- Powerdown core/peripherals
- Powerdown all

Clock Signals

The ADSP-2191 can be clocked by a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. If a crystal oscillator is used, the crystal should be connected across the `CLKIN` and `XTAL` pins, with two capacitors connected. Capacitor values are dependent on crystal type and should be specified by the crystal manufacturer. A parallel-resonant, fundamental frequency, microprocessor-grade crystal should be used for this configuration.

If a buffered, shaped clock is used, this external clock connects to the DSP's `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. When an external clock is used, the `XTAL` input must be left unconnected.

The DSP provides a user programmable 1x to 31x multiplication of the input clock—including some fractional values—to support 128 external-to-internal (DSP core) clock ratios.

Booting Modes

The ADSP-2191 has seven mechanisms for automatically loading internal program memory after reset. The `BMODE2-0` pins, sampled during hardware reset, and three bits in the System Configuration (`SYSCR`) register implement these modes:

- Boot from 16-bit external memory
- Boot from 8-bit EPROM
- Boot from host
- Execute from 8-bit external memory (no boot)
- Boot from UART
- Boot from SPI 4 Kbits
- Boot from SPI 512 Kbits

JTAG Port

The JTAG port on the ADSP-2191 supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. Emulators use the JTAG port to monitor and control the DSP during emulation. Emulators using this port provide full-speed emulation with access to inspect and modify memory, registers, and processor stacks. JTAG-based emulation is non-intrusive and does not affect target system loading or timing.

Differences from Previous DSPs

This section identifies differences between the ADSP-219x DSPs and previous ADSP-2100 family DSPs: ADSP-210x, ADSP-211x, ADSP-217x, and ADSP-218x. The ADSP-219x preserves much of the core ADSP-2100 family architecture, while extending performance and functionality. For background information on previous ADSP-2100 family DSPs, see the *ADSP-2100 Family User's Manual*.

Chip enhancements also lead to some differences in the instruction sets between these DSPs. For more information, see the ADSP-219x *DSP Instruction Set Reference*.

This section describes the following differences:

- "Computational Units and Data Register File" on page 1-25
- "Arithmetic Status (ASTAT) Register Latency" on page 1-25
- "NORM and EXP Instruction Execution" on page 1-25
- "Shifter Result (SR) Register as Multiplier Dual Accumulator" on page 1-25
- "Shifter Exponent (SE) Register is Not Memory Accessible" on page 1-26
- "Software Condition (SWCOND) Register and Condition Code (CCODE) Register" on page 1-26
- "Unified Memory Space" on page 1-28
- "Data Memory Page (DMPG1 and DMPG2) Registers" on page 1-28
- "Data Address Generator (DAG) Addressing Modes" on page 1-28
- "Base Registers for Circular Buffers" on page 1-29
- "Program Sequencer, Instruction Pipeline, and Stacks" on page 1-30
- "Conditional Execution (Difference in Flag Input Support)" on page 1-30
- "Execution Latencies (Different for JUMP Instructions)" on page 1-31

Computational Units and Data Register File

The ADSP-219x DSP computational units differ from those on the ADSP-218x, because the ADSP-219x data registers act as a register file for unconditional, single-function instructions. In these instructions, any data register may be an input to any computational unit. For conditional and/or multifunction instructions, the ADSP-219x and ADSP-218x DSP families have the same data register usage restrictions — `AX` and `AY` for ALU, `MX` and `MY` for the multiplier, and `SI` for shifter inputs. For more information, see "Multifunction Computations" on page 2-64.

Arithmetic Status (ASTAT) Register Latency

The ADSP-219x `ASTAT` register has a one-cycle effect latency. This is discussed in "ALU Status Flags" on page 2-19.

NORM and EXP Instruction Execution

The ADSP-219x `NORM` and `EXP` instructions execute slightly differently from previous ADSP-218x DSPs. This issue is discussed in "Normalize, ALU Result Overflow" on page 2-48.

Shifter Result (SR) Register as Multiplier Dual Accumulator

The ADSP-219x architecture introduces a new 16-bit register in addition to the `SR0` and `SR1` registers, the combination of which comprise the 40-bit- wide `SR` register on ADSP-218x DSPs. This new register, called `SR2`, can be used in multiplier or shift operations (lower 8 bits) and as a full 16-bit-wide scratch register. As a result, the ADSP-219x DSP has two 40-bit-wide accumulators, `MR` and `SR`. The `SR` dual accumulator has replaced the multiplier feedback register `MF`, as shown in the following example:

Table 1-1. SR2 Register

ADSP-218x Instruction	ADSP-219x Instruction (Replacement)

`MF=MR+MX0*MY1(UU);`
`IF NOT MV MR=AR*MF;`
	`SR=MR+MX0*MY1(UU);`
	`IF NOT MV MR=AR*SR2;`

Shifter Exponent (SE) Register is Not Memory Accessible

The ADSP-218x DSPs use SE as a data or scratch register. The SE register of the ADSP-219x architecture is not accessible from the data or program memory buses. Therefore, the multifunction instructions of the ADSP-218x that use SE as a data or scratch register should use one of the data file registers (DREG) as a scratch register on the ADSP-219x DSP.

Table 1-2. SE is Not Memory Accessible

ADSP-218x Instruction	ADSP-219x Instruction (Replacement)

`SR=Lshift MR1(HI),`
`SE=DM(I6,M5);`
	`SR=Lshift MR1(HI),`
	`AX0=DM(I6,M5);`

Software Condition (SWCOND) Register and Condition Code (CCODE) Register

The ADSP-219x DSP changes support for the ALU signed (AS) condition and supports additional arithmetic and status condition testing with the Condition Code (CCODE) register and software condition (SWCOND) test. The two conditions are SWCOND and NOT SWCOND. The usage of the ADSP-219x's and most ADSP-218x's arithmetic conditions (EQ, NE, GE, GT, LE, LT, AV, Not AV, AC, Not AC, MV, Not MV) are compatible.

The new shifter overflow (SV) condition of the ADSP-219x architecture is a good example of how the CCODE register and SWCOND test work. The ADSP-219x DSP's Arithmetic Status (ASTAT) register contains a bit indicating the status of the shifter's result. The shifter is a computational unit that performs arithmetic or logical bitwise shifts on fields within a data register. The result of the operation goes into the Shifter Result (SR2, SR1, and SR0, which are combined into SR) register. If the result overflows the SR register, the shifter overflow (SV) bit in the ASTAT register records this overflow/underflow condition for the SR result register (0 = no overflow or underflow, 1 = overflow or underflow).

For the most part, bits (status condition indicators) in the ASTAT register correspond to condition codes that appear in conditional instructions. For example, the ALU zero (AZ) bit in ASTAT corresponds to the ALU result equals zero (EQ) condition and would be used in code like this:

```
IF EQ AR = AX0 + AY0;
/* if the ALU result (AR) register is zero, add AX0 and AY0 */
```

The SV status condition in the ASTAT bits does not correspond to a condition code that can be directly used in a conditional instruction. To test for this status condition, software selects a condition to test by loading a value into the Condition Code (CCODE) register and uses the software condition (SWCOND) condition code in the conditional instruction. The DSP code would look like this:

```
CCODE = 0x09; NOP;              // set CCODE for SV condition
IF SWCOND SR = MR0 * SR1 (UU);  // mult unsigned X and Y
```

The NOP instruction after loading the CCODE register accommodates the one-cycle effect latency of the CCODE register.

The ADSP-218x DSP supports two conditions to detect the sign of the ALU result. On the ADSP-219x, these two conditions (POS and NEG) are supported as AS and NOT AS conditions in the CCODE register. For more information on CCODE register values and SWCOND conditions, see "Conditional Sequencing" on page 3-41.

Unified Memory Space

The ADSP-219x architecture has a unified memory space with separate memory blocks to differentiate between 24- and 16-bit memory. In the unified memory, the term *program* or *data memory* only has semantic significance; the address determines the "PM" or "DM" functionality. It is best to revise any code with non-symbolic addressing in order to use the new tools.

Data Memory Page (DMPG1 and DMPG2) Registers

The ADSP-219x processor introduces a paged memory architecture that uses 16-bit DAG registers to access 64K pages. The 16-bit DAG registers correspond to the lower 16 bits of the DSP's address buses, which are 24-bit wide. To store the upper 8 bits of the 24-bit address, the ADSP-219x DSP architecture uses two additional registers, DMPG1 and DMPG2. DMPG1 and DMPG2 work with the DAG registers I0-I3 and I4-I7, respectively.

Data Address Generator (DAG) Addressing Modes

The ADSP-219x architecture provides additional flexibility over the ADSP-218x DSP family in DAG addressing modes:

- Pre-modify without update addressing in addition to the post-modify with update mode of the ADSP-218x instruction set:

```
DM(I0+M1)  = AR;   /* pre-modify syntax  */
DM(I0+=M1) = AR;   /* post-modify syntax */
```

- Pre-modify and post-modify with an 8-bit twos-complement immediate modify value instead of an M register:

```
AX0=PM(I5+-4);  /* pre-modify syntax (for modifier = -4)*/
AX0=PM(I5+=4);  /* post-modify syntax (for modifier = 4) */
```

- DAG modify with an 8-bit twos-complement immediate-modify value:

  ```
  MODIFY(I7+=0x24);
  ```

Base Registers for Circular Buffers

The ADSP-219x processor eliminates the existing hardware restriction of the ADSP-218x DSP architecture on a circular buffer starting address. ADSP-219x enables declaration of any number of circular buffers by designating `B0-B7` as the base registers for addressing circular buffers; these base registers are mapped to the "register" space on the core.

Program Sequencer, Instruction Pipeline, and Stacks

The ADSP-219x DSP core and inputs to the sequencer differ for various members of the ADSP-219x family DSPs. The main differences between the ADSP-218x and ADSP-219x sequencers are that the ADSP-219x sequencer has:

- A 6-stage instruction pipeline, which works with the sequencer's loop and PC stacks, conditional branching, interrupt processing, and instruction caching.
- A wider branch execution range, supporting:
  - — 13-bit, non-delayed or delayed relative conditional `JUMP`
  - — 16-bit, non-delayed or delayed relative unconditional `JUMP` or `CALL`
  - — Conditional non-delayed or delayed indirect `JUMP` or `CALL` with address pointed to by a DAG register
  - — 24-bit conditional non-delayed absolute long `JUMP` or `CALL`
- A narrowing of the `DO`/`UNTIL` termination conditions to counter expired (`CE`) and `FOREVER`.

Conditional Execution (Difference in Flag Input Support)

Unlike the ADSP-218x DSP family, ADSP-219x DSPs do not directly support a conditional `JUMP`/`CALL` based on flag input. Instead, the ADSP-219x supports this type of conditional execution with the `CCODE` register and `SWCOND` condition. For more information, see "Software Condition (SWCOND) Register and Condition Code (CCODE) Register" on page 1-26.

The ADSP-219x architecture has 16 programmable flag pins that can be configured as either inputs or outputs. The flags can be checked by reading the `FLAGS` register, or by using a software condition flag.

Table 1-3. Conditional Execution

ADSP-218x Instruction	ADSP-219x Instruction (Replacement)
`If Not FLAG_IN AR=MR0 And 8192;`	`SWCOND=0x03;` `If Not SWCOND AR=MR0 And 8192;`
	`IOPG = 0x06;` `AX0=IO(FLAGS);` `AX0=Tstbit 11 OF AX0;` `If EQ AR=MR0 And 8192;`

Execution Latencies (Different for JUMP Instructions)

The ADSP-219x processor has an instruction pipeline (unlike ADSP-218x DSPs) and branches execution for immediate `JUMP` and `CALL` instructions in four clock cycles if the branch is taken. To minimize branch latency, ADSP-219x programs can use the delayed branch option on jumps and calls, reducing branch latency by two cycles. This savings comes from execution of two instructions following the branch before the `JUMP/CALL` occurs.

Development Tools

The ADSP-219x is supported by VisualDSP++®, an easy-to-use project management environment, comprised of an Integrated Development Environment (IDE) and Debugger. VisualDSP++ lets you manage projects from start to finish from within a single, integrated interface. Because the project development and debug environments are integrated, you can move easily between editing, building, and debugging activities.

Flexible Project Management. The IDE provides flexible project management for the development of DSP applications. The IDE includes access to all the activities necessary to create and debug DSP projects. You can create or modify source files or view listing or map files with the IDE editor. This powerful editor is part of the IDE and includes multiple language syntax highlighting, OLE drag and drop, bookmarks, and standard editing operations such as undo/redo, find/replace, copy/paste/cut, and go to.

Also, the IDE includes access to the DSP C/C++ compiler, C run-time library, assembler, linker, loader, simulator, and splitter. You specify options for these tools through dialog boxes. These dialog boxes are easy to use and make configuring, changing, and managing your projects simple. The options you select control how the tools process inputs and generate outputs, and the options have a one-to-one correspondence to the tools' command-line switches. You can define these options once or modify them to meet changing development needs. You also can access the Tools from the operating system command line if you choose.

Greatly Reduced Debugging Time. The debugger has an easy-to-use, common interface for all processor simulators and emulators available through Analog Devices and third parties or custom developments. The debugger has many features that greatly reduce debugging time. You can view C source interspersed with the resulting assembly code. You can profile execution of a range of instructions in a program; set simulated watchpoints on hardware and software registers, program and data memory; and trace instruction execution and memory accesses. These features enable you to correct coding errors, identify bottlenecks, and examine DSP performance. You can use the custom register option to select any combination of registers to view in a single window. The debugger can also generate inputs, outputs, and interrupts so you can simulate real world application conditions.

Software Development Tools. Software development tools, which support the ADSP-219x DSP family, let you develop applications that take full advantage of the architecture, including shared memory and memory overlays. Software development tools include C/C++ compiler, C run-time library, DSP and math libraries, assembler, linker, loader, simulator, and splitter.

C/C++ Compiler & Assembler. The C/C++ compiler generates efficient code that is optimized for both code density and execution time. The C/C++ compiler allows you to include assembly language statements inline. Because of this, you can program in C and still use assembly for time-critical loops. You can also use pretested math, DSP, and C run-time library routines to help shorten your time to market. The ADSP-219x DSP family assembly language is based on an algebraic syntax that is easy to learn, program, and debug.

Linker & Loader. The linker provides flexible system definition through Linker Description Files (`.LDF`). In a single LDF, you can define different types of executables for a single or multiprocessor system. The linker resolves symbols over multiple executables, maximizes memory use, and easily shares common code among multiple processors. The loader supports creation of PROM, host, SPI, and UART boot images. The loader allows multiprocessor system configuration with smaller code and faster boot time.

3rd-Party Extensible. The VisualDSP++ environment enables third-party companies to add value using Analog Devices' published set of application programming interfaces (API). Third-party products—real-time operating systems, emulators, high-level language compilers, multiprocessor hardware —can interface seamlessly with VisualDSP++ thereby simplifying the tools integration task. VisualDSP++ follows the COM API format. Two API tools, Target Wizard and API Tester, are also available for use with the API set. These tools help speed the time-to-market for vendor products. Target Wizard builds the programming shell based on API features the vendor requires. The API tester exercises the individual features independently of VisualDSP++. Third parties can use a subset of these APIs that meet their application needs. The interfaces are fully supported and backward compatible.

Further details and ordering information are available in the VisualDSP++ development tools data sheet. This data sheet can be requested from any Analog Devices sales office or distributor.

2 COMPUTATIONAL UNITS

This chapter provides the following topics:

Overview

The DSP's computational units perform numeric processing for DSP algorithms. The three computational units are the arithmetic/logic unit (ALU), multiplier/accumulator (multiplier), and shifter. 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 in a single cycle.

The computational units handle different types of operations. The ALU performs arithmetic and logic operations. The multiplier does multiplication and executes multiply/add and multiply/subtract operations. The shifter executes logical shifts and arithmetic shifts. Also, the shifter can derive exponents.

Data flow paths through the computational units are arranged in parallel, as shown in Figure 2-1 on page 2-3. The output of any computational unit may serve as the input of any computational unit on the next instruction cycle. Data moving in and out of the computational units goes through a data register file, consisting of sixteen primary registers and sixteen secondary registers. Two ports on the register file connect to the PM and DM data buses, allowing data transfer between the computational units and memory.

The DSP'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. For information on the data registers, see "Data Register File" on page 2-61.

Figure 2-1 on page 2-3 provides a graphical guide to the other topics in this chapter. First, a description of the `MSTAT` register shows how to set rounding, data format, and other modes for the computational units. Next, an examination of each computational unit provides details on operation and a summary of computational instructions. Looking at inputs to the computational units, details on register files, and data buses identify how to flow data for computations. Finally, details on the DSP's advanced parallelism reveal how to take advantage of conditional and multifunction instructions.

The diagrams in Figure 2-1 on page 2-3 describe the relationship between the ADSP-219x data register file and computational units: multiplier, ALU, and shifter.

The ALU stores the computation results either AR or in AF, where only AR is part of the register file. The AF register is intended for intermediate ALU data store and has a dedicated feedback path to the ALU. It cannot be accessed by move instructions.

There are two 40-bit units, MR and SR, built by the 16-bit registers SR2, SR1, SR0 and MR2, MR1, MR0. The individual register may input to any computation unit, but grouped together they function as accumulators for the MAC unit (multiply and accumulate). SR also functions as a shifter result register.

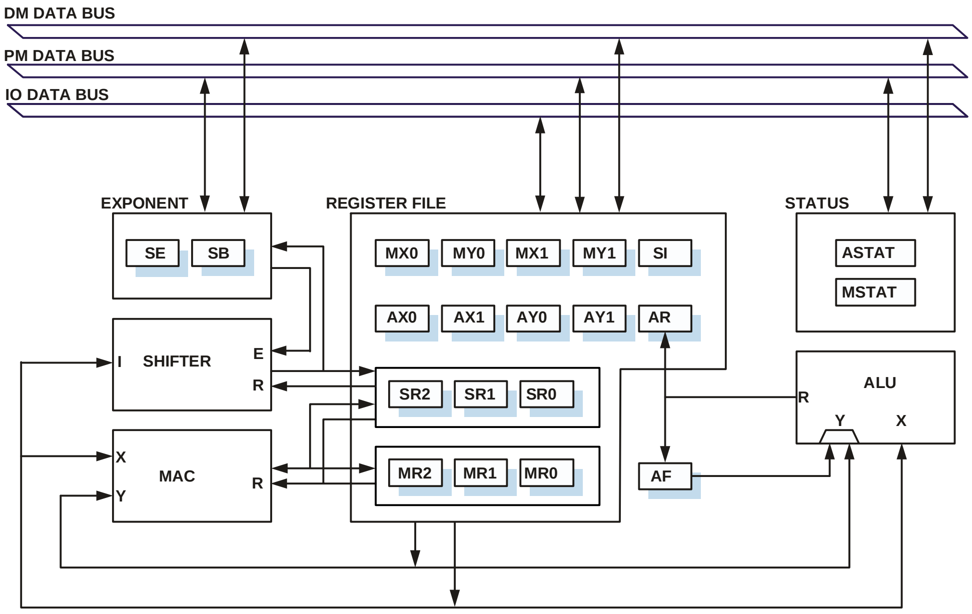


Figure 2-1. Register Access—Unconditional, Single-Function Instructions

Figure 2-1 on page 2-3 shows how unconditional, single-function multiplier, ALU, and shifter instructions have unrestricted access to the data registers in the register file. Due to opcode limitations, conditional and multi-function instructions provide ADSP-218x legacy register access only. Details are located in the corresponding sections.

The MR2 and SR2 registers differ from the other results registers. As a data register file register, MR2 and SR2 are 16-bit registers that may be X- or Y-inputs to the multiplier, ALU, or shifter. As result registers (part of MR or SR), only the lower 8-bits of MR2 or SR2 hold data (the upper 8-bits are sign extended). This difference (16-bits as input, 8-bits as output) influences how code can use the MR2 and SR2 registers. This sign extension appears in Figure 2-12 on page 2-32.

Using register-to-register move instructions, the data registers can load (or be loaded from) the Shifter Block (SB) and Shifter Exponent (SE) registers, but the SB and SE registers may not provide X- or Y-input to the computational units. The SB and SE registers serve as additional inputs to the shifter.

The MR2 and SR2 registers differ from the other results registers. As a data register file register, MR2 and SR2 are 16-bit registers that may be X- or Y-inputs to the multiplier, ALU, or shifter. As result registers (part of MR or SR), only the lower 8-bits of MR2 or SR2 hold data (the upper 8-bits are sign extended). This difference (16-bits as input, 8-bits as output) influences how code can use the MR2 and SR2 registers. This sign extension appears in Figure 2-12 on page 2-32.

Using register-to-register move instructions, the data registers can load (or be loaded from) the Shifter Block (SB) and Shifter Exponent (SE) registers, but the SB and SE registers may not provide X- or Y-input to the computational units. The SB and SE registers serve as additional inputs to the shifter.

The shaded boxes behind the data register file and the SB, SE, and AF registers indicate that secondary registers are available for these registers. There are two sets of data registers. Only one bank is accessible at a time. The additional bank of registers can be activated (such as during an interrupt service routine) for extremely fast context switching. A new task, like an interrupt service routine, can be executed without transferring current states to storage. For more information, see “Secondary (Alternate) Data Registers” on page 2-63.

The Mode Status (MSTAT) register input sets arithmetic modes for the computational units, and the Arithmetic Status (ASTAT) register records status/conditions for the computation operations’ results.

Data Formats

ADSP-219x DSPs are 16-bit, fixed-point machines. Most operations assume a twos complement number representation, while others assume unsigned numbers or simple binary strings. Special features support multiword arithmetic and block floating-point. For detailed information on each number format, see “Numeric Formats” on page C-1.

In ADSP-219x family arithmetic, signed numbers are always in twos complement format. These DSPs do not use signed magnitude, ones complement, BCD, or excess-n formats.

This section provides the following topics:

Binary String

This format is the least complex binary notation; sixteen bits are treated as a bit pattern. Examples of computations using this format are the logical operations: NOT, AND, OR, and 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, having nearly twice the magnitude of a signed number of the same length. The DSP treats the least significant words of multiple precision numbers as unsigned numbers.

Signed Numbers: Twos Complement

In ADSP-219x DSP arithmetic, the term “signed” refers to twos complement. Most ADSP-219x family operations presume or support twos complement arithmetic.

Signed Fractional Representation: 1.15

ADSP-219x DSP arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 (“one dot fifteen”). In the 1.15 format, there is one sign bit (the MSB) and fifteen fractional bits representing values from –1 up to one LSB less than +1.

Figure 2-2 on page 2-7 shows the bit weighting for 1.15 numbers. These are examples of 1.15 numbers and their decimal equivalents.

1.15 NUMBER (HEXADECIMAL)	DECIMAL EQUIVALENT
0X0001	0.000031
0X7FFF	0.999969
0XFFFF	–0.000031
0X8000	–1.000000

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

Figure 2-2. Bit Weighting for 1.15 Numbers

ALU Data Types

All operations on the ALU treat operands and results as 16-bit binary strings, except the signed division primitive (`DIVS`). ALU result status bits treat the results as signed, indicating status with the overflow (`AV`) condition code and the negative (`AN`) flag.

The logic of the overflow bit (AV) is based on twos complement arithmetic. It is set if the 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 generates a positive result; a change in the sign bit signifies an overflow and sets AV. Adding a negative and a positive may result in either a negative or positive result, but cannot overflow.

The logic of the carry bit (AC) is based on unsigned-magnitude arithmetic. It is set if a carry is generated from bit 16 (the MSB). The (AC) bit is most useful for the lower word portions of a multiword operation.

ALU results generate status information. For more information on using ALU status, see "ALU Status Flags" on page 2-19.

Except for division, the ALU operations do not need to distinguish between signed or unsigned, integer or fractional formats. Formats are a matter of result interpretation only.

Multiplier Data Types

The multiplier produces results that are binary strings. The inputs are "interpreted" according to the information given in the instruction itself (signed times signed, unsigned times unsigned, a mixture, or a rounding operation). The 32-bit result from the multiplier is assumed to be signed, in that it is sign-extended across the full 40-bit width of the MR or SR register set.

The ADSP-219x DSPs support two modes of format adjustment: fractional mode for fractional operands (1.15 format with 1 signed bit and 15 fractional bits) and integer mode for integer operands (16.0 format).

When the processor multiplies two 1.15 operands, the result is a 2.30 (two sign bits and 30 fractional bits) number. In fractional mode, the multiplier automatically shifts the multiplier product (P) left one bit before transferring the result to the multiplier result register (MR). This shift causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. This result format appears in Figure 2-3 on page 2-14.

In 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; it would change the numerical representation. This result format appears in Figure 2-4 on page 2-15.

Multiplier results generate status information. For more information on using multiplier status, see "Multiplier Status Flags" on page 2-34.

Shifter Data Types

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

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

Shifter results generate status information. For more information on using shifter status, see "Shifter Status Flags" on page 2-54.

Arithmetic Formats Summary

Table 2-1 on page 2-10, Table 2-2 on page 2-10, and Table 2-3 on page 2-11 summarize some of the arithmetic characteristics of computational operations.

Table 2-1. ALU Arithmetic Formats

Operation	Operands Formats	Result Formats
Addition	Signed or unsigned	Interpret flags
Subtraction	Signed or unsigned	Interpret flags
Logical Operations	Binary string	same as operands
Division	Explicitly signed/unsigned	same as operands
ALU Overflow	Signed	same as operands
ALU Carry Bit	16-bit unsigned	same as operands
ALU Saturation	Signed	same as operands

Table 2-2. Multiplier Arithmetic Formats

Operation (by Mode)	Operands Formats	Result Formats
Multiplier, Fractional Mode		
Multiplication (MR/SR)	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
Mult / Add	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
Mult / Subtract	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
Multiplier Saturation	Signed	same as operands
Multiplier, Integer Mode		
Multiplication (MR/SR)	16.0 Explicitly signed/unsigned	32.0 no shift
Mult / Add	16.0 Explicitly signed/unsigned	32.0 no shift
Mult / Subtract	16.0 Explicitly signed/unsigned	32.0 no shift
Multiplier Saturation	Signed	same as operands

Table 2-3. Shifter Arithmetic Formats

Operation	Operands Formats	Result Formats
Logical Shift	Unsigned / binary string	same as operands
Arithmetic Shift	Signed	same as operands
Exponent Detection	Signed	same as operands

Setting Computational Modes

The MSTAT and ICNTL registers control the operating mode of the computational units. Figure A-2 on page A-10 lists all the bits in MSTAT, and Figure A-5 on page A-16 lists all the bits in ICNTL. The following bits in MSTAT and ICNTL control computational modes:

- **ALU overflow latch mode.** MSTAT Bit 2 (AV_LATCH) determines how the ALU overflow flag, AV, gets cleared (0=AV is "not-sticky", 1=AV is "sticky").

- **ALU saturation mode.** MSTAT Bit 3 (AR_SAT) determines (for signed values) whether ALU AR results that overflowed or underflowed are saturated or not (0=unsaturated, 1=saturated).

- **Multiplier result mode.** MSTAT Bit 4 (M_MODE) selects fractional 1.15 format (=0) or integer 16.0 format (=1) for all multiplier operations. The multiplier adjusts the format of the result according to the selected mode.

- **Multiplier biased rounding mode.** ICNTL Bit 7 (BIASRND) selects unbiased (=0) or biased (=1) rounding for multiplier results.

This section provides the following topics:

Latching ALU Result Overflow Status

The DSP supports an ALU overflow latch mode with the AV_LATCH bit in the MSTAT register. This bit determines how the ALU overflow flag, AV, gets cleared.

If AV_LATCH is disabled (=0), the AV bit is "not-sticky". When an ALU overflow sets the AV bit in the ASTAT register, the AV bit only remains set until cleared by a subsequent ALU operation that does not generate an overflow (or is explicitly cleared).

If AV_LATCH is enabled (=1), the AV bit is "sticky". When an ALU overflow sets the AV bit in the ASTAT register, the AV bit remains set until the application explicitly clears it.

Saturating ALU Results on Overflow

The DSP supports an ALU saturation mode with the AR_SAT bit in the MSTAT register. This bit determines (for signed values) whether ALU AR results that overflowed or underflowed are saturated or not. This bit enables (if set, =1) or disables (if cleared, =0) saturation for all subsequent ALU operations. If AR_SAT is disabled, AR results remain unsaturated and is returned unchanged. If AR_SAT is enabled, AR results are saturated according to the state of the AV and AC status flags in ASTAT shown in Table 2-4 on page 2-13.

Table 2-4. ALU Result Saturation With AR_SAT Enabled

AV	AC	AR register
0	0	ALU output not saturated
0	1	ALU output not saturated
1	0	ALU output saturated, maximum positive 0x7FFF
1	1	ALU output saturated, maximum negative 0x8000

The AR_SAT bit in MSTAT only affects the AR register. Only the results written to the AR register are saturated. If results are written to the AF register, wraparound occurs, but the AV and AC flags reflect the saturated result.

Using Multiplier Integer and Fractional Formats

For multiply/accumulate functions, the DSP provides two modes: fractional mode for fractional numbers (1.15), and integer mode for integers (16.0).

In the fractional mode, the 32-bit product output is format adjusted—sign-extended and shifted one bit to the left—before being added to MR. For example, bit 31 of the product lines up with bit 32 of MR (which is bit 0 of MR2) and bit 0 of the product lines up with bit 1 of MR (which is bit 1 of MR0). The LSB is zero-filled. The fractional multiplier result format appears in Figure 2-3 on page 2-14.

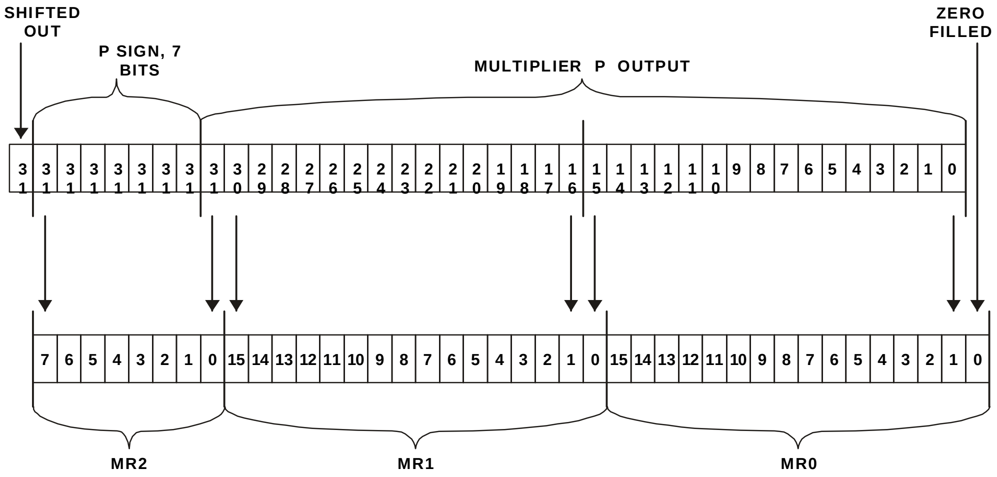


Figure 2-3. Fractional Multiplier Results Format

After adjustment the result of a 1.15 by 1.15 fractional multiplication is available in 1.31 format (`MR1:MR0` or `SR1:SR0`). If 32-bit precision is not required MR1 or SR1 hold the result in 1.15 data representation. MR2 and SR2 do not contain multiplication results. They are needed for accumulation only.

In integer mode, the 32-bit Product register is not shifted before being added to `MR`. Figure 2-4 on page 2-15 shows the integer-mode result placement. After a 16.0 by 16.0 multiplication `MR1:MR0` (`SR1:SR0`) hold the 32.0 result.

The mode is selected by the `M_MODE` bit in the Mode Status (`MSTAT`) register. If `M_MODE` is set (=1), integer mode is selected. If `M_MODE` is cleared (=0), fractional mode is selected. In either mode, 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 `MR` register to form the final 40-bit result.

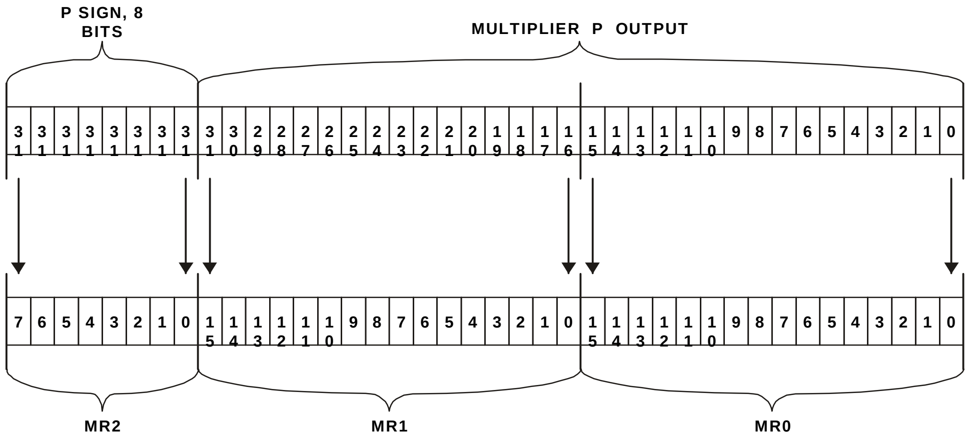


Figure 2-4. Integer Multiplier Results Format

Rounding Multiplier Results

The DSP supports multiplier results rounding (`RND` option) on most multiplier operations. With the `BIASRND` bit in the `ICNTL` register, programs select whether the `Rnd` option provides biased or unbiased rounding.

Unbiased Rounding

Unbiased rounding uses the multiplier's capability for rounding the 40-bit result at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. The rounded output is directed to either `MR` or `SR`. When rounding is selected, `MR1/SR1` contains the rounded 16-bit result; the rounding effect in `MR1/SR1` affects `MR2/SR2` as well. The `MR2/MR1` and `SR2/SR1` registers represent the rounded 24-bit result.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding is to add a 1 into bit position 15 of the adder chain. This method causes a net positive bias, because the midway value (when `MR0`=0x8000) is always rounded upward. The accumulator eliminates this bias by forcing bit 16 in the result output to zero when it detects this midway point. This has the effect of rounding odd `MR1` values upward and even `MR1` values downward, yielding a zero large-sample bias assuming uniformly distributed values.

Using x to represent any bit pattern (not all zeros), here are two examples of rounding. The example in Figure 2-5 on page 2-16 shows a typical rounding operation for `MR`; these also apply for `SR`.

```
                      ...MR2..|.......MR1......|.......MR0......
Unrounded value:──►xxxxxxxx|xxxxxxxx00100101|1xxxxxxxxxxxxxxx
Add 1 and carry:──►........|................|1...............
Rounded value:  ──►xxxxxxxx|xxxxxxxx00100110|0xxxxxxxxxxxxxxx
```

Figure 2-5. Typical Unbiased Multiplier Rounding Operation

The compensation to avoid net bias becomes visible when the lower 15 bits are all zero and bit 15 is one (the midpoint value) as shown in Figure 2-6 on page 2-16.

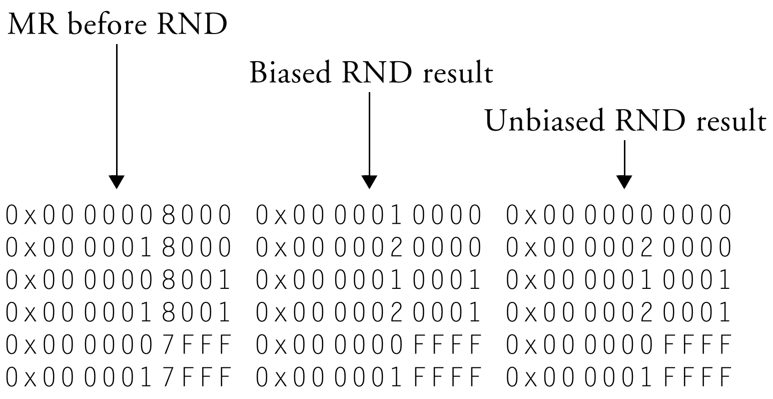
```
                      ...MR2..|.......MR1......|.......MR0......
Unrounded value:──►xxxxxxxx|xxxxxxxx01100110|1000000000000000
Add 1 and carry:──►........|................|1...............
MR bit 16=1:    ──►xxxxxxxx|xxxxxxxx01100111|0000000000000000
Rounded value:  ──►xxxxxxxx|xxxxxxxx01100110|0000000000000000
```

Figure 2-6. Avoiding Net Bias in Unbiased Multiplier Rounding Operation

In Figure 2-6 on page 2-16, MR bit 16 is forced to zero. This algorithm is employed on every rounding operation, but is only evident when the bit patterns shown in the lower 16 bits of the last example are present.

Biased Rounding

The BIASRND bit in the ICNTL register enables biased rounding. When the BIASRND bit is cleared (=0), the RND option in multiplier instructions uses the normal unbiased rounding operation (as discussed in "Unbiased Rounding" on page 2-15). When the BIASRND bit is set to 1, the DSP uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with MR0 set to 0x8000 round up, rather than only rounding odd MR1 values up. For an example, see Figure 2-7 on page 2-17.



```
0x00 0000 8000  0x00 0001 0000  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
```

Figure 2-7. Bias Rounding in Multiplier Operation

This mode only has an effect when the MR0 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 GSM speech compression routines. Unbiased rounding is preferred for most algorithms. Note that the content of MR0 and SR0 is invalid after rounding.

Using Computational Status

The multiplier, ALU, and shifter update overflow and other status flags in the DSP's Arithmetic Status (ASTAT) register. To use status conditions from computations in program sequencing, use conditional instructions to test the exception flags in the ASTAT register after the instruction executes. This method permits monitoring each instruction's outcome.

More information on ASTAT appears in the sections that describe the computational units. For summaries relating instructions and status bits, see "ALU Status Flags" on page 2-19, "Multiplier Status Flags" on page 2-34, and "Shifter Status Flags" on page 2-54.

Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-bit fixed-point operands and output 16-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction
- Fixed-point add with carry, subtract with borrow, increment, decrement
- Logical AND, OR, XOR, or NOT
- Functions: ABS, PASS, division primitives

This section provides the following topics:

ALU Operation

ALU instructions take one or two inputs: X input and Y input. For unconditional, single-function instructions, these inputs (also known as operands) can be any data registers in the register file. Most ALU operations return one result, but in PASS operations the ALU operation returns no result (only status flags are updated). ALU results are written to the ALU Result (AR) register or ALU Feedback (AF) register.

The DSP transfers input operands from the register file during the first half of the cycle and transfers results to the result register during the second half of the cycle. With this arrangement, the ALU can read and write the AR register file location in a single cycle.

ALU Status Flags

ALU operations update status flags in the DSP's Arithmetic Status (ASTAT) register. Table A-1 on page A-9 lists all the bits in this register. Table 2-5 on page 2-20 shows the bits in ASTAT that flag ALU status (a 1 indicates the condition is true) for the most recent ALU operation.

Table 2-5. ALU Status Bits in the ASTAT Register

Flag	Name	Definition
AZ	Zero	Logical NOR of all the bits in the ALU result register. True if ALU output equals zero.
AN	Negative	Sign bit of the ALU result. True if the ALU output is negative.
AV	Overflow	Exclusive-OR of the carry outputs of the two most significant adder stages. True if the ALU overflows.
AC	Carry	Carry output from the most significant adder stage.
AS	Sign	Sign bit of the ALU X input port. Affected only by the ABS instruction.
AQ	Quotient	Quotient bit generated only by the DIVS and DIVQ instructions.

Flag updates occur at the end of the cycle in which the status is generated and are available in the next cycle.

On previous 16-bit, fixed-point DSPs (ADSP-2100 family), the `POS` (`AS` bit =1) and `NEG` (`AS` bit =0) conditions permit checking the ALU result's sign. On ADSP-219x-based DSPs, the `CCODE` register and `SWCOND` condition support this feature.

Unlike previous ADSP-218x DSPs, `ASTAT` writes on ADSP-219x-based DSPs have a one cycle effect latency. Code being ported from ADSP-218x to ADSP-219x-based DSPs that check ALU status during the instruction following an `ASTAT` clear (`ASTAT`=0) instruction may not function as intended. Re-arranging the order of instructions to accommodate the one cycle effect latency on the ADSP-219x-based `ASTAT` register corrects this issue.

ALU Instruction Summary

Table 2-6 on page 2-21 lists the ALU instructions and describes how they relate to `ASTAT` flags. As indicated by the table, the ALU handles flags the same whether the result goes to the `AR` or `AF` registers. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In Table 2-6 on page 2-21, note the meaning of the following symbols:

- **Dreg, Dreg1, Dreg2** indicate any register file location
- **XOP, YOP** indicate any X- and Y-input registers, indicating a register usage restriction for conditional and/or multifunction instructions. For more information, see "Multifunction Computations" on page 2-64.
- * indicates the flag may be set or cleared, depending on results of instruction
- ** indicates the flag is cleared, regardless of the results of instruction
- – indicates no effect

Table 2-6. ALU Instruction Summary

Instruction	ASTAT Status Flags					
	AZ	AV	AN	AC	AS	AQ
|AR, AF| = Dreg1 + |Dreg2, Dreg2 + C, C |;	*	*	*	*	–	–
[IF Cond] |AR, AF| = Xop + |Yop, Yop + C, C, Const, Const + C|;	*	*	*	*	–	–
|AR, AF| = Dreg1 - |Dreg2, Dreg2 + C 1, +C -1|;	*	*	*	*	–	–
[IF Cond]|AR,AF| = Xop - |Yop,Yop+C-1,+C-1,Const,Const+C -1|;	*	*	*	*	–	–
|AR, AF| = Dreg2 - |Dreg1, Dreg1 + C -1|;	*	*	*	*	–	–
[IF Cond] |AR, AF| = Yop - |Xop, Xop+C-1|;	*	*	*	*	–	–
[IF Cond] |AR,AF| = - |Xop+C -1, Xop+Const, Xop+Const+C-1|;	*	*	*	*	–	–
|AR, AF| = Dreg1 |AND, OR, XOR| Dreg2;	*	**	*	**	–	–
[IF Cond] |AR, AF| = Xop |AND, OR, XOR| |Yop, Const|;	*	**	*	**	–	–
[IF Cond]|AR,AF| = |TSTBIT,SETBIT,CLRBIT,TGLBIT| n of Xop;	*	**	*	**	–	–
|AR, AF| = PASS |Dreg1, Dreg2, Const|;	*	**	*	**	–	–
|AR, AF| = PASS 0;	**	**	*	**	–	–

Table 2-6. ALU Instruction Summary (Cont'd)

Instruction	ASTAT Status Flags					
	AZ	AV	AN	AC	AS	AQ
[IF Cond] |AR, AF| = PASS |Xop, Yop, Const|;	*	**	*	**	–	–
|AR, AF| = NOT |Dreg|;	*	**	*	**	–	–
[IF Cond] |AR, AF| = NOT |Xop, Yop|;	*	**	*	**	–	–
|AR, AF| = ABS Dreg;	*	**	**	**	*	–
[IF Cond] |AR, AF| = ABS Xop;	*	**	**	**	*	–
|AR, AF| = Dreg +1;	*	*	*	*	–	–
[IF Cond] |AR, AF| = Yop +1;	*	*	*	*	–	–
|AR, AF| = Dreg -1;	*	*	*	*	–	–
[IF Cond] |AR, AF| = Yop -1;	*	*	*	*	–	–
DIVS Yop, Xop;	–	–	–	–	–	*
DIVQ Xop;	–	–	–	–	–	*

ALU Data Flow Details

Figure 2-8 shows a more detailed diagram of the ALU, which appears in Figure 2-1 on page 2-3.

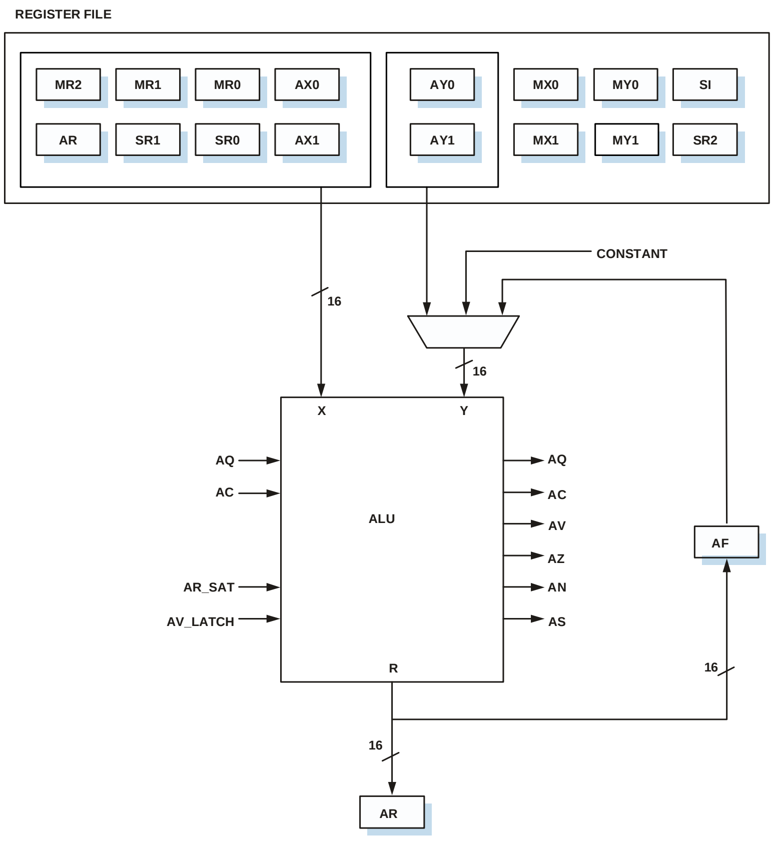


Figure 2-8. ALU Block Diagram

The ALU is 16 bits wide with two 16-bit input ports, X and Y, and one output port, R. The ALU accepts a carry-in signal (CI) which is the carry bit (AC) from the processor arithmetic status register (ASTAT). The ALU generates six status signals:

- Zero (AZ)
- Negative (AN)
- Carry (AC)
- Overflow (AV)
- X-input sign (AS)
- Quotient (AQ)

All arithmetic status signals are latched into the Arithmetic Status (ASTAT) register at the end of the cycle. For information on how each instruction affects the ALU flags, see Table 2-6 on page 2-21.

Unless a NONE= instruction is executed, the output of the ALU goes into the ALU Feedback (AF) register or the ALU Result (AR) register, which is part of the register file. The AF register is an ALU internal register.

In unconditional and single-function instructions, both the X and the Y port may read any register of the register file including AR. Alternatively, the Y port may access the ALU Feedback (AF) register.

For conditional and multi-function instructions only, a subset of registers can be used as input operands. For legacy support this register usage restriction mirrors the ADSP-218x instruction set. Then the X port can access the AR, SR1, SR0, MR2, MR1, MR0, AX0, and AX1 registers. The Y port accesses AY0, AY1, and AF.

If the X port accesses AR, SR1, SR0, MR2, MR1, MR0, AX0, or AX1, the Y operator may be a constant coded in the instruction word.

For more information on register usage restrictions in conditional and multifunction instructions, see "Multifunction Computations" on page 2-64.

The ALU can read and write any of its associated registers in the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. A register read gets the value loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This read/write pattern lets an input register provide an operand to the ALU at the beginning of the cycle and be updated with the next operand from memory at the end of the same cycle. Also, this read/write pattern lets a result register be stored in memory and updated with a new result in the same cycle.

Multiprecision operations are supported in the ALU with the carry-in signal and ALU carry (AC) status bit. The carry-in signal is the AC status bit that was generated by a previous ALU operation. The "add with carry" (+C) operation is intended for adding the upper portions of multiprecision numbers. The "subtract with borrow" (C–1 is effectively a "borrow") operation is intended for subtracting the upper portions of multiprecision numbers.

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), add-subtract division algorithm. The division can be either signed or unsigned, but the dividend and divisor must both be of the same type. More details on using division and programming examples are available in the *ADSP-219x DSP Instruction Set Reference*.

A single-precision divide, with a 32-bit dividend (numerator) and a 16-bit divisor (denominator), yielding a 16-bit quotient, executes in 16 cycles. Higher- and lower-precision quotients can also be calculated. The divisor can be stored in AX0, AX1, or any of the R registers. The upper half of a signed dividend can start in either AY1 or AF. The upper half of an unsigned dividend must be in AF. The lower half of any dividend must be in AY0. At the end of the divide operation, the quotient is in AY0.

The first of the two primitive instructions "divide-sign" (DIVS) is executed at the beginning of the division when dividing signed numbers. This operation computes the sign bit of the quotient by performing an exclusive OR of the sign bits of the divisor and the dividend. The AY0 register is shifted one place so that the computed sign bit is moved into the LSB position. The computed sign bit is also loaded into the AQ bit of the arithmetic status register. The MSB of AY0 shifts into the LSB position of AF, and the upper 15 bits of AF are loaded with the lower 15 R bits from the ALU, which simply passes the Y input value straight through to the R output. The net effect is to left shift the AF-AY0 register pair and move the quotient sign bit into the LSB position. The operation of Divs is illustrated in Figure 2-9 on page 2-27.

When dividing unsigned numbers, the DIVS operation is not used. Instead, the AQ bit in the arithmetic status register (ASTAT) should be initialized to zero by manually clearing it. The AQ bit indicates to the following operations that the quotient should be assumed positive.

The second division primitive is the "divide-quotient" (DIVQ) instruction, which generates one bit of quotient at a time and is executed repeatedly to compute the remaining quotient bits.

For unsigned single-precision divides, the Divq instruction is executed 16 times to produce 16 quotient bits. For signed single-precision divides, the DIVQ instruction is executed 15 times after the sign bit is computed by the DIVS operation. DIVQ instruction shifts the AY0 register left by one bit so that the new quotient bit can be moved into the LSB position.

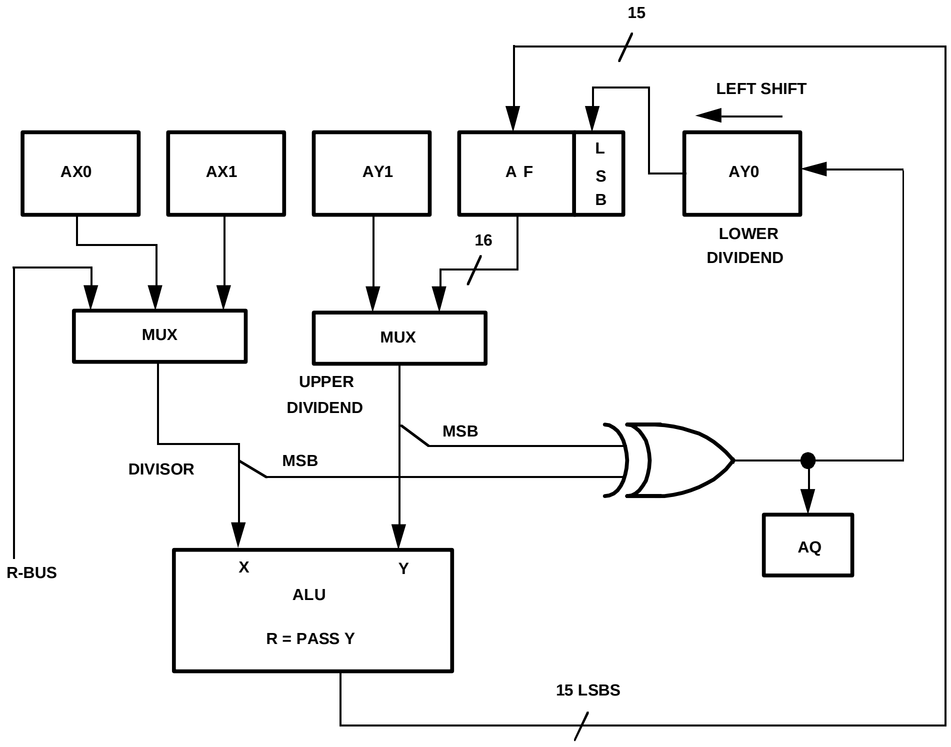


Figure 2-9. DIVS Operation

The status of the AQ bit generated from the previous operation determines the ALU operation to calculate the partial remainder. If AQ = 1, the ALU adds the divisor to the partial remainder in AF. If AQ = 0, the ALU subtracts the divisor from the partial remainder in AF.

The ALU output R is offset loaded into AF just as with the Divs operation. The AQ bit is computed as the exclusive-OR of the divisor MSB and the ALU output MSB, and the quotient bit is this value inverted. The quotient bit is loaded into the LSB of the AY0 register which is also shifted left by one bit. The DIVQ operation is illustrated in Figure 2-10 on page 2-28.

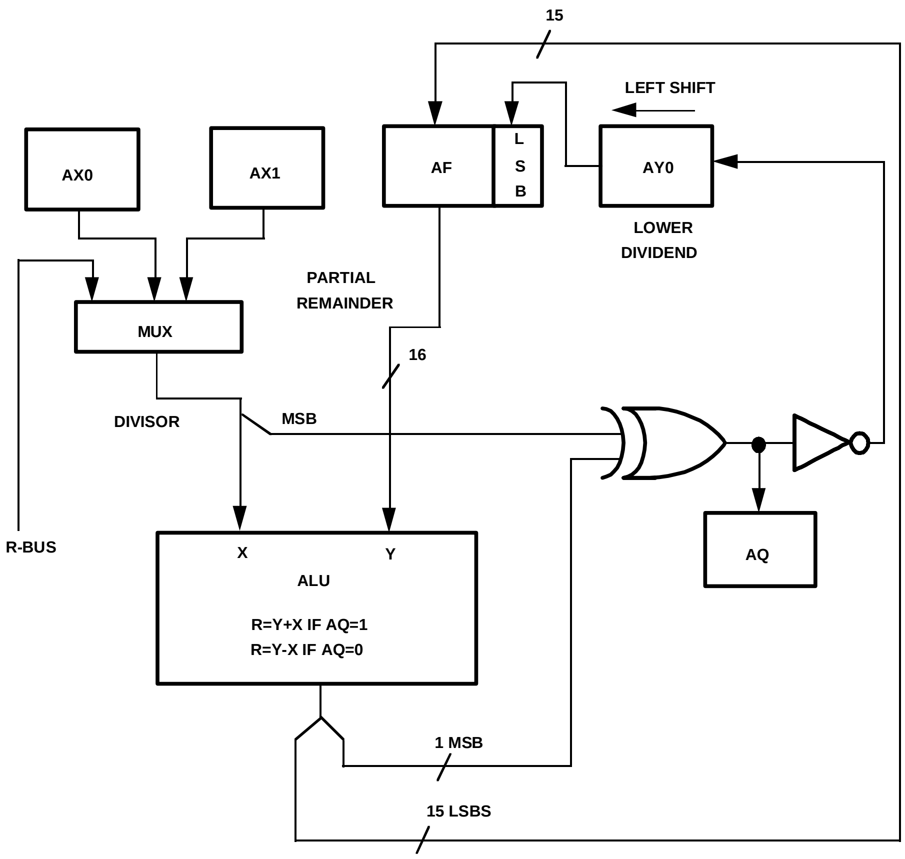


Figure 2-10. DIVQ Operation

The format of the quotient for any numeric representation can be determined by the format of the dividend and divisor as shown in Figure 2-11 on page 2-29. Let NL represent the number of bits to the left of the binary point, let NR represent the number of bits to the right of the binary point of the dividend, let DL represent the number of bits to the left of the binary point, and let DR represent the number of bits to the right of the binary point of the divisor. Then, the quotient has NL–DL+1 bits to the left of the binary point and has NR–DR–1 bits to the right of the binary point.

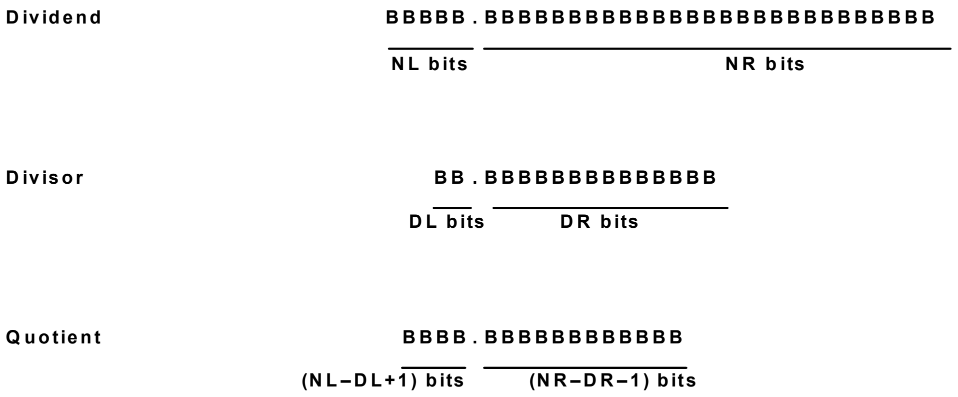


Figure 2-11. Quotient Format

Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format) the result is fully fractional (in 1.15 format), and the dividend must be smaller than the divisor for a valid result.

To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), the program must shift the dividend one bit to the left (into 31.1 format) before dividing. Additional discussion and code examples can be found in the ADSP-219x *DSP Instruction Set Reference*.

The algorithm overflows if the result cannot be represented in the format of the quotient as calculated in Figure 2-11 on page 2-29 or when the divisor is zero or less than the dividend in magnitude. For additional information see the section "Divide Primitives: `DIVS` and `DIVQ`" in the *ADSP-219x DSP Instruction Set Reference*.

Multiply/Accumulates (Multiplier)

The multiplier performs fixed-point multiplication and multiply/accumulate operations. Multiply/accumulates are available with either cumulative addition or cumulative subtraction. Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 40-bit results. Inputs are treated as fractional or integer, unsigned or twos complement. Multiplier instructions include:

- Multiplication
- Multiply/accumulate with addition, rounding optional
- Multiply/accumulate with subtraction, rounding optional
- Rounding, saturating, or clearing result register

This section provides the following topics:

Multiplier Operation

The multiplier takes two inputs: X input and Y input. For unconditional, single-function instructions, these inputs (also known as operands) can be any data registers in the register file. The multiplier accumulates results in either the Multiplier Result (`MR`) register or Shifter Result (`SR`) register. The results can also be rounded or saturated.

On previous 16-bit, fixed-point DSPs (ADSP-2100 family), only the Multiplier Result (`MR`) register can accumulate results for the multiplier. On ADSP-219x DSPs, both `MR` and `SR` registers can accumulate multiplier results.

The multiplier transfers input operands during the first half of the cycle and transfers results during the second half of the cycle. With this arrangement, the multiplier can read and write the same result register in a single cycle.

Depending on the multiplier mode (`M_MODE`) setting, operands are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. Each operand may be either an unsigned or a twos complement value. If inputs are fractional, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Multiplier instruction options (required within the multiplier instruction) specify inputs' data format(s)—`SS` for signed, `UU` for unsigned, `SU` for signed X-input and unsigned Y-input, and `US` for unsigned X-input and signed Y-input.

In fractional mode the multiplier expects data in 1.15 format (`SS`). The primary intention of the (`UU`), (`SU`), and (`US`) options is to enable multi-precision multiplication such as 1.31 by 1.31. Therefore all multiplication types perform an implicit left shift in fractional mode.

Placing Multiplier Results in the MR or SR Registers

As shown in Figure 2-12 on page 2-32, the `MR` register is divided into three sections: `MR0` (bits 0-15), `MR1` (bits 16-31), and `MR2` (bits 32-39). Similarly, the `SR` register is divided into three sections: `SR0` (bits 0-15), `SR1` (bits 16-31), and `SR2` (bits 32-39). Each of these registers is part of the register file

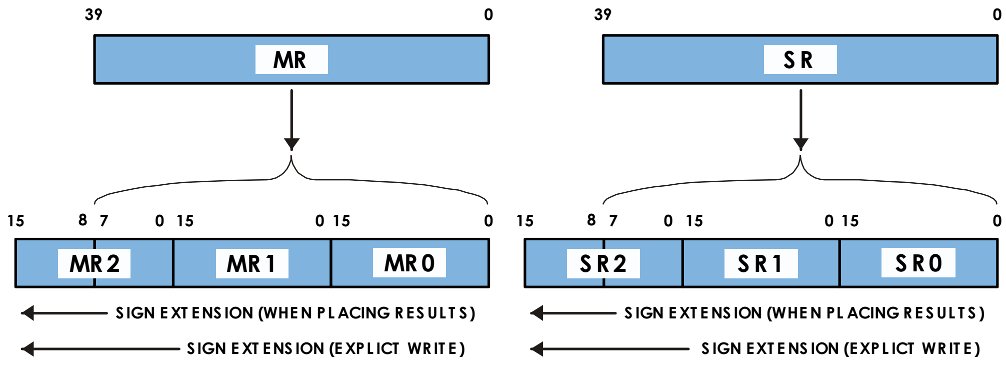

39
0
MR
39
0
SR
15
8
7
0
15
0
15
0
MR2
MR1
MR0
SR2
SR1
SR0
SIGN EXTENSION (WHEN PLACING RESULTS)
SIGN EXTENSION (EXPLICT WRITE)

Figure 2-12. Placing Multiplier Results

When the multiplier writes to either of the result registers, the 40-bit result goes into the lower 40 bits of the combined register (`MR2`, `MR1`, and `MR0` or `SR2`, `SR1`, and `SR0`), and the MSB is sign-extended into the upper eight bits of the uppermost register (`MR2` or `SR2`). When an instruction explicitly loads the middle result register (MR1 or SR1), the DSP also sign-extends the MSB of the data into the related uppermost register (MR2 or SR2). These sign extension operations appear in Figure 2-12 on page 2-32.

To load the MR2 register with a value other than MR1's sign extension, programs must load MR2 after MR1 has been loaded. Loading MR0 affects neither MR1 nor MR2; no sign extension occurs in MR0 loads. This technique also applies to SR2, SR1, and SR0.

Clearing, Rounding, or Saturating Multiplier Results

Besides using the results registers to accumulate, the multiplier also can clear, round, or saturate result data in the results registers. These operations work as follows:

- The clear operation—[MR,SR]=0—clears the specified result register to zero.
- The rounding operation—[MR,SR]=RND [MR,SR]—applies only to fractional results—integer results are not affected. This explicit rounding operation generates the same results as using the Rnd option in other multiplier instructions. For more information, see "Rounding Multiplier Results" on page 2-15.
- The saturate operation—SAT [MR,SR]—sets the specified result register to the maximum positive or negative value if an overflow or underflow has occurred. The saturation operation depends on the overflow status bit (MV or SV) and the MSB of the corresponding result register (MR2 or SR2). For more information, see "Saturating Multiplier Results on Overflow" on page 2-34.

Multiplier Status Flags

Multiplier operations update two status flags in the computational unit's arithmetic status register (ASTAT). Table A-1 on page A-9 lists all the bits in these registers. The following bits in ASTAT flag multiplier status (a 1 indicates the condition) for the most recent multiplier operation:

- **Multiplier overflow.** Bit 6 (MV) records an overflow/underflow condition for MR result register. If cleared (=0), no overflow or underflow has occurred. If set (=1), an overflow or underflow has occurred.
- **Shifter overflow.** Bit 8 (SV) records an overflow/underflow condition for SR result register. If cleared (=0) no overflow or underflow has occurred. If set (=1), an overflow or underflow has occurred.

Flag updates occur at the end of the cycle in which the status is generated and are available on the next cycle. The MV overflow flags are not updated if the individual 16-bit registers are loaded by move instructions. In such cases the proper update of MV can be forced with the pseudo instruction MR=MR;. The assembler translates that into MR=MR+MX0*0(SS); opcode. Similarly, use SR=SR; to update SV.

Saturating Multiplier Results on Overflow

The adder/subtracter generates overflow status signal every time a multiplier operation is executed. When the accumulator result in MR or SR interpreted as a twos complement number crosses the 32-bit (MR1/MR2) boundary (overflows), the multiplier sets the MV or SV bit in the ASTAT register.

The multiplier saturation instruction provides control over a multiplication result that has overflowed or underflowed. It saturates the value in the specified register only for the cycle in which it executes. It does not enable a mode that continuously saturates results until disabled like the ALU, because accumulation of saturated values mathematically returns erroneous results. Used at the end of a series of multiply and accumulate operations, the saturation instruction prevents the algorithm from post-processing overflowed results, when reading MR1 (SR1) without caring about MR2 (SR2).

For every operation it performs, the multiplier generates an overflow status signal MV (SV when SR is the specified result register), which is recorded in the ASTAT status register. The multiplier sets MV = 1 when the upper-nine bits in MR are anything other than all 0s or all 1s, setting MV when the accumulator result—interpreted as a signed, twos complement number—crosses the 32-bit boundary and spills over from MR1 into MR2. Otherwise, the multiplier clears MV = 0.

The explicit saturation instructions SAT MR; and SAT SR; evaluate the content of the 40-bit MR and SR registers rather than just testing the overflow flags MV and SV. The instructions examine whether the nine MSBs of MR/SR are all 0 or all 1. If no overflow occurred (all nine MSBs equal 0 or 1), the instructions do not alter the MR1/SR1 and MR0/SR0 registers, but the eight MSBs of MR2/SR2 are signed-extended.

If the SAT MR/SR; instructions detect an overflow (any of the nine MSBs of the 40-bit accumulator differs from the others) bit 7 of MR2/SR2 is used to determine whether an overflow or an underflow occurred. If this bit is zero, MR2/SR2 is set to 0x0000, MR1/SR1 to 0x7FFF and MR0/SR0 to 0xFFFF, representing the maximum positive 32-bit value. If this bit reads one, MR2/SR2 is set to 0xFFFF, MR1/SR1 to 0x8000 and MR0/SR0 to 0x0000, representing the maximum negative 32-bit value.

Avoid result overflows beyond the MSB of the result register. In such a case, the true sign bit of the result is irretrievably lost, and saturation may not produce a correct result. It takes over 255 overflows to lose the sign.

Multiplier Instruction Summary

Table 2-7 on page 2-36 lists the multiplier instructions and how they relate to ASTAT flags. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In Table 2-7 on page 2-36, note the meaning of the following symbols:

- **Dreg1, Dreg2** indicate any register file location
- **XOP, YOP** indicate any X- and Y-input registers, indicating a register usage restriction for conditional and/or multifunction instructions. For more information, see "Multifunction Computations" on page 2-64.
- * indicates the flag may be set or cleared, depending on results of instruction
- ** indicates the flag is cleared, regardless of the results of instruction
- – indicates no effect

Table 2-7. Multiplier Instruction Summary

Instruction	**ASTAT Status Flags**	
	MV	**SV**
|MR, SR| = Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond] |MR, SR| = Xop * Yop [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond] |MR, SR| = Yop * Xop [(|RND, SS, SU, US, UU|)];	*	*
|MR, SR| = |MR, SR| + Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond]|MR, SR| = |MR, SR| + Xop * Yop [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond] |MR, SR| = |MR, SR| + Yop * Xop [(|RND, SS, SU, US, UU|)];	*	*
|MR, SR| = |MR, SR| - Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond] |MR, SR| = |MR, SR| - Xop * Yop [(|RND, SS, SU, US, UU|)];	*	*
[IF Cond] |MR, SR| = |MR, SR| - Yop * Xop [(|RND, SS, SU, US, UU|)];	*	*

Table 2-7. Multiplier Instruction Summary (Cont'd)

Instruction	ASTAT Status Flags	
	MV	SV
[IF Cond] |MR, SR| = 0;	**	**
[IF Cond] MR = MR [(RND)];	*	–
[IF Cond] SR = SR [(RND)];	–	*
SAT [MR,SR];	–	–

Multiplier Data Flow Details

Figure 2-13 shows a more detailed diagram of the multiplier/accumulator, which appears in Figure 2-1 on page 2-3.

The multiplier has two 16-bit input ports X and Y, and a 32-bit product output port Product. The 32-bit product is stored in the multiplier result (MR or SR) register immediately, or passed to a 40-bit adder/subtracter, which adds or subtracts the new product to/from the previous content of the MR or SR registers. For accumulation, the MR and SR registers are 40 bits wide. These registers each consist of smaller 16-bit registers which are part of the register file: MR0, MR1, MR2, SR0, SR1, and SR2. For more information on these registers, see Figure 2-12 on page 2-32.

The adder/subtracters are greater than 32 bits to allow for intermediate overflow in a series of multiply/accumulate operations. A multiply overflow (MV or SV) status bit is set when an accumulator has overflowed beyond the 32-bit boundary—when there are significant (non-sign) bits in the top nine bits of the MR or SR registers (based on twos complement arithmetic).

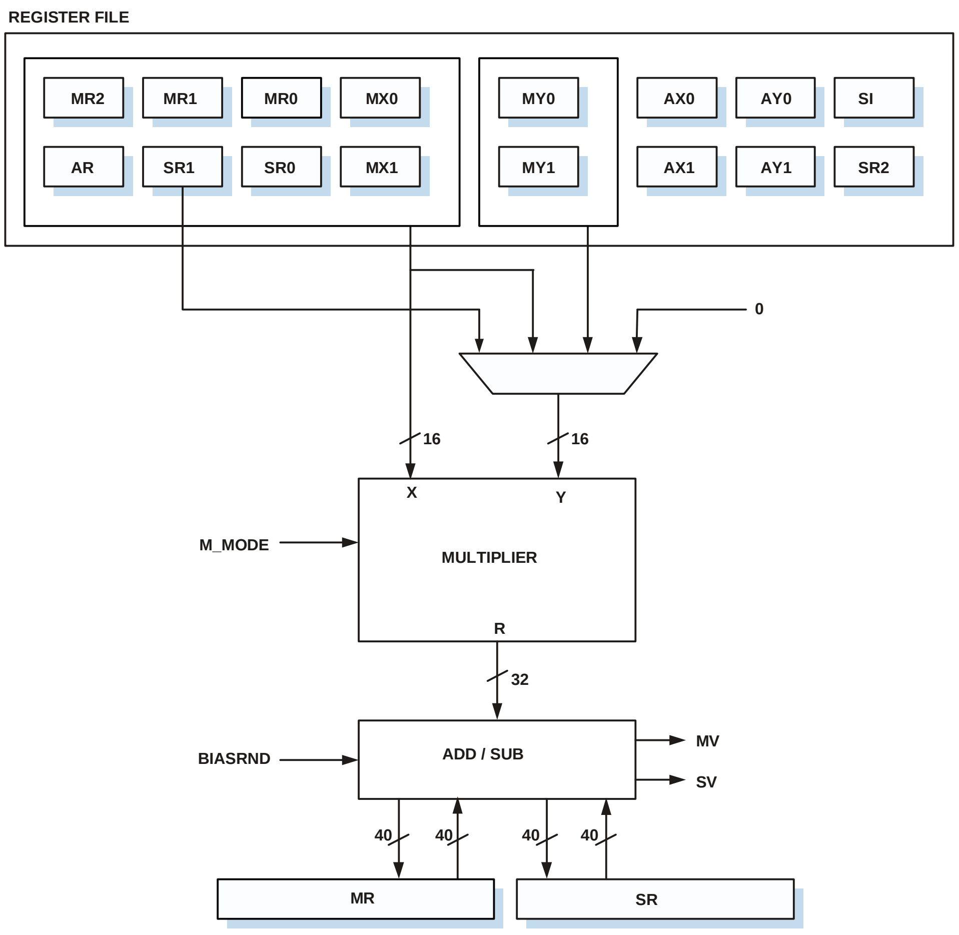


Figure 2-13. Multiplier Block Diagram

Register usage restrictions apply only to conditional and multi-function instructions. Then the multiplier's X port can read the `MX0`, `MX1`, `AR`, `MR2`, `MR1`, `MR0`, `SR1`, and `SR2` registers, and the Y port can read `MY0`, `MY1`, and `SR1` (due to a special pipe). The Y port can also be redirected to the X port to square a single X operand.

ⓘ On previous 16-bit, fixed-point DSPs (ADSP-2100 family), a dedicated Multiplier Feedback (`MF`) register is available. On ADSP-219x DSPs, there is no `MF` register, instead code should use `SR1`.

ⓘ For more information on register usage restrictions in conditional and multifunction instructions, see "Multifunction Computations" on page 2-64.

The multiplier reads and writes any of its associated registers within the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. A register read gets the value loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This read/write pattern lets an input register provide an operand to the multiplier at the beginning of the cycle and be updated with the next operand from memory at the end of the same cycle. This pattern also lets a result register be stored in memory and updated with a new result in the same cycle.

Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16-bit inputs, yielding a 40-bit output (`SR`). These functions include arithmetic shift (`ASHIFT`), logical shift (`LSHIFT`), and normalization (`NORM`). The shifter also performs derivation of exponent (`EXP`) and derivation of common exponent (`EXPADJ`) for an entire block of numbers. These shift functions can be combined to implement numerical format control, including full floating-point representation and multiprecision operations.

This section provides the following topics:

Shifter Operations

The shifter instructions (`ASHIFT`, `LSHIFT`, `NORM`, `EXP`, and `EXPADJ`) can be used in a variety of ways, depending on the underlying arithmetic requirements. The following sections present single- and multiple-precision examples for these functions:

The shift functions (arithmetic shift, logical shift, and normalize) can be optionally specified with `[SR OR]` to facilitate multiprecision operations. `[SR OR]` logically ORs the shift result with the current contents of `SR`. This option is used to join 16-bit inputs with the 40-bit value in `SR`. When `[SR OR]` is not used, the shift value is passed through to `SR` directly.

Almost all shifter instructions have two or three options: `(HI)`, `(LO)`, and `(HIX)`. Each option enables a different exponent detector mode that operates only while the instruction executes. The shifter interprets and handles the input data according to the selected mode.

For the derive exponent (Exp) and block exponent adjust (EXPADJ) operations, the shifter calculates the shift code—the direction and number of bits to shift—then stores the value in SE (for EXP) or SB (for EXPADJ). For the ASHIFT, LSHIFT, and NORM operations, a program can supply the value of the shift code directly to the SE register or use the result of a previous EXP or EXPADJ operation.

For the ASHIFT, LSHIFT, and NORM operations:

- (HI) Operation references the upper half of the output field.
- (LO) Operation references the lower half of the output field.

For the exponent derive (EXP) operation:

- (HIX) Use this mode for shifts and normalization of results from ALU operations.

  Input data is the result of an add or subtract operation that may have overflowed. The shifter examines the ALU overflow bit (AV). If AV=1, the effective exponent of the input is +1 (this value indicates that overflowed occurred before the EXP operation executed). If AV=0, no overflow occurred and the shifter performs the same operations as the (HI) mode.
- (HI) Input data is a single-precision signed number or the upper half of a double-precision signed number. The number of leading sign bits in the input operand, which equals the number of sign bits minus one, determines the shift code. By default, the EXPADJ operation always operates in this mode.
- (LO) Input data is the lower half of a double-precision signed number. To derive the exponent on a double-precision number, the program must perform the EXP operation twice, once on the upper half of the input, and once on the lower half.

Derive Block Exponent

The EXPADJ instruction detects the exponent of the number largest in magnitude in an array of numbers. The steps for a typical block exponent derivation are as follows:

1. **Load SB with –16.** The SB register contains the exponent for the entire block. The possible values at the conclusion of a series of EXPADJ operations range from –15 to 0. The exponent compare logic updates the SB register if the new value is greater than the current value. Loading the register with –16 initializes it to a value certain to be less than any actual exponents detected.

2. **Process the first array element** as follows:

   Array(1) = 11110101 10110001
   Exponent = –3
   –3 > SB (–16)
   SB gets –3

3. **Process next array element** as follows:

   Array(2) = 00000001 01110110
   Exponent = –6
   –6 < –3
   SB remains –3

4. Continue processing array elements.

When and if an array element is found whose exponent is greater than SB, that value is loaded into SB. When all array elements have been processed, the SB register contains the exponent of the largest number in the entire block. No normalization is performed. Expadj is purely an inspection operation. The value in SB could be transferred to SE and used to normalize the block on the next pass through the shifter. Or, SB could be associated with that data for subsequent interpretation.

Immediate Shifts

An immediate shift 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. For examples using this instruction, see the *ADSP-219x DSP Instruction Set Reference*. The data value controlling the shift is an 8-bit signed number. The `SE` register is not used or changed by an immediate shift.

The following example shows the input value downshifted relative to the upper half of `SR` (`SR1`). This is the (`HI`) version of the shift:

```
SI = 0xB6A3;
SR = LSHIFT SI BY -5 (HI);
```

Input (`SI`) is: `1011 0110 1010 0011`

Shift value is: –5

`SR` (shifted by):

```
0000 0000 0000 0101 1011 0101 0001 1000 0000 0000

---sr2---|-------sr1---------|-------sr0---------|
```

The next example uses the same input value, but shifts in the other direction, referenced to the lower half (`LO`) of `SR`:

```
SI = 0xB6A3;

SR = LSHIFT SI BY 5 (LO);
```

Input (`SI`) is: `1011 0110 1010 0011`

Shift value: is: +5

SR is shifted by:

```
0000 0000 0000 0000 0001 0110 1101 0100 0110 0000
---sr2---|-------sr1---------|-------sr0---------|
```

Note that a negative shift cannot place data (except a sign extension) into SR2, but a positive shift with value greater than 16 puts data into SR2. This next example also sets the SV bit (because the MSB of SR1 does not match the value in SR2):

```
SI = 0xB6A3;
```

```
SR = LSHIFT SI BY 17 (LO);
```

Input (SI) is: `1011 0110 1010 0011`

Shift value: +17

SR (shifted by):

```
0000 0001 0110 1101 0100 0110 0000 0000 0000 0000
---sr2---|--------sr1--------|--------sr0--------|
```

In addition to the direction of the shifting operation, the shift may be arithmetic (ASHIFT) or logical (LSHIFT).

The following example shows a logical shift, relative to the upper half of SR (HI):

```
SI = 0xB6A3;
```

```
SR = LSHIFT SI BY -5 (HI);
```

Input (SI): `10110110 10100011`

Shift value: -5

SR (shifted by):

```
0000 0000 0000 0101 1011 0101 0001 1000 0000 0000
---sr2---|-------sr1---------|------sr0----------
```

The next example uses the same input value, but performs an arithmetic shift:

```
SI = 0xB6A3;

SR = ASHIFT SI BY -5 (HI);
```

Input (SI): `10110110 10100011`

```
Shift value:-5
```

SR (shifted by):

```
1111 1111 1111 1101 1011 0101 0001 1000 0000 0000
---sr2---|-------sr1---------|-------sr0---------
```

Denormalize

Denormalizing refers to shifting a number according to a predefined exponent. In effect, the operation performs a floating-point to fixed-point conversion.

Denormalizing requires a sequence of operations. First, the SE register must contain the exponent value. This value may be explicitly loaded or may be the result of a previous operation. Next, the shift itself is performed, taking its shift value from the SE register, not from an immediate data value.

Two examples of denormalizing a double-precision number follow. The first example shows a denormalization in which the upper half of the number is shifted first, followed by the lower half. Because computations may produce output in either order, the second example shows the same operation in the other order—lower half first.

The following de-normalization example processes the upper half first. Some important points here are: (1) always select the arithmetic shift for the higher half (HI) of the twos complement input (or logical for unsigned), and (2) the first half processed does not use the [SR OR] option.

```
SI = 0xB6A3;                  /* first input, upper half result */
SE = -3;                      /* shifter exponent */
SR = ASHIFT SI BY -3 (HI);    /* must use HI option */
```

First input (SI): is1011011010100011

SR (shifted by):

```
1111 1111 1111 0110 1101 0100 0110 0000 0000 0000
--sr2----|------sr1---------|-------sr0---------
```

Continuing this example, next, the lower half is processed. Some important points here are: (1) always select a logical shift for the lower half of the input, and (2) the second half processed must use the [SR OR] option to avoid overwriting the previous half of the output value.

```
SI = 0x765D;               /* second input, lower half result} */
                           /* SE = -3 still */
SR = SR OR LSHIFT SI BY -3 (Lo);   /* must use LO option */
```

Second input (SI): is 0111 0110 0101 1101

SR ORed, shifted:

```
1111 1111 1111 0110 1101 0100 0110 1110 1100 1011

---sr2---|--------sr1--------|-------sr0---------
```

The following de-normalization example uses the same input, but processes it in the opposite (lower half first) order. The same important points from before apply: (1) the high half is always arithmetically shifted,

(2) the low half is logically shifted, (3) the first input is passed straight through to SR, and (4) the second half is ORed, creating a double-precision value in SR.

```
SI = 0x765D;                /* first input, lower half result */
SE = -3;                    /* shifter exponent */
SR = LSHIFT SI BY -3 (LO);  /* must use the LO option */
SI = 0xB6A3;                /* second input, upper half result */
SR = SR OR ASHIFT SI BY -3 (Hi);   /* must use HI option */
```

The first input (SI) is: 0111 0110 0101 1101

SR is shifted by:

```
0000 0000 0000 0000 0000 0000 0000 1110 1100 1011
---sr2---|-------sr1---------|--------sr0--------
```

The second input (SI) is: 1011 0110 1010 0011

SR is ORed, shifted by:

```
1111 1111 1111 0110 1101 0100 0110 1110 1100 1011
---sr2---|--------sr1--------|--------sr0--------
```

Normalize, Single-Precision Input

Numbers with redundant sign bits require normalizing. Normalizing a number is the process of shifting a twos complement number within a field so that the rightmost sign bit lines up with the MSB position of the field and recording how many places the number was shifted. The operation can be thought of as a fixed-point to floating-point conversion, generating an exponent and a mantissa.

Normalizing is a two-stage process. The first stage derives the exponent. The second stage does the actual shifting. The first stage uses the EXP instruction, which detects the exponent value and loads it into the SE register. The EXP instruction recognizes a (HI) and (LO) modifier. The second stage uses the Norm instruction. Norm recognizes (HI) and (LO) and also has the [SR OR] option. NORM uses the negated value of the SE register as its shift control code. The negated value is used so that the shift is made in the correct direction.

This is a normalization example for a single-precision input. First, the EXP instruction derives the exponent:

```
AR = 0xF6D4;           /* single-precision input */
SE = EXP AR (HI);      /* Detects exponent with Hi modifier */
```

Input (AR) is: 1111 0110 1101 0100

Exponent (SE) is -3

Next for this single-precision example, the NORM instruction normalizes the input using the derived exponent in SE:

```
SR = NORM AR (HI);
```

Input (AR) is: 1111 0110 1101 0100

SR (normalized):

```
1111 1111 1011 0110 1010 0000 0000 0000 0000 0000
---sr2---|-------sr1---------|-------sr0---------
```

For a single-precision input, the normalize operation can use the (HI) or (LO) modifier, depending on whether the result is needed in SR1 or SR0.

Normalize, ALU Result Overflow

For single-precision data, there is a special normalization situation—normalizing ALU results (AR) that may have overflowed—that requires the HI-extended (HIX) modifier. When using this modifier, the shifter reads the arithmetic status word (ASTAT) overflow bit (AV) and the carry bit (AC) in conjunction with the value in AR. If AV is set (=1), an overflow has occurred. AC contains the true sign of the twos complement value.

Given the following conditions,

`AR= 1111 1010 0011 0010`

`AV=` 1 (indicating overflow)

`AC=` 0 (the true sign bit of this value)

`SE = EXP AR (HIX); SR = NORM AR (HI);`

The normalize operation is as follows:

1. Detect Exponent, Modifier = `HIX`

   `SE` is set to +1.

2. Normalize, Modifier = `HI`, `SE` = 1

   `AR = 1111 1010 0011 0010`

   `SR` (normalized):

   ```
   0000 0000 0111 1101 0001 1001 0000 0000 0000 0000
   ---sr2---|--------sr1--------|--------sr0--------
   ```

The AC bit is supplied as the sign bit, MSB of SR above.

The Norm instruction differs slightly between the ADSP-2191 and previous 16-bit, fixed-point DSPs in the ADSP-2100 family. The difference only can be seen when performing overflow normalization.

- On the ADSP-2191, the NORM instruction checks only that (SE == +1) for performing the shift in of the AC flag (overflow normalization).
- On previous ADSP-2100 family DSPs, the NORM instruction checks both that (SE == +1) and (AV == 1) before shifting in the AC flag.

The Exp (HIX) instruction always sets (SE = +1) when the AV flag is set, so this execution difference only appears when NORM is used without a preceding EXP instruction.

The HIX operation executes properly whether or not there has actually been an overflow as demonstrated by this second example:

AR is: 1110 0011 0101 1011

AV = 0 (indicating no overflow)

AC = 0 (not meaningful when AV = 0)

1. Detect Exponent, Modifier = HIX

   SE is set to: –2

2. Normalize, Modifier = HI, SE = –2

   AR = 1110 0011 0101 1011

SR (normalized):

```
1111 1111 1000 1101 0110 1000 0000 0000 0000 0000

---sr2---|--------sr1--------|--------sr0--------
```

The AC bit is not used as the sign bit. As Figure 2-15 on page 2-59 shows, the HIX mode is identical to the HI mode when AV is not set. When the NORM, LO operation is done, the extension bit is zero; when the NORM, HI operation is done, the extension bit is AC.

Normalize, Double-Precision Input

For double-precision values, the normalization process follows the same general scheme as with single-precision values. The first stage detects the exponent and the second stage normalizes the two halves of the input. For normalizing double-precision values, there are two operations in each stage.

For the first stage, the upper half of the input must be operated on first. This first exponent derivation loads the exponent value into SE. The second exponent derivation, operating on the lower half of the number does not alter the SE register unless SE = –15. This happens only when the first half contained all sign bits. In this case, the second operation loads a value into SE (see Figure 2-16 on page 2-62). This value is used to control both parts of the normalization that follows.

For the second stage, now that SE contains the correct exponent value, the order of operations is immaterial. The first half (whether HI or LO) is normalized without the [SR OR] and the second half is normalized with [SR OR] to create one double-precision value in SR. The (HI) and (LO) modifiers identify which half is being processed.

The following example normalizes double-precision values:

1. Detect Exponent, Modifier = `HI`

   First input: `1111 0110 1101 0100` (upper half)
   `SE` set to: -3

2. Detect Exponent, Modifier = `LO`

   Second input: `0110 1110 1100 1011`
   `SE` unchanged: -3

   Normalize, Modifier = `HI`, [`SR OR`], `SE` = –3

   First input: `1111 0110 1101 0100`

   `SR` (normalized):

   ```
   1111 1111 1011 0110 1010 0000 0000 0000 0000 0000
   ---sr2---|--------sr1--------|--------sr0--------
   ```

3. Normalize, Modifier = `Lo`, [`SR OR`], `SE` = –3

   Second input: `0110 1110 1100 1011`

   `SR` (normalized):

   ```
   1111 1111 1011 0110 1010 0011 0111 0110 0101 1000

   ---sr2---|--------sr1--------|--------sr0--------
   ```

If the upper half of the double-precision input contains all sign bits, the `SE` register value is determined by the second derive exponent operation as shown in this second double-precision normalization example:

1. Detect Exponent, Modifier = `HI`

   First input: `1111 1111 1111 1111` (upper half)
   `SE` set to: -15

2. Detect Exponent, Modifier = `LO`

   Second input: `1111 0110 1101 0100`
   `SE` now set to: -19

3. Normalize, Modifier=`Hi`, No [`SR OR`], `SE` = –19 (negated)

   First input: `1111 1111 1111 1111`

   `SR` (normalized):

   ```
   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
   ---sr2---|--------sr1--------|--------sr0--------
   ```

   Note that all values of `SE` less than –15 (resulting in a shift of +16 or more) upshift the input completely off scale.

4. Normalize, Modifier=`Lo`, [`SR OR`], `SE` = –19 (negated)

   Second input: `1111 0110 1101 0100`

   `SR` (normalized):

   ```
   1111 1111 1011 0110 1010 0000 0000 0000 0000 0000

   ---sr2---|--------sr1--------|--------sr0--------
   ```

Shifter Status Flags

The shifter's logical shift, arithmetic shift, normalize, and derive exponent operations update status flags in the computational unit's Arithmetic Status register (ASTAT). Table A-1 on page A-9 lists all the bits in this register. The following bits in ASTAT flag shifter status (a "1" indicates the condition) for the most recent shifter derive exponent operation:

- **Shifter result overflow.** Bit 7 (SV) indicates overflow (if set, =1) when the MSB of SR1 does not match the eight LSBs of SR2 or indicates no overflow (if clear, =0) The SV is set by multiply/accumulate and shift instructions.
- **Shifter input sign for exponent extract only.** Bit 8 (SS) The SS flag is updated if by derive exponent instructions with the (HI) or (HIX) options set and inputs to the subsequent (LO) instruction.

Flag updates occur at the end of the cycle in which the status is generated and are available on the next cycle.

On previous 16-bit, fixed-point DSPs (ADSP-2100 family), the Shifter Results (SR) register is 32 bits wide and has no overflow detection. On ADSP-219x DSPs, the SR register is 40 bits wide, and overflow in SR is indicated with the SV flag.

Shifter Instruction Summary

Table 2-8 on page 2-55 lists the shifter instructions and indicate how they relate to ASTAT flags. For more information on assembly language syntax, see the *ADSP-219x DSP Instruction Set Reference*. In Table 2-8 on page 2-55, note the meaning of the following symbols:

- **Dreg** indicates any register file location
- * indicates the flag may be set or cleared, depending on results of instruction
- – indicates no effect

Table 2-8. Shifter Instruction Summary

Instruction	ASTAT Status Flags	
	SV	SS
[IF Cond] SR = [SR OR] ASHIFT Dreg [(|HI, LO|)];	*	–
SR = [SR OR] ASHIFT Dreg BY <Imm8> [(|HI, LO|)];	*	–
[IF Cond] SR = [SR OR] LSHIFT Dreg [(|HI, LO|)];	*	–
SR = [SR OR] LSHIFT Dreg BY <Imm8> [(|HI, LO|)];	*	–
[IF Cond] SR = [SR OR] NORM Dreg [(|HI, LO|)];	*	–
[IF Cond] SR = [SR OR] NORM Dreg BY<Imm8> [(|HI, LO|)];	*	–
[IF Cond] SE = EXP Dreg [(|HIX, HI, LO|)];	–	*[1]
[IF Cond] SB = EXPADJ Dreg;	–	–

1 The SS bit is the MSB of input for the HI option. For the HIX option, the SS bit is the MSB of input (for AV = 0) or inverted MSB of input (for AV = 1). There is no effect on SS flag for the LO option.

Shifter Data Flow Details

Figure 2-14 on page 2-56 shows a more detailed diagram of the shifter, which appears in Figure 2-1. The shifter has the following components: the shifter array, the OR/PASS logic, the exponent detector, and the exponent compare logic.

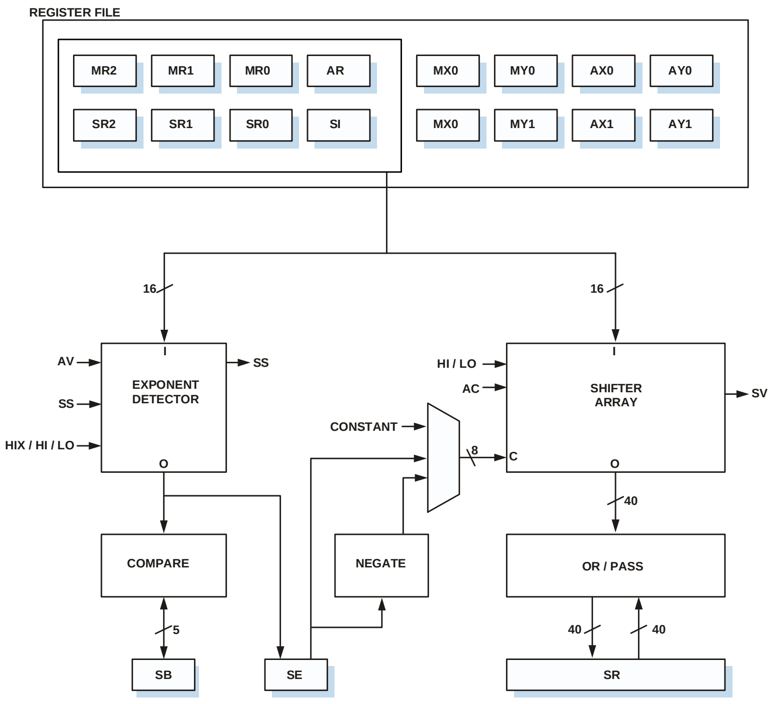


Figure 2-14. Shifter Block Diagram

The shifter array is a 16x40 barrel shifter. It accepts a 16-bit input and can place it anywhere in the 40-bit output field, from off-scale right to off-scale left, in a single cycle. This spread gives 57 possible placements within the 40-bit field. The placement of the 16 input bits is determined by a shift control code (C) and a `HI/LO` option.

Most shifter instructions accept any register of the data register file as an input. This includes immediate shift instructions as well as conditional and multi-function instructions with register to register moves. Restrictions apply to multi-function instructions with parallel data load/store from/to memory. Then, the shifter still accepts `SI`, `AR`, `MR2`, `MR1`, `MR0`, `SR2`, `SR1`, and `SR0`.

For more information on register usage restrictions in conditional and multifunction instructions, see "Multifunction Computations" on page 2-64.

The shifter input provides input to the shifter array and the exponent detector. The shifter result (`SR`) register is 40 bits wide and is divided into three sections: `SR0`, `SR1`, and `SR2`. These individual 16-bit registers are part of the register file. The `SR` register is also fed back to the OR/PASS logic to allow double-precision shift operations.

The `SE` register (shifter exponent) holds the exponent during normalize and denormalize operations. Although it is a 16-bit register for general purposes, shifter operations only use the 8 LSBs. Derive-exponent instructions sign-extend the results to 16 bit. `SE` is not part of the register file but may be accessed through the DM and the PM bus.

The `SB` register (shifter block) is important in block floating-point operations where it holds the block exponent value, which is the value by which the block values must be shifted to normalize the largest value. `SB` holds the most recent block exponent value. Although it is a 16-bit register for general purposes, block exponent operations use the five LSBs only, but sign-extend to 16 bits. `SB` is not part of the register file but can be accessed through DM and PM bus.It is a twos complement, 5.0 value.

Any of the `SI`, `SE`, or `SR` registers can be read and written in the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. All register reads get values loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This allows an input register to provide an operand to the shifter at the beginning of the cycle and be updated with the next operand at the end of the same cycle. It also allows a result register to be stored in memory and updated with a new result in the same cycle.

The shifting of the input is determined by a control code (C) and a HI/LO option. The control code is an 8-bit signed value which indicates the direction and number of places the input is to be shifted. Positive codes indicate a left shift (upshift) and negative codes indicate a right shift (downshift). The control code can come from three sources: the content of the shifter exponent (SE) register, the negated content of the SE register, or an immediate value from the instruction. The ASHIFT and LSHIFT instructions use SE directly, whereas the NORM instructions take the negated SE.

The HI/LO option determines the reference point for the shifting. In the HI state, all shifts are referenced to SR1 (the upper half of the output field), and in the LO state, all shifts are referenced to SR0 (the lower half). The HI/LO feature is useful when shifting 32-bit values because it allows both halves of the number to be shifted with the same control code. HI/LO option is selectable each time the shifter is used.

The shifter fills any bits to the right of the input value in the output field with zeros, and bits to the left are filled with the extension bit. The extension bit can be fed by three possible sources depending on the instruction being performed. The three sources are the MSB of the input, the AC bit from the arithmetic status register (ASTAT), or a zero.

Figure 2-15 on page 2-59 shows the shifter array output as a function of the control code and Hi/Lo signal. In the figure, ABCDEFGHIJKLMNPR represents the 16-bit input pattern, and X stands for the extension bit.

The OR/PASS logic allows the shifted sections of a multiprecision number to be combined into a single quantity. In some shifter instructions, the shifted output may be logically ORed with the contents of the SR register; the shifter array is bitwise ORed with the current contents of the SR regis-

HI Reference Shift Value	LO Reference Shift Value	Shifter Results ---SR2--	-------SR1-------		-------SR0-------	
+24 to +127	**+40 to +127**	**00000000**	**00000000**	**00000000**	**00000000**	**00000000**
+23	+39	R0000000	00000000	00000000	00000000	00000000
+22	+38	PR000000	00000000	00000000	00000000	00000000
+21	+37	NPR00000	00000000	00000000	00000000	00000000
+20	+36	MNPR0000	00000000	00000000	00000000	00000000
+19	+35	LMNPR000	00000000	00000000	00000000	00000000
+18	+34	KLMNPR00	00000000	00000000	00000000	00000000
+17	+33	JKLMNPR0	00000000	00000000	00000000	00000000
+16	**+32**	**IJKLMNPR**	**00000000**	**00000000**	**00000000**	**00000000**
+15	+31	HIJKLMNP	R0000000	00000000	00000000	00000000
+14	+30	GHIJKLMN	PR000000	00000000	00000000	00000000
+13	+29	FGHIJKLM	NPR00000	00000000	00000000	00000000
+12	+28	EFGHIJKL	MNPR0000	00000000	00000000	00000000
+11	+27	DEFGHIJK	LMNPR000	00000000	00000000	00000000
+10	+26	CDEFGHIJ	KLMNPR00	00000000	00000000	00000000
+ 9	+25	BCDEFGHI	JKLMNPR0	00000000	00000000	00000000
+ 8	**+24**	**ABCDEFGH**	**IJKLMNPR**	**00000000**	**00000000**	**00000000**
+ 7	+23	XABCDEFG	HIJKLMNP	R0000000	00000000	00000000
+ 6	+22	XXABCDEF	GHIJKLMN	PR000000	00000000	00000000
+ 5	+21	XXXABCDE	FGHIJKLM	NPR00000	00000000	00000000
+ 4	+20	XXXXABCD	EFGHIJKL	MNPR0000	00000000	00000000
+ 3	+19	XXXXXABC	DEFGHIJK	LMNPR000	00000000	00000000
+ 2	+18	XXXXXXAB	CDEFGHIJ	KLMNPR00	00000000	00000000
+ 1	+17	XXXXXXXA	BCDEFGHI	JKLMNPR0	00000000	00000000
0	**+16**	**XXXXXXXX**	**ABCDEFGH**	**IJKLMNPR**	**00000000**	**00000000**
- 1	+15	XXXXXXXX	XABCDEFG	HIJKLMNP	R0000000	00000000
- 2	+14	XXXXXXXX	XXABCDEF	GHIJKLMN	PR000000	00000000
- 3	+13	XXXXXXXX	XXXABCDE	FGHIJKLM	NPR00000	00000000
- 4	+12	XXXXXXXX	XXXXABCD	EFGHIJKL	MNPR0000	00000000
- 5	+11	XXXXXXXX	XXXXXABC	DEFGHIJK	LMNPR000	00000000
- 6	+10	XXXXXXXX	XXXXXXAB	CDEFGHIJ	KLMNPR00	00000000
- 7	+ 9	XXXXXXXX	XXXXXXXA	BCDEFGHI	JKLMNPR0	00000000
- 8	**+ 8**	**XXXXXXXX**	**XXXXXXXX**	**ABCDEFGH**	**IJKLMNPR**	**00000000**
- 9	+ 7	XXXXXXXX	XXXXXXXX	XABCDEFG	HIJKLMNP	R0000000
-10	+ 6	XXXXXXXX	XXXXXXXX	XXABCDEF	GHIJKLMN	PR000000
-11	+ 5	XXXXXXXX	XXXXXXXX	XXXABCDE	FGHIJKLM	NPR00000
-12	+ 4	XXXXXXXX	XXXXXXXX	XXXXABCD	EFGHIJKL	MNPR0000
-13	+ 3	XXXXXXXX	XXXXXXXX	XXXXXABC	DEFGHIJK	LMNPR000
-14	+ 2	XXXXXXXX	XXXXXXXX	XXXXXXAB	CDEFGHIJ	KLMNPR00
-15	+ 1	XXXXXXXX	XXXXXXXX	XXXXXXXA	BCDEFGHI	JKLMNPR0
-16	**0**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**	**ABCDEFGH**	**IJKLMNPR**
-17	- 1	XXXXXXXX	XXXXXXXX	XXXXXXXX	XABCDEFG	HIJKLMNP
-18	- 2	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXABCDEF	GHIJKLMN
-19	- 3	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXABCDE	FGHIJKLM
-20	- 4	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXABCD	EFGHIJKL
-21	- 5	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXABC	DEFGHIJK
-22	- 6	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXAB	CDEFGHIJ
-23	- 7	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXA	BCDEFGHI
-24	**- 8**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**	**ABCDEFGH**
-25	- 9	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XABCDEFG
-26	-10	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXABCDEF
-27	-11	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXABCDE
-28	-12	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXABCD
-29	-13	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXABC
-30	-14	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXAB
-31	-15	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXA
-32 to -128	**-16 to -128**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**	**XXXXXXXX**

Figure 2-15. Shifter Array Output Placement

ter before being loaded there. When the [SR OR] option is not used in the instruction, the shifter array output is passed through and loaded into the shifter result (SR) register unmodified.

The exponent detector derives an exponent for the shifter input value. The exponent detector operates in one of three ways, which determine how the input value is interpreted. In the HI state, the input is interpreted as a single-precision number or the upper half of a double-precision number. The exponent detector determines the number of leading sign bits and produces a code, which indicates how many places the input must be up-shifted to eliminate all but one of the sign bits. The code is negative so that it can become the effective exponent for the mantissa formed by removing the redundant sign bits.

In the HI-extend state (HIX), the input is interpreted as the result of an add or subtract performed in the ALU which may have overflowed. So, the exponent detector takes the arithmetic overflow (AV) status into consideration. If AV is set, a +1 exponent is output to indicate an extra bit is needed in the normalized mantissa (the ALU carry bit); if AV is not set, HI-extend functions exactly like the Hi state. When performing a derive exponent function in HI or HI-extend modes, the exponent detector also outputs a shifter sign (SS) bit which is loaded into the arithmetic status register (ASTAT). The sign bit is the same as the MSB of the shifter input except when AV is set; when AV is set in HI-extend state, the MSB is inverted to restore the sign bit of the overflowed value.

In the LO state, the input is interpreted as the lower half of a double-precision number. In the Lo state, the exponent detector interprets the SS bit in the arithmetic status register (ASTAT) as the sign bit of the number. The SE register is loaded with the output of the exponent detector only if SE contains –15. This occurs only when the upper half–which must be processed first–contained all sign bits. The exponent detector output is also offset by –16, because the input is actually the lower 16 bits of a 40-bit value. Figure 2-16 on page 2-62 gives the exponent detector characteristics for all three modes.

The exponent compare logic is used to find the largest exponent value in an array of shifter input values. The exponent compare logic in conjunction with the exponent detector derives a block exponent. The comparator compares the exponent value derived by the exponent detector with the value stored in the Shifter Block Exponent (SB) register and updates the SB register only when the derived exponent value is larger than the value in SB register.

Data Register File

The DSP's computational units have a data register file: a set of data registers that transfer data between the data buses and the computation units. DSP programs use these registers for local storage of operands and results.

The register file appears in Figure 2-1 on page 2-3. The register file consists of 16 primary registers and 16 secondary (alternate) registers. All of the data registers are 16 bits wide.

Program memory data accesses and data memory accesses to/from the register file occur on the PM data bus and DM data bus, respectively. One PM data bus access and/or one DM data bus access can occur in one cycle. Transfers between the register files and the DM or PM data buses can move up to 16-bits of valid data on each bus.

If an operation specifies the same register file location as both an input and output, the read occurs in the first half of the cycle and the write in the second half. With this arrangement, the DSP uses the old data as the operand, before updating the location with the new result data. If writes to the same location take place in the same cycle, only the write with

S = Sign bit
N = Non-sign bit
D = Don’t care bit

HI Mode			HIX Mode	
Shifter Array Input	Output	AV	Shifter Array Input	Output
		1	DDDDDDDD DDDDDDDD	+1
SNDDDDDD DDDDDDDD	0	0	SNDDDDDD DDDDDDDD	0
SSNDDDDD DDDDDDDD	-1	0	SSNDDDDD DDDDDDDD	-1
SSSNDDDD DDDDDDDD	-2	0	SSSNDDDD DDDDDDDD	-2
SSSSNDDD DDDDDDDD	-3	0	SSSSNDDD DDDDDDDD	-3
SSSSSNDD DDDDDDDD	-4	0	SSSSSNDD DDDDDDDD	-4
SSSSSSND DDDDDDDD	-5	0	SSSSSSND DDDDDDDD	-5
SSSSSSSN DDDDDDDD	-6	0	SSSSSSSN DDDDDDDD	-6
SSSSSSSS NDDDDDDD	-7	0	SSSSSSSS NDDDDDDD	-7
SSSSSSSS SNDDDDDD	-8	0	SSSSSSSS SNDDDDDD	-8
SSSSSSSS SSNDDDDD	-9	0	SSSSSSSS SSNDDDDD	-9
SSSSSSSS SSSNDDDD	-10	0	SSSSSSSS SSSNDDDD	-10
SSSSSSSS SSSSNDDD	-11	0	SSSSSSSS SSSSNDDD	-11
SSSSSSSS SSSSSNDD	-12	0	SSSSSSSS SSSSSNDD	-12
SSSSSSSS SSSSSSND	-13	0	SSSSSSSS SSSSSSND	-13
SSSSSSSS SSSSSSSN	-14	0	SSSSSSSS SSSSSSSN	-14
SSSSSSSS SSSSSSSS	-15	0	SSSSSSSS SSSSSSSS	-15

	LO Mode	
SS	Shifter Array Input	Output
S	NDDDDDDD DDDDDDDD	-15
S	SNDDDDDD DDDDDDDD	-16
S	SSNDDDDD DDDDDDDD	-17
S	SSSNDDDD DDDDDDDD	-18
S	SSSSNDDD DDDDDDDD	-19
S	SSSSSNDD DDDDDDDD	-20
S	SSSSSSND DDDDDDDD	-21
S	SSSSSSSN DDDDDDDD	-22
S	SSSSSSSS NDDDDDDD	-23
S	SSSSSSSS SNDDDDDD	-24
S	SSSSSSSS SSNDDDDD	-25
S	SSSSSSSS SSSNDDDD	-26
S	SSSSSSSS SSSSNDDD	-27
S	SSSSSSSS SSSSSNDD	-28
S	SSSSSSSS SSSSSSND	-29
S	SSSSSSSS SSSSSSSN	-30
S	SSSSSSSS SSSSSSSS	-31

Figure 2-16. Exponent Detector Characteristics

higher precedence actually occurs. The DSP determines precedence for the write from the type of the operation; from highest to lowest, the precedence is:

1. Move operations: register-to-register, register-to-memory, or memory-to-register

2. Compute operations: ALU, multiplier, or shifter

Secondary (Alternate) Data Registers

Computational units have a secondary register set. To facilitate fast context switching, the DSP includes secondary register sets for data, results, and data address generator registers. Bits in the MSTAT register control when secondary registers become accessible. While inaccessible, the contents of secondary registers are not affected by DSP operations. The secondary register sets for data and results are described in this section.

ⓘ There is a one-cycle latency between writing to MSTAT and being able to access an secondary register set.

ⓘ For more information on secondary data address generator registers, see the "Secondary (Alternate) DAG Registers" on page 4-4.

The MSTAT register controls access to the secondary registers. Table A-2 on page A-10 lists all the bits in MSTAT. The SEC_REG bit in MSTAT controls secondary registers (a "1" enables the secondary set). When set (=1), secondary registers are enabled for the AX0, AX1, AY0, AY1, MX0, MX1, MY0, MY1, SI, SB, SE, AR, MR, and SR registers.

The following example demonstrates how code should handle the one cycle of latency from the instruction, setting the bit in MSTAT to when the secondary registers may be accessed.

```
AR = MSTAT;
AR = SETBIT SEC_REG OF AR;
```

```
MSTAT=AR;              /* activate secondary reg. file */
NOP;                   /* wait for access to secondaries */
AX0 = 7;
```

It is more efficient (no latency) to use the mode enable instruction to select secondary registers. In the following example, note that the swap to secondary registers is immediate:

```
ENA SEC_REG;            /* activate secondary reg. file */
AX0 = 7;                /* now use the secondaries */
```

Multifunction Computations

Using the many parallel data paths within its computational units, the DSP supports multiple-parallel (multifunction) computations. These instructions complete in a single cycle, and they combine parallel operation of the multiplier, ALU, or shifter with data move operations. The multiple operations perform the same as if they were in corresponding single-function computations. Multifunction computations also handle flags in the same way as the single-function computations.

To work with the available data paths, the computation units constrain which data registers may hold the input operands for multifunction computations. These constraints limit which registers may hold the X-input and Y-input for the ALU, multiplier, and shifter. For details, refer to the ALU, multiplier and shifter sections of this manual and to the *ADSP-219x DSP Instruction Set Reference*.

For unconditional, single-function instructions, any of the registers within the register file may serve as X- or Y-inputs (see Figure 2-1 on page 2-3). The following code example shows the differences between conditional versus unconditional instructions and single-function versus multifunction instructions.

```
/* Conditional computation instructions begin with an IF clause.
The DSP tests whether the condition is true before executing the
instruction. */

AR = AX0 + AY0;        /* unconditional: add X and Y ops*/
If EQ AR = AX0 + AY0;  /* conditional: if AR=0, add X and Y ops*/

/* Multifunction instructions are sets of instruction that exe-
cute in a single cycle. The instructions are delimited with
commas, and the combined multifunction instruction is terminated
with a semicolon. */

AR = AX0-AY0;             /* single function ALU subtract */
AX0 = MR1;                /* single function */
                          /* register-to-register move */
AR = AX0-AY0, AX0 = MR1; /* multifunction, both in 1 cycle */
```

The upper part of the Shifter Result (SR) register may not serve as feedback to ALU and multiplier. For information on the SR2, SB, SE, MSTAT, and ASTAT registers see the discussion on page 2-4.

Only the ALU and multiplier X- and Y-operand registers (MX0, MX1, MY0, MY1, AX0, AY1) have memory data bus access in dual-memory read multifunction instructions.

Table 2-9 lists the multifunction instructions. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols:

- **ALU, MAC, SHIFT** indicate any ALU, multiplier, or shifter instruction
- **Dreg** indicates any register file location
- **XOP, YOP** indicate any X- and Y-input registers, indicating a register usage restriction for conditional and/or multifunction instructions.

Table 2-9. ADSP-219x Multifunction Instruction Summary

Instruction[1]
|<ALU>, <MAC>|, Xop = DM(Ia += Mb), Yop = PM(Ic += Md);
Xop = DM(Ia += Mb), Yop = PM(Ic += Md);
|<ALU>, <MAC>,<SHIFT> |, Dreg = |DM(Ia += Mb), PM(Ic += Md)|;
|<ALU>, <MAC>, <SHIFT>|, |DM(Ia += Mb), PM(Ic += Md)| = Dreg;
|<ALU>, <MAC>, <SHIFT>|, Dreg = Dreg;

1 Multifunction instructions are sets of instruction that execute in a single cycle. The instructions are delimited with commas, and the combined multifunction instruction is terminated with a semicolon.

3 PROGRAM SEQUENCER

This chapter provides the following topics:

Overview

The DSP's program sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the DSP. Program flow in the DSP is mostly linear with the processor executing program instructions sequentially. This linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in Figure 3-1 on page 3-2. Non-sequential structures direct the DSP to execute an instruction that is not at the next sequential address. These structures include:

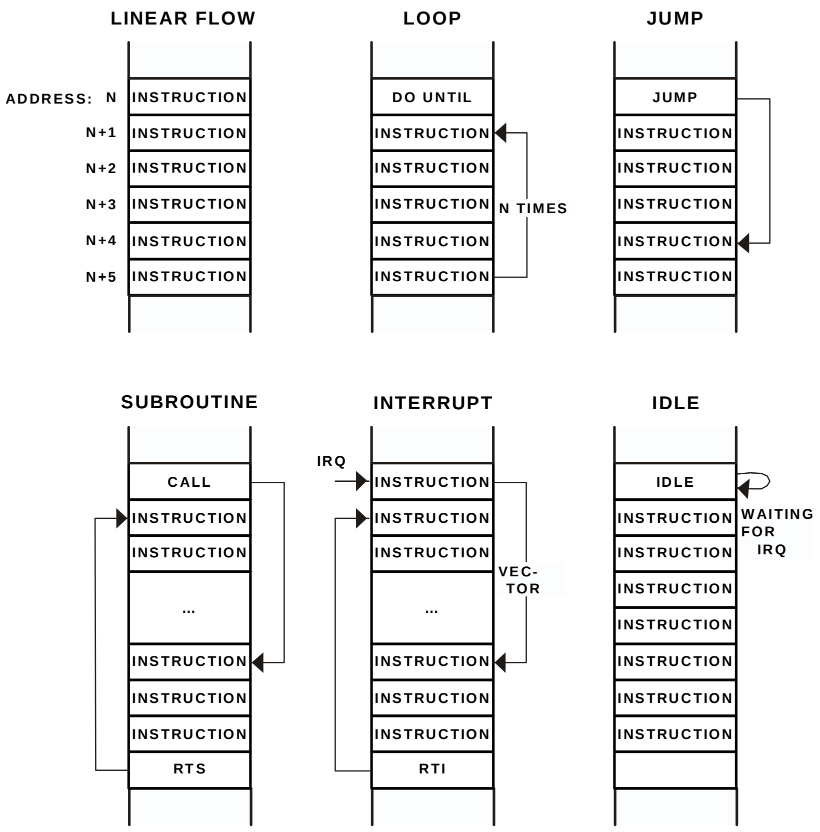


Figure 3-1. Program Flow Variations

Loops. One sequence of instructions executes several times with near-zero overhead.

- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow transfers permanently to another part of program memory.

- **Interrupts.** Subroutines in which a runtime event triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute. As part of this process, the sequencer handles the following tasks:

- Increments the fetch address
- Maintains stacks
- Evaluates conditions
- Decrements the loop counter
- Calculates new addresses
- Maintains an instruction cache
- Handles interrupts

To accomplish these tasks, the sequencer uses the blocks shown in Figure 3-2 on page 3-5. The sequencer's address multiplexer selects the value of the next fetch address from several possible sources. The fetched address enters the instruction pipeline, ending with the program counter (PC). The pipeline contains the 24-bit addresses of the instructions currently being fetched, decoded, and executed. The `PC` couples with the `PC` stack, which stores return addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses.

ⓘ Figure 3-2 on page 3-5 uses the following abbreviations: ADDR=address, BRAN=branch, IND=indirect, DIR=direct, RT=return, RB=rollback, INCR=increment, PC-REL=PC relative, PC=program counter.

ⓘ The diagram in Figure 3-2 on page 3-5 also describes the relationship between the program sequencer in the ADSP-219x DSP core and inputs to that sequencer that differ for various members of the ADSP-219x family DSPs.

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

With selective caching, the instruction cache lets the DSP access data in memory and fetch an instruction (from the cache) in the same cycle. The program sequencer uses the cache if there is a data access which uses the PM bus (called a PM data access) or if a data access over the DM bus uses the same block of memory as the current instruction fetch (a block conflict).

In addition to providing data addresses, the Data Address Generators (DAGs) provide instruction addresses for the sequencer's indirect branches.

The sequencer evaluates conditional instructions and loop termination conditions using information from the status registers. The loop stacks support nested loops. The status stack stores status registers for implementing interrupt service routines.

Table 3-1 on page 3-6 and Table 3-2 on page 3-7 list the registers within and related to the program sequencer. All registers in the program sequencer are register group 1 (`Reg1`), register group 2 (`Reg2`), or register

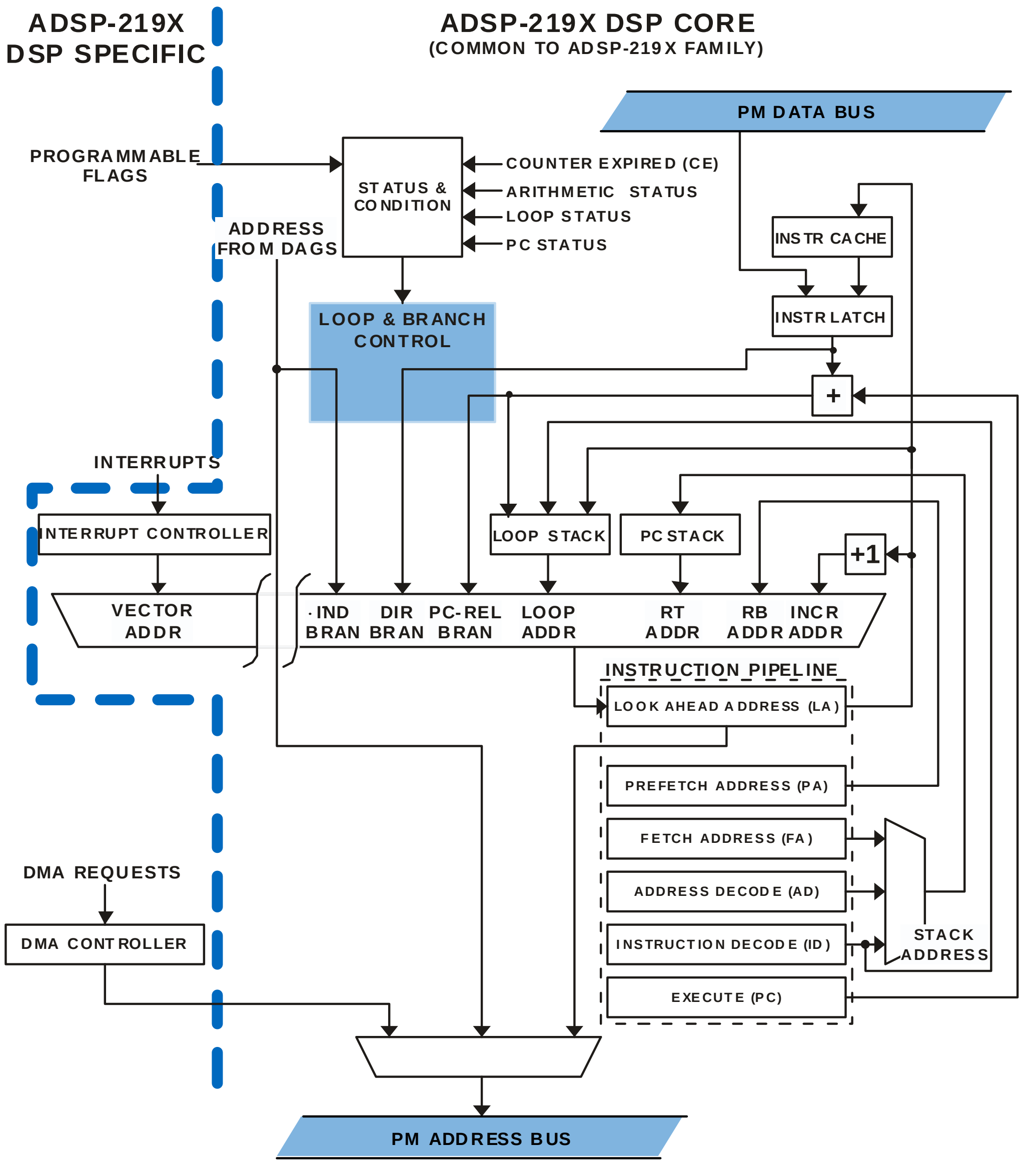


Figure 3-2. Program Sequencer Block Diagram

group 3 (`Reg3`) registers, so they are accessible to other data (`Dreg`) registers and to memory. All the sequencer's registers are directly readable and writable, except for the `PC`. Manual pushing or popping the `PC` stack is done using explicit instructions and the PC Stack Page (`STACKP`) register and PC Stack Address (`STACKA`) register, which are readable and writable. Pushing or popping the loop stacks and status stack also requires explicit instructions. For information on using these stacks, see "Stacks and Sequencing" on page 3-36.

A set of system control registers configures or provides input to the sequencer. These registers include `ASTAT`, `MSTAT`, `CCODE`, `IMASK`, `IRPTL`, and `ICNTL`. Writes to some of these registers do not take effect on the next cycle. For example, after a write to the `MSTAT` register to enable ALU saturation mode, the change does not take effect until one cycles after the write. With the lists of sequencer and system registers, Table 3-1 on page 3-6 and Table 3-2 on page 3-7 summarize the number of extra cycles (latency) for a write to take effect (effect latency) and for a new value to appear in the register (read latency). A "0" indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed, and a "1" indicates one extra cycle.

Table 3-1. Program Sequencer Register Effect Latencies

Register	Contents	Bits	Effect Latency
CNTR	loop count loaded on next Do/Until loop	16	1[1]
IJPG	Jump Page (upper eight bits of address)	8	1
IOPG	I/O Page (upper eight bits of address)	8	1
DMPG1	DAG1 Page (upper eight bits of address)	8	1
DMPG2	DAG2 Page (upper eight bits of address)	8	1

1 CNTR has a one-cycle latency before an If Not CE instruction, but has zero latency otherwise.

Table 3-2. System Register Effect Latencies

Register	Contents	Bits	Effect Latency
ASTAT	Arithmetic status	9	1
MSTAT	Mode status	7	0[1]
SSTAT	System status	8	n/a
CCODE	Condition Code	16	1
IRPTL	Interrupt latch	16	1
IMASK	Interrupt mask	16	1
ICNTL	Interrupt control	16	1
CACTL	Cache control	3	5[2]

1 Changing MSTAT bits with the Ena or Dis mode instruction has a 0 effect latency; when writing to MSTAT or performing a Pop Sts, the effect latencies vary based on the altered bits.

2 Except for the CFZ bit, which has an effect latency of four cycles.

Instruction Pipeline

The instruction pipeline takes account for memory read latencies. Once an address emits to on-chip memory it takes two cycles until the data is available to the core. That is why the LA stage generates the address for the instruction fetched in the FA stage, and the AD stage outputs the address(es) for the data read in the PC cycle.

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 DSP executes instructions from program memory in sequential order by incrementing the look-ahead address. Using its instruction pipeline, the DSP processes instructions in six clock cycles:

- **Look-Ahead Address (LA).** The DSP determines the source for the instruction from inputs to the look-ahead address multiplexer.
- **Prefetch Address (PA)** and **Fetch Address (FA).** The DSP reads the instruction from either the on-chip instruction cache or from program memory.
- **Address Decode (AD)** and **Instruction Decode (ID).** The DSP decodes the instruction, generating conditions that control instruction execution.
- **Execute (PC).** The DSP executes the instruction; the operations specified by the instruction complete in a single cycle.

These cycles overlap in the pipeline, as shown in Table 3-3 on page 3-9. In sequential program flow, when one instruction is being fetched, the instruction fetched three cycles previously is being executed. With few exceptions, sequential program flow has a throughput of one instruction per cycle. The exceptions are the two-cycle instructions: 16- or 24-bit immediate data write to memory with indirect addressing, long jump (`LJUMP`), and long call (`LCALL`).

Any non-sequential program flow can potentially decrease the DSP's instruction throughput. Non-sequential program operations include:

- Data accesses that conflict with instruction fetches
- Jumps
- Subroutine calls and returns
- Interrupts and return
- Loops (of less than five instructions)

Table 3-3. Pipelined Execution Cycles

Cycles	LA	PA	FA	AD	ID	PC
1	0x08 ↘					
2	0x09 ↘	0x08 ↘				
3	0x0A ↘	0x09 ↘	0x08 ↘			
4	0x0B ↘	0x0A ↘	0x09 ↘	0x08 ↘		
5	0x0C ↘	0x0B ↘	0x0A ↘	0x09 ↘	0x08 ↘	
6	0x0D ↘	0x0C ↘	0x0B ↘	0x0A ↘	0x09 ↘	0x08
7	0x0E ↘	0x0D ↘	0x0C ↘	0x0B ↘	0x0A ↘	0x09
8	0x0F	0x0E	0x0D	0x0C	0x0B	0x0A

**Look Ahead Address (LA). Prefetch Address (PA). Fetch Address (FA).
Address Decode (AD). Instruction Decode (ID). Execute (PC).**

Instruction Cache

Usually, the sequencer fetches an instruction from memory on each cycle. Occasionally, bus constraints prevent some of the data and instructions from being fetched in a single cycle. To alleviate these data flow constraints, the DSP has an instruction cache, which appears in Figure 3-3 on page 3-11.

When the DSP executes an instruction that requires data access over the PM data bus, there is a *bus conflict* because the sequencer uses the PM data bus for fetching instructions.

When a data transfer over the DM bus accesses the same memory block from which the DSP is fetching an instruction, there is a *block conflict* because only one bus can access a block at a time.

To avoid bus and block conflicts, the DSP caches these instructions, reducing delays. Except for enabling or disabling the cache, its operation requires no intervention. For more information, see "Using the Cache" on page 3-12 and "Optimizing Cache Usage" on page 3-12.

The first time the DSP encounters a fetch conflict, the DSP must wait to fetch the instruction on the following cycle, causing a delay. The DSP automatically writes the fetched instruction to the cache to prevent the same delay from happening again.

The sequencer checks the instruction cache on every PM data access or block conflict. If the needed instruction is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

Because of the six-stage instruction pipeline, as the DSP executes an instruction (at address n) that requires a PM data access or block conflict, this execution creates a conflict with the instruction fetch (at address n+3), assuming sequential execution. The cache stores the fetched instruction (n+3), not the instruction requiring the data access.

If the instruction needed to avoid a conflict is in the cache, the cache provides the instruction while the data access is performed. If the needed instruction is not in the cache, the instruction fetch from memory takes place in the cycle following the data access, incurring one cycle of overhead. If the cache is enabled and not frozen, the fetched instruction is loaded into the cache, so that it is available the next time the same conflict occurs.

Figure 3-3 on page 3-11 shows a block diagram of the instruction cache. The cache holds 64 instruction-address pairs. These pairs (or cache entries) are arranged into 32 (31-0) cache sets according to the instruction address' five least significant bits (4-0). The two entries in each set (entry

0 and entry 1) have a valid bit, indicating whether the entry contains a valid instruction. The least recently used (LRU) bit for each set indicates which entry was not used last (0=entry 0, and 1=entry 1).

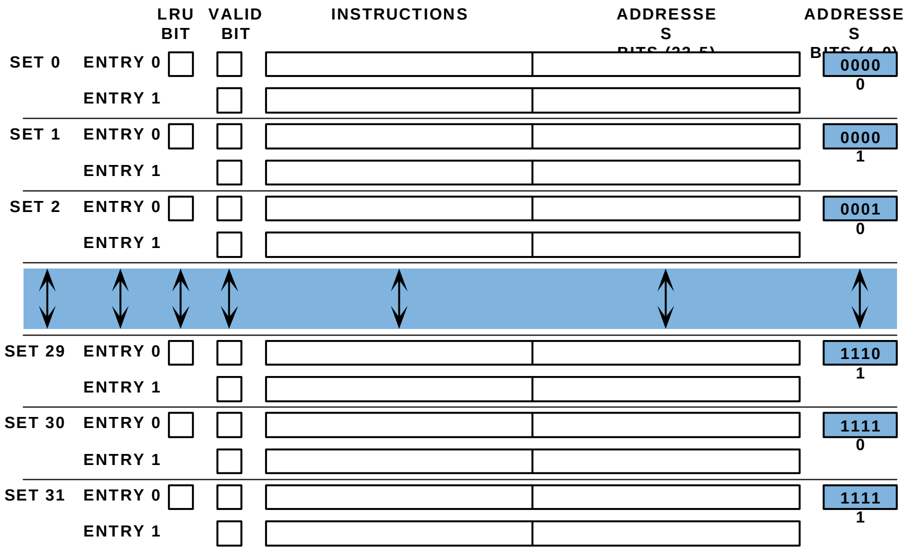


Figure 3-3. Instruction Cache Architecture

The cache places instructions in entries according to the five LSBs of the instruction's address. When the sequencer checks for an instruction to fetch from the cache, it uses the five address LSBs as an index to a cache set. Within that set, the sequencer checks the addresses and valid bits of the two entries, looking for the needed instruction. If the cache contains the instruction, the sequencer uses the entry and updates the LRU bit to indicate the entry did not contain the needed instruction.

When the cache does not contain a needed instruction, the cache loads a new instruction and its address, placing these in the least recently used entry of the appropriate cache set and toggling the LRU bit.

Using the Cache

After a DSP reset, the cache is cleared (containing no instructions), unfrozen, and enabled. From then on, the `CACTL` register controls the operating mode of the instruction cache. As a system control register, `CACTL` can be accessed by `reg(CACTL)=dreg;` and `dreg=reg(CACTL);` instructions. Table A-6 on page A-20 lists all the bits in `CACTL`. The following bits in `CACTL` control cache modes:

Cache DM bus access Enable. Bit 5 (`CDE`) directs the sequencer to cache conflicting DM bus accesses (if 1) or not to cache conflicting DM bus accesses (if 0).

Cache Freeze. Bit 6 (`CFZ`) directs the sequencer to freeze the contents of the cache (if 1) or let new entries displace the entries in the cache (if 0).

Cache PM bus access Enable. Bit 7 (`CPE`) directs the sequencer to cache conflicting PM bus accesses (if 1) or not to cache conflicting PM bus accesses (if 0).

After reset `CDE` and `CPE` are set and `CFZ` is cleared.

When program memory changes, programs need to resynchronize the instruction cache with program memory using the `FLUSH CACHE` instruction. This instruction flushes the instruction cache, invalidating all instructions currently cached, so the next instruction fetch results in a memory access.

Optimizing Cache Usage

Usually, cache operation is efficient and requires no intervention, but certain ordering of instructions can work against the cache's architecture and can degrade cache efficiency. When the order of PM bus data accesses or block conflicts and instruction fetches continuously displaces cache entries and loads new entries, the cache is not being efficient. Rearranging the order of these instructions remedies this inefficiency.

An example of code that works against cache efficiency appears in Table 3-4 on page 3-14. The program memory data access at address `0x0100` in the loop, `Outer`, causes the cache to load the instruction at `0x0103` (into set 19). Each time the program calls the subroutine, `Inner`, the program memory data accesses at `0x0300` and `0x500` displace the instruction at `0x0103` by loading the instructions at `0x0303` and `0x0503` (also into set 19). If the program only calls the `Inner` subroutine rarely during the `Outer` loop execution, the repeated cache loads do not greatly influence performance. If the program frequently calls the subroutine while in the loop, the cache inefficiency has a noticeable effect on performance. To improve cache efficiency on this code (if for instance, execution of the `Outer` loop is time-critical), it would be good to rearrange the order of some instructions. Moving the subroutine call up one location (starting at `0x02FE`) would work here, because with that order the two cached instructions end up in cache set 18 instead of set 19.

Because the least significant five address bits determine which cache set stores an instruction, instructions in the same cache set are multiples of 64 address locations apart. As demonstrated in the optimization example, it is a rare combination of instruction sequences that can lead to “cache thrashing”—iterative swapping of cache entries.

Branches and Sequencing

One of the types of non-sequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL`/return instruction begins execution at a new location, other than the next sequential address. For descriptions on how to use the `JUMP` and `CALL`/return instructions, see the *ADSP-219x DSP Instruction Set Reference*. Briefly, these instructions operate as follows:

Table 3-4. Cache-Inefficient Code

Address	Instruction
0x00FE	`CNTR=1024;`
0x00FF	`Do Outer Until CE;`
0x0100	`AX0=DM(I0+=M0), PM(I4+=M4)=AY0;`
…	
0x0103	`If EQ Call Inner;`
0x0104	`AR=AX1 + AY1;`
0x0105	`MR=MX0*MY0 (SS);`
0x0106	`Outer: SR=MX1*MY1(SS);`
0x0107	`PM(I7+=M7)=SR1;`
…	
0x02FF	`Inner: SR0=AY0;`
0x0300	`AY0=PM(I5+=M5);`
…	
0x0500	`PM(I5+=M5)=AY1;`
…	
0x05FF	`Rts;`

A `JUMP` or a `CALL`/return instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically pushes the return address (the next sequential address after the `CALL` instruction) onto the `PC` stack. 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 at the top of the `PC` stack. The two types of return instructions are return from subroutine (`RTS`) and return from interrupt (RTI). While the return from subroutine (RTS) only pops the return address off the PC stack, the return from interrupt (RTI) pops the return address and pops the status stack.

There are a number of parameters that programs can specify for branches:

JUMP and CALL/return instructions can be conditional. The program sequencer can evaluate status conditions to decide whether to execute a branch. If no condition is specified, the branch is always taken. For more information on these conditions, see "Conditional Sequencing" on page 3-41.

- JUMP and CALL/return instructions can be immediate or delayed. Because of the instructions pipeline, an immediate branch incurs four lost (overhead) cycles. A delayed branch incurs two cycles of overhead. For more information, see "Delayed Branches" on page 3-17.
- JUMP and CALL/return instructions can be used within DO/UNTIL counter (CE) or infinite (FOREVER) loops, but a JUMP or CALL instruction may not be the last instruction in the loop. For information, see "Restrictions on Ending Loops" on page 3-24.

The sequencer block diagram in Figure 3-2 on page 3-5 shows that branches can be direct or indirect. The difference is that the sequencer generates the address for a direct branch, and the PM data address generator (DAG2) produces the address for an indirect branch.

Direct branches are JUMP or CALL/return instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative 16-bit address. To branch farther, the LJUMP or LCALL instructions use a 24-bit address. Some instruction examples that cause a direct branch are:

```
JUMP fft1024;      /* fft1024 is an address label */
CALL 10;           /* 10 is a PC-relative address */
```

Indirect branches are JUMP or CALL/return instructions that use a dynamic—changes at runtime—address that comes from the PM data address generator. For more information on the data address generator, see "DAG Operations" on page 4-9. Some instruction examples that cause an indirect branch are:

```
JUMP (I6);          /* (i6) is a DAG2 register */
CALL (I7);          /* (i7) is a DAG2 register */}
```

Indirect Jump Page (IJPG) Register

The IJPG register provides the upper eight address bits for indirect Jump and Call instructions. When performing an indirect branch, the sequencer gets the lower 16 bits of the branch address from the I register specified in the JUMP or CALL instruction and uses the IJPG register to complete the address.

At power up, the DSP initializes the IJPG register to 0x0. Initializing the Indirect Jump Page (IJPG) register only is necessary when the instruction is located on a page other than the current page.

Changing the contents of the sequencer page register is not automatic and requires explicit programming.

Conditional Branches

The sequencer supports conditional branches. These are JUMP or Call/return instructions whose execution is based on testing an IF condition. For more information on condition types in If condition instructions, see "Conditional Sequencing" on page 3-41.

Delayed Branches

The instruction pipeline influences how the sequencer handles branches. For immediate branches—JUMP and CALL/return instructions not specified as delayed branches (DB), four instruction cycles are lost (NOPs) as the pipeline empties and refills with instructions from the new branch.

As shown in Table 3-5 on page 3-17 and Table 3-6 on page 3-18, the DSP does not execute the four instructions after the branch, which are in the fetch and decode stages. For a CALL, the next instruction (the instruction after the CALL) is the return address. During the four lost (no-operation) cycles, the pipeline fetches and decodes the first instruction at the branch address.

Table 3-5. Pipelined Execution Cycles for Immediate Branch (JUMP/CALL)

Cycles	LA	PA	FA	AD	ID	PC
1	j	n+4→nop[1]	n+3→nop[1]	n+2→nop[1]	n+1→nop[1]	n
2	j+1	j	n+4→nop[1]	n+3→nop[1]	n+2→nop[1]	Nop[2]
3	j+2	j+1	j	n+4→nop[1]	n+3→nop[1]	Nop
4	j+3	j+2	j+1	j	n+4→nop[1]	Nop
5	j+4	j+3	j+2	j+1	j	Nop
6	j+5	j+4	j+3	j+2	j+1	j
Note that n is the branching instruction, and j is the instruction branch address.						

1 n+1, n+2, n+3, and n+4 are suppressed.
2 For call, return address (n+1) is pushed on the PC stack.

For delayed branches—JUMP and CALL/return instructions with the delayed branches (DB) modifier, only two instruction cycles are lost in the pipeline, because the DSP executes the two instructions after the branch while the pipeline fills with instructions from the new branch.

Table 3-6. Pipelined Execution Cycles for Immediate Branch (Return)

Cycles	LA	PA	FA	AD	ID	PC
1	r	n+4→nop[1]	n+3→nop[1]	n+2→nnop[1]	n+1→nop[1]	n
2	r+1	r	n+4→nop[1]	n+3→nop[1]	n+2→nop[1]	Nop[2]
3	r+2	r+1	r	n+4→nop[1]	n+3→nop[1]	Nop
4	r+3	r+2	r+1	r	n+4→nop[1]	Nop
5	r+4	r+3	r+2	r+1	r	Nop
6	r+5	r+4	r+3	r+2	r+1	r
Note that n is the branching instruction, and r is the instruction branch address.						

1 n+1, n+2, n+3, and n+4 are suppressed.
2 r (n+1 in Table 3-5 on page 3-17) the return address is popped from PC stack.

As shown in Table 3-7 on page 3-19 and Table 3-8 on page 3-19, the DSP executes the two instructions after the branch, while the instruction at the branch address is fetched and decoded. In the case of a `CALL`, the return address is the third instruction after the branch instruction. While delayed branches use the instruction pipeline more efficiently than immediate branches, it is important to note that delayed branch code can be harder to understand because of the instructions between the branch instruction and the actual branch.

Besides being somewhat more challenging to code, there are also some limitations on delayed branches that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instruc-

Table 3-7. Pipelined Execution Cycles for Delayed Branch (Jump/Call)

Cycles	LA	PA	FA	AD	ID	PC
1	j	n+4→nop1[1]	n+3→nop[1]	n+2	n+1	n
2	j+1	j	n+4→nop[1]	n+3→nop[1]	n+2	n+1[2]
3	j+2	j+1	j	n+4→nop[1]	n+3→nop[1]	n+2[2]
4	j+3	j+2	j+1	j	n+4→nop[1]	Nop[3]
5	j+4	j+3	j+2	j+1	j	Nop
6	j+5	j+4	j+3	j+2	j+1	j
Note that n is the branching instruction, and j is the instruction branch address.						

1 n+3 and n+4 are suppressed.
2 Delayed branch slots.
3 For call, return address (n+3) is pushed on the PC stack.

Table 3-8. Pipelined Execution Cycles for Delayed Branch (Return)

Cycles	LA	PA	FA	AD	ID	PC
1	r[1]	n+4→nop[2]	n+3→nop[2]	n+2	n+1	n
2	r+1	r	n+4→nop[2]	n+3→nop[2]	n+2	n+1[3]
3	r+2	r+1	r	n+4→nop[2]	n+3→nop[2]	n+2[3]
4	r+3	r+2	r+1	r	n+4→nop[2]	Nop
5	r+4	r+3	r+2	r+1	r	Nop
6	r+5	r+4	r+3	r+2	r+1	r
Note that n is the branching instruction, and r is the instruction branch address.						

1 r (n+1 in Table 3-7 on page 3-19) the return address is popped from PC.
2 stackn+3 and n+4 are suppressed.
3 Delayed branch slots.

tions that follow it must execute sequentially, the instructions in the two locations (delayed branch slots) that follow a delayed branch instruction may not be any of the following:

- Other branches (no `JUMP`, `CALL`, or `RTI/RTS` instructions)
- Any stack manipulations (no `PUSH` or `POP` instructions or writes to the `PC` stack)
- Any loops or other breaks in sequential operation (no `DO`/`UNTIL` or `IDLE` instructions)
- Two-cycle instructions may not appear in the second delay branch slot; these instructions may appear in the first delay branch slot.

Development software for the DSP flags these types of instructions in the two locations after a delayed branch instruction as code errors.

Interrupt processing is also influenced by delayed branches and the instruction pipeline. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the DSP does not immediately process an interrupt that occurs in between a delayed branch instruction and either of the two instructions that follow. Any interrupt that occurs during these instructions is latched, but not processed until the branch is complete.

Loops and Sequencing

Another type of non-sequential program flow that the sequencer supports is looping. A loop occurs when a `DO`/`UNTIL` instruction causes the DSP to repeat a sequence of instructions infinitely (`FOREVER`) or until the counter expires (`CE`).

The condition for terminating a loop with the DO/UNTIL logic is loop Counter Expired (CE). This condition tests whether the loop has completed the number of iterations loaded from the CNTR register. Loops that exit with conditions other than CE (using a conditional JUMP) have some additional restrictions. For more information, see "Restrictions on Ending Loops" on page 3-24. For more information on condition types in DO/UNTIL instructions, see "Conditional Sequencing" on page 3-41.

The DO/UNTIL instruction uses the sequencer's loop and condition features, which appear in Figure 3-2 on page 3-5. These features provide efficient software loops, without the overhead of additional instructions to branch, test a condition, or decrement a counter. The following code example shows a DO/UNTIL loop that contains three instructions and iterates 30 times.

```
CNTR=30; DO the_end UNTIL CE;     /* loop iterates 30 times */
         AX0 = DM(I0+=M0), AY0 = PM(I4+=M4);
         AR = AX0-AY0;
the_end: DM(I1+=M0) = AR;         /* last instruction in loop */
```

When executing a DO/UNTIL instruction, the program sequencer pushes the address of the loop's last instruction and loop's termination condition onto the loop-end stack. The sequencer also pushes the loop-begin address—address of the instruction following the DO/UNTIL instruction—onto the loop-begin stack.

The sequencer's instruction pipeline architecture influences loop termination. Because instructions are pipelined, the sequencer must test the termination condition and decrement the counter at the end of the loop. Based on the test's outcome, the next fetch either exits the loop or returns to the beginning of the loop.

The `DO/UNTIL` instruction supports infinite loops, using the `FOREVER` condition instead of `CE`. Software can use a conditional `JUMP` instruction to exit such an infinite loop.

When using a conditional `JUMP` to exit any `DO/UNTIL` loop, software must perform some loop stack maintenance (`POP LOOP`). For more information, see "Stacks and Sequencing" on page 3-36.

The condition test occurs when the DSP is executing the last instruction in the loop (at location e, where e is the end-of-loop address). If the condition tests false, the sequencer repeats the loop, fetching the instruction from the loop-begin address, which is stored on the loop-begin stack. If the condition tests true, the sequencer terminates the loop, fetching the next instruction after the end of the loop and popping the loop stacks. For more information, see "Stacks and Sequencing" on page 3-36.

Table 3-9 on page 3-22 and Table 3-10 on page 3-23 show the pipeline states for loop iteration and termination.

Table 3-9. Pipelined Execution Cycles for Loop Back (Iteration)

Cycles	LA	PA	FA	AD	ID	PC
1	e[1]	e–1	e–2	e–3	e–4	e–5
2	b[2]	e	e–1	e–2	e–3	e–4
3	b+1	b	e	e–1	e–2	e–3
4	b+2	b+1	b	e	e–1	e–2
Note that e is the loop end instruction, and b is the loop begin instruction.						

Table 3-9. Pipelined Execution Cycles for Loop Back (Iteration)

Cycles	LA	PA	FA	AD	ID	PC
5	b+3	b+2	b+1	b	e	e–1
6	b+4[3]	b+3[3]	b+2[3]	b+1[3]	b[3]	e[3]
7	b+5	b+4	b+3	b+2	b+1	b
Note that e is the loop end instruction, and b is the loop begin instruction.						

1 Termination condition tests false.

2 Loop start address is top of loop-begin stack.

3 For loops of less than six instructions (shorter than the pipeline), the pipeline retains the instructions in the loop (e through b+4). On the first iteration of such a short loop, there is a branch penalty of four Nops while the pipeline sets up for the short loop.

Table 3-10. Pipelined Execution Cycles for Loop Termination

Cycles	LA	PA	FA	AD	ID	PC
1	e[1]	e–1	e–2	e–3	e–4	e–5
2	e+1	e	e–1	e–2	e–3	e–4
3	e+2	e+1	e	e–1	e–2	e–3
4	e+3	e+2	e+1	e	e–1	e–2
5	e+4	e+3	e+2	e+1	e	e–1
6	e+5	e+4	e+3	e+2	e+1	e
7	e+6	e+5	e+4	e+3	e+2	e+1[2]
Note that e is the loop end instruction.						

1 Termination condition tests true.

2 Loop aborts and loop stacks pop.

Managing Loop Stacks

To support low overhead looping, the DSP stores information for loop processing in three stacks: loop-begin stack, loop-end stack, and counter stack. The sequencer manages these stacks for loops that terminate when the counter expires (`DO`/`UNTIL CE`), but does not manage these stacks for loops that terminate with a conditional `JUMP`. For information on managing loop stacks, see "Stacks and Sequencing" on page 3-36.

Restrictions on Ending Loops

The sequencer's loop features (which optimize performance in many ways) limit the types of instructions that may appear at or near the end of the loop. The only absolute restriction is that the last instruction in a loop (at the loop end label) may not be a `CALL`/return, a `JUMP (DB)`, or a two-cycle instruction.

There are restrictions on placing nested loops. For example, nested loops may not use the same end-of-loop instruction address.

Use care if using `PUSH LOOP` or `POP LOOP` instruction inside loops. Perform any stack maintenance outside of loops.

Interrupts and Sequencing

This section provides the following topics:

Overview

Another type of non-sequential program flow that the sequencer supports is interrupt processing. Interrupts may stem from a variety of conditions, both internal and external to the processor. In response to an interrupt, the sequencer processes a subroutine call to a predefined address, the interrupt vector. The DSP assigns a unique vector to each interrupt.

The ADSP-219x DSP core supports 16 prioritized interrupts. The four highest-priority interrupts (reset, powerdown, stack, and kernel) are part of the DSP core and are common to all ADSP-219x DSPs. The rest of the interrupt levels are assignable to peripherals off the DSP core and vary with the particular DSP. For information on working with peripheral interrupts, see "System Interrupt Controller" on page 6-1 and "Configuring and Servicing Interrupts" on page 14-27.

The DSP supports a number of prioritized, individually-maskable off-core interrupts, some of which can be either level- or edge-sensitive. External interrupts occur when an off-chip device asserts one of the DSP's interrupt inputs. The DSP also supports internal interrupts. An internal interrupt can stem from stack overflows or a program writing to the interrupt's bit in the IRPTL register. Several factors control the DSP's response to an interrupt.

The DSP responds to an interrupt request when:

- The DSP is executing instructions or is in an IDLE state
- The interrupt is not masked
- Interrupts are globally enabled
- A higher-priority request is not pending

When the DSP responds to an interrupt, the sequencer branches program execution with a call to the corresponding interrupt vector address. Within the DSP's program memory, the interrupt vectors are grouped in an area called the interrupt vector table. The interrupt vectors in this table are spaced at intervals that varies with the particular DSP; this spacing permits placing most interrupt service routines at the vector location—instead of branching to the actual interrupt service routine from the vector location. For a list of interrupt vector addresses and their associated latch and mask bits, see "System Interrupt Controller" on page 6-1. Each interrupt vector has associated latch and mask bits. Table A-4 on page A-16 lists the latch and mask bits.

To process an interrupt, the DSP's program sequencer:

1. Outputs the appropriate interrupt vector address
2. Pushes the next PC value (the return address) onto the PC stack
3. Pushes the current value of the ASTAT and MSTAT registers onto the status stack
4. Clears the appropriate bit in the interrupt latch register (IRPTL)

At the end of the interrupt service routine, the sequencer processes the return from interrupt (RTI) instruction and:

1. Returns to the address stored at the top of the PC stack
2. Pops this value off of the PC stack
3. Pops the status stack

All interrupt service routines should end with a return-from-interrupt (RTI) instruction. Although the interrupt vector table holds space for a reset service routine, it is important to note that DSP reset/startup routines do not operate the same as other interrupt service routines. After reset, the PC stack is empty, so there is no return address. The last instruction of the reset service routine should be a JUMP to the start of the program.

If software writes to a bit in IRPTL forcing an interrupt, the processor recognizes the interrupt in the following cycle, and four cycles of branching to the interrupt vector follow the recognition cycle.

The DSP responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (four cycles). Table 3-11 on page 3-28, Table 3-12 on page 3-29, and Table 3-13 on page 3-29 show the pipelined execution cycles for interrupt processing.

For all interrupts, on-core and off-core, only one instruction is executed after the interrupt is recognized for service (and before the two instructions are aborted) while the processor fetches and decodes the first instruction of the service routine. For more information on interrupt latency, see "System Interrupt Controller" on page 6-1.

If nesting is enabled and a higher-priority interrupt occurs immediately after a lower-priority interrupt, the service routine of the higher-priority interrupt is delayed by at least three additional cycles. For more information, see "Nesting Interrupts" on page 3-34.

Table 3-11. Pipelined Execution Cycles for Interrupt During Single-Cycle Instruction

Cycles	LA	PA	FA	AD	ID	PC
1	n+4	n+3	n+2	n+1	n	n–1[1]
2	v	n+4→nop[2]	n+3→nop[2]	n+2→nop[2]	n+1→nop[2]	n[3]
3	v+1	v	n+4→nop[2]	n+3→nop[2]	n+2→nop[2]	Nop[2]
4	v+2	v+1	v	n+4→nop[2]	n+3→nop[2]	Nop
5	v+3	v+2	v+1	v	n+4→nop[2]	Nop
6	v+4	v+3	v+2	v+1	v	Nop
7	v+5	v+4	v+3	v+2	v+1	v[4]
Note that n is the single-cycle instruction, and v is the interrupt vector instruction.						

1 Interrupt occurs.
2 n+1 pushed on PC stack; ASTAT/MSTAT pushed onto status stack; n+1 suppressed.
3 Interrupt recognized.
4 Interrupt vector output.

Certain DSP operations that span more than one cycle hold off interrupt processing. If an interrupt occurs during one of these operations, the DSP latches the interrupt, but delays its processing. The operations that delay interrupt processing are as follows:

- A branch (`JUMP` or `CALL`/return) instruction and the following cycle, whether it is an instruction (in a delayed branch) or a `NOP` (in a non-delayed branch)
- The first of the two cycles used to perform a PM bus data access and an instruction fetch
- The set up cycles for loops shorter than the instruction pipeline (fewer than five instructions).

Table 3-12. Pipelined Execution Cycles for Interrupt During Instruction with Conflicting PM Data Access (Instruction not Cached)

Cycles	LA	PA	FA	AD	ID	PC
1	n+4	n+3	n+2	n+1	n	n–1[1]
2	—	n+4	n+3	n+2	n+1	n[2]
3	v[3]	n+5→nop[4]	n+4→nop[4]	n+3→nop[4]	n+2→nop[4]	Nop[4]
4	v+1	v	n+5→nop[4]	n+4→nop[4]	n+3→nop[4]	Nop[4]
5	v+2	v+1	v	n+5→nop[4]	n+4→nop[4]	Nop
6	v+3	v+2	v+1	v	n+5→nop[4]	Nop
7	v+4	v+3	v+2	v+1	v	Nop
8	v+5	v+4	v+3	v+2	v+1	v[5]
Note that n is the single-cycle instruction, and v is the interrupt vector instruction.						

1 Interrupt occurs.
2 Interrupt recognized, but not processed; PM data access.
3 Interrupt processed.
4 n+1 pushed on PC stack; ASTAT/MSTAT pushed onto status stack; n+1 suppressed.
5 Interrupt vector output.

Table 3-13. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction

Cycles	LA	PA	FA	AD	ID	PC
1	n+4	n+3	n+2	n+1	n	n–1[1]
2	j	n+4→nop	n+3→nop	n+2	n+1	n[2]
3	j+1	j	n+4→nop	n+3→nop	n+2	n+1
4	v[3]	j+1→nop[4]	j→nop[4]	n+4→nop[4]	n+3→nop[4]	n+2
Note that n is the delayed branch instruction, j is the instruction at the branch address, and v is the interrupt vector instruction.						

Table 3-13. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction (Cont'd)

Cycles	LA	PA	FA	AD	ID	PC
5	v+1	v	j+1→nop[4]	j→nop[4]	n+4→nop[4]	Nop3
6	v+2	v+1	v	j+1→nop[4]	j→nop[4]	Nop[4]
7	v+3	v+2	v+1	v	j+1→nop[4]	Nop[4]
8	v+4	v+3	v+2	v+1	v	Nop[5]
9	v+5	v+4	v+3	v+2	v+1	v[6]
Note that n is the delayed branch instruction, j is the instruction at the branch address, and v is the interrupt vector instruction.						

1 Interrupt occurs.
2 Interrupt recognized, but not processed.
3 Interrupt processed.
4 ASTAT/MSTAT pushed onto status stack; n+3 suppressed.
5 j pushed on PC stack; j+1 suppressed.
6 Interrupt vector output.

- Any waitstates for external memory accesses
- Any external memory access that is required when the DSP does not have control of the external bus or during a host bus grant

Sensing Interrupts

The DSP supports two types of interrupt sensitivity—the signal shape that triggers the interrupt. On interrupt pins, either the input signal's edge or level can trigger an external interrupt. For more information on interrupt sensitivity and timing, see "System Interrupt Controller" on page 6-1.

Masking Interrupts

The sequencer supports interrupt masking—latching an interrupt, but not responding to it. Except for the emulator ($\overline{\texttt{EMU}}$), reset ($\overline{\texttt{RESET}}$), and power-down interrupts, all interrupts are maskable. If a masked interrupt is latched, the DSP responds to the latched interrupt if it is later unmasked.

Interrupts can be masked globally or selectively. Bits in the `ICNTL` and `IMASK` registers control interrupt masking. Figure A-5 on page A-16 lists the bits in `ICNTL`, and Table A-4 on page A-16 lists the bits in `IMASK`. These bits control interrupt masking as follows:

- **Global interrupt enable.** `ICNTL`, Bit 5 (`GIE`) directs the DSP to enable (if 1) or disable (if 0) all interrupts
- **Interrupt mask.** `IMASK`, Bits 15-0 direct the DSP to enable (if 1) or disable/mask (if 0) the corresponding interrupt

At runtime, it is recommended to control the `GIE` bit through the "`ENA INT;`" and "`DIS INT;`" instruction pair. All maskable interrupts are disabled at reset. For booting, the DSP automatically unmasks associated interrupts and uses the selected peripheral as the source for boot data.

Latching Interrupts

When the DSP recognizes an interrupt, the DSP's Interrupt Latch (`IRPTL`) register latches the interrupt—sets a bit to record that the interrupt occurred. The bits in this register indicate all interrupts that are currently being serviced or are pending. Because the `IRPTL` register is readable and writable, interrupts can be set or cleared in software. For example, "`IRPTL = 0;`" clears all interrupt requests.

Programs can use the `SETINT` and `CLRINT` instructions to set or clear individual interrupts in `IRPTL` without the risk of affecting other incoming interrupts.

When returning from an interrupt, the sequencer clears the corresponding bit in IRPTL. During execution of the interrupt's service routine, the DSP core cannot latch the same interrupt again while the service routine is executing.

The interrupt latch bits in IRPTL correspond to interrupt mask bits in the IMASK register. In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) to 15 (lowest). Interrupt priority determines which interrupt is serviced first when more than one occurs in the same cycle. Priority also determines which interrupts are nested when the DSP has interrupt nesting enabled. For more information, see "Nesting Interrupts" on page 3-34.

Depending on the assignment of interrupts to peripherals, one event can cause multiple interrupts, and multiple events can trigger the same interrupt. For more information, see "System Interrupt Controller" on page 6-1.

Interrupt Vector Table

If an interrupt is latched and granted, program execution jumps to the interrupt's vector address. Table 3-14 shows the interrupt vectors associated with the 16 core interrupt channels.

Table 3-14. Interrupt Priorities/Addresses

Interrupt	IMASK/ IRPTL	Vector Address[1]
Emulator (NMI)— Highest Priority	NA	NA
Reset (NMI)	0	0x00 0000
Power-Down (NMI)	1	0x00 0020
Emulation Kernel	2	0x00 0040
Loop and PC Stack	3	0x00 0060

Table 3-14. Interrupt Priorities/Addresses (Cont'd)

Interrupt	**IMASK/ IRPTL**	**Vector Address**[1]
User Assigned Interrupt	4	0x00 0080
User Assigned Interrupt	5	0x00 00A0
User Assigned Interrupt	6	0x00 00C0
User Assigned Interrupt	7	0x00 00E0
User Assigned Interrupt	8	0x00 0100
User Assigned Interrupt	9	0x00 0120
User Assigned Interrupt	10	0x00 0140
User Assigned Interrupt	11	0x00 0160
User Assigned Interrupt	12	0x00 0180
User Assigned Interrupt	13	0x00 01A0
User Assigned Interrupt	14	0x00 01C0
User Assigned Interrupt—Lowest Priority	15	0x00 01E0

1 These interrupt vectors start at address 0x10000 when the DSP is in "no-boot", run-form-external memory mode.

Every interrupt vector features 32 memory locations. If the interrupt service routine takes less than 33 instructions, an additional jump can be saved. The interrupt vector tables resides at on-chip address `0x000000` unless the `RMODE` bit of the System Configuration (`SYSCR`) register is set. This can also be altered at runtime.

Stacking Status During Interrupts

To run in an interrupt-driven system, programs depend on the DSP being restored to its pre-interrupt state after an interrupt is serviced. The sequencer's status stack eases the return from interrupt process by eliminating some interrupt service overhead—register saves and restores. For a description of stack operations, see "Stacks and Sequencing" on page 3-36.

Nesting Interrupts

The sequencer supports interrupt nesting—responding to another interrupt while a previous interrupt is being serviced. Bits in the ICNTL, IMASK, and IRPTL registers control interrupt nesting. Table A-5 on page A-16 lists the bits in ICNTL, Table A-5 on page A-16 lists the bits in IMASK, and IRPTL. These bits control interrupt nesting as follows:

- **Interrupt nesting enable.**ICNTL, Bit 4 (INE), directs the DSP to enable (if 1) or disable (if 0) interrupt nesting.
- **Interrupt Mask.**IMASK, 16 Bits, selectively masks the interrupts. For each bit's corresponding interrupt, these bits direct the DSP to unmask (enable, if 1) or mask (disable, if 0) the matching interrupt.
- **Interrupt Latch.** IRPTL, 16 Bits, latch interrupts. For each bit's corresponding interrupt, these bits indicate that the DSP has latched (pending, if 1) or not latched (not pending, if 0) the matching interrupt.

When interrupt nesting is disabled, a higher-priority interrupt cannot interrupt a lower-priority interrupt's service routine. Other interrupts are latched as they occur, but the DSP processes them after the active routine finishes.

When interrupt nesting is enabled, a higher-priority interrupt can interrupt a lower-priority interrupt's service routine. Lower interrupts are latched as they occur, but the DSP processes them after the nested routines finish.

Programs should only change the interrupt nesting enable (`INE`) bit while outside of an interrupt service routine.

If nesting is enabled and a higher-priority interrupt occurs immediately after a lower-priority interrupt, the service routine of the higher-priority interrupt is delayed by up to several cycles. This delay allows the first instruction of the lower-priority interrupt routine to be executed, before it is interrupted.

If an interrupt re-occurs while its service routine is running and nesting is enabled, the DSP does not latch the re-occurrence in `IRPTL`. The DSP waits until the return from interrupt (`RTI`) completes, before permitting the interrupt to latch again.

Interrupt Latency

To service an interrupt, the DSP requires one core cycle for synchronization, another cycle for recognition, and four cycles to branch to the interrupt vector. Additionally servicing can be delayed by the following conditions:

- Higher or even priority interrupt is running
- Lower priority interrupt is running, but nesting is disabled
- Interrupted code executes a delayed branch
- Interrupted code enters a hardware loop
- Interrupted code causes a cache miss
- External memory waitstates
- Host bus grant situations

Placing the DSP in Idle Mode

The sequencer supports placing the DSP in IDLE—a special instruction that halts the processor core—until an interrupt occurs. When executing an Idle instruction, the sequencer fetches one more instruction at the current fetch address and then suspends operation. The DSP's I/O processor is not affected by the IDLE instruction—DMA transfers to or from internal memory continue uninterrupted—depending on the IDLE mode.

The processor's on-chip peripherals continue to run during IDLE. When an interrupt occurs, the processor responds normally. After two cycles used to fetch and decode the first instruction of the interrupt service routine, the processor resumes execution with the service routine.

For information on using IDLE with power-down modes, see "Using Clock Modes" on page 14-37.

Stacks and Sequencing

The sequencer includes five stacks: PC stack, loop-begin stack, loop-end stack, counter stack, and status stack. These stacks preserve information about program flow during execution branches. Figure 3-4 on page 3-37 shows how these stacks relate to each other and to the registers that load (PUSH) or are loaded from (POP) these stacks. Besides showing the operations that occur during explicit push and pop instructions, Figure 3-4 on page 3-37 also indicates which stacks the DSP automatically pushes and pops when processing different types of branches: loops (DO/UNTIL), calls (CALL/return), and interrupts.

These stacks have differing depths. The PC stack is 33 locations deep; the status stack is 16 locations deep; and the loop begin, loop end, and counter stacks are eight locations deep. A stack is full when all entries are

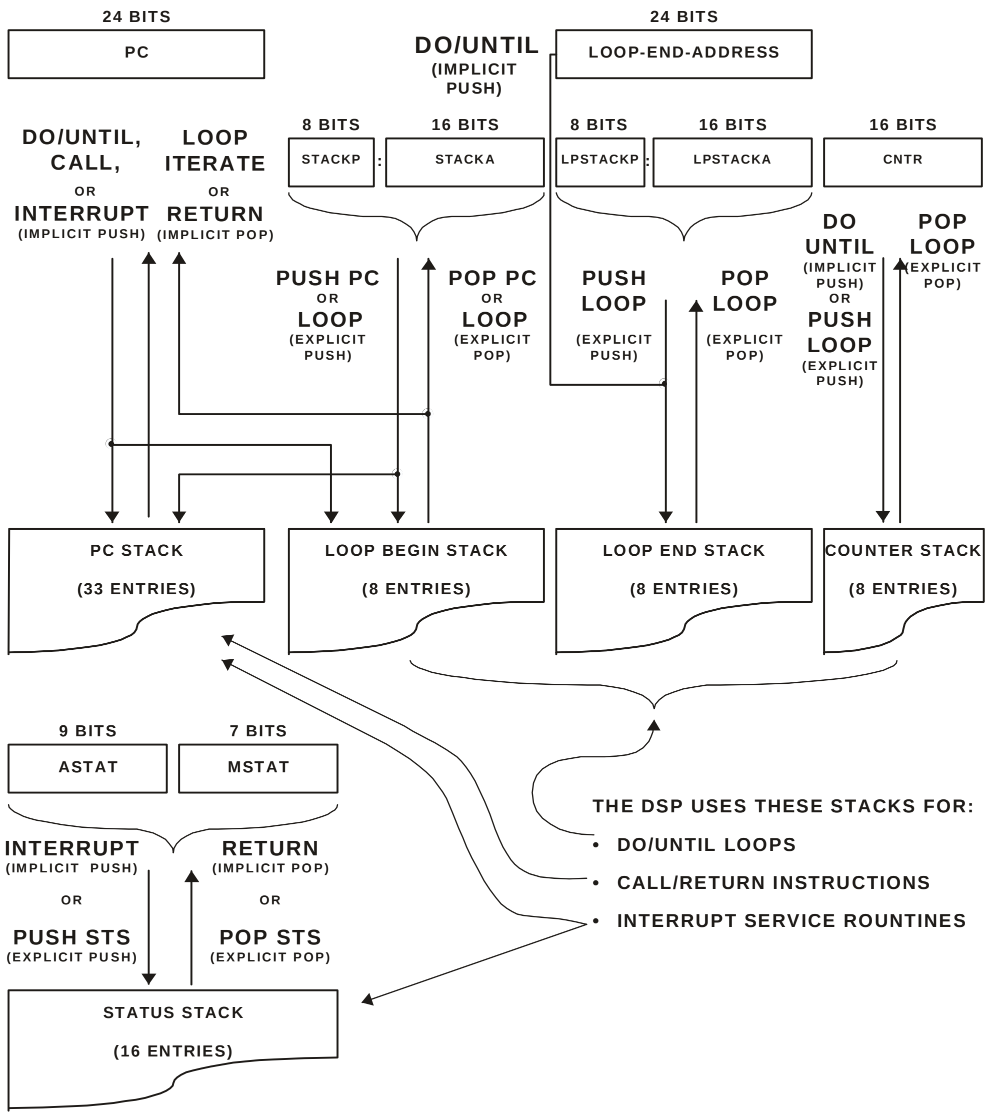

24 BITS
PC
DO/UNTIL (IMPLICIT PUSH)
24 BITS
LOOP-END-ADDRESS
DO/UNTIL, CALL, OR INTERRUPT (IMPLICIT PUSH)
LOOP ITERATE OR RETURN (IMPLICIT POP)
8 BITS
16 BITS
STACKP
STACKA
8 BITS
16 BITS
LPSTACKP
LPSTACKA
16 BITS
CNTR
PUSH PC OR LOOP (EXPLICIT PUSH)
POP PC OR LOOP (EXPLICIT POP)
PUSH LOOP (EXPLICIT PUSH)
POP LOOP (EXPLICIT POP)
DO UNTIL (IMPLICIT PUSH) OR PUSH LOOP (EXPLICIT PUSH)
POP LOOP (EXPLICIT POP)
PC STACK (33 ENTRIES)
LOOP BEGIN STACK (8 ENTRIES)
LOOP END STACK (8 ENTRIES)
COUNTER STACK (8 ENTRIES)
9 BITS
7 BITS
ASTAT
MSTAT
INTERRUPT (IMPLICIT PUSH) OR PUSH STS (EXPLICIT PUSH)
RETURN (IMPLICIT POP) OR POP STS (EXPLICIT POP)
STATUS STACK (16 ENTRIES)
THE DSP USES THESE STACKS FOR:
• DO/UNTIL LOOPS
• CALL/RETURN INSTRUCTIONS
• INTERRUPT SERVICE ROUNTINES

Figure 3-4. Program Sequencer Stacks

occupied. Bits in the SSTAT register indicate the stack status. Figure A-3 on page A-11 lists the bits in the SSTAT register. The SSTAT bits that indicate stack status are:

- **PC stack empty.** Bit 0 (PCSTKEMPTY) indicates that the PC stack contains at least one pushed address (if 0) or PC stack is empty (if 1).
- **PC stack full.** Bit 1 (PCSTKFULL) indicates that the PC stack contains at least one empty location (if 0) or PC stack is full (if 1).
- **PC stack level.** Bit 2 (PCSTKLVL) indicates that the PC stack contains between 3 and 28 pushed addresses (if 0) or PC stack is at or above the high-watermark—28 pushed addresses, or it is at or below the low-watermark—3 pushed addresses (if 1).
- **Loop stack empty.** Bit 4 (LPSTKEMPTY) indicates that the loop stack contains at least one pushed address (if 0) or the loop stack is empty (if 1).
- **Loop stack full.** Bit 5 (LPSTKFULL) indicates that the loop stack contains at least one empty location (if 0) or the loop stack is full (if 1).
- **Status stack empty.** Bit 6 (STSSTKEMPTY) indicates that the status stack contains at least one pushed status (if 0) or status stack is empty (if 1).
- **Stacks overflowed.** Bit 7 (STKOVERFLOW) indicates that an overflow/underflow has not occurred (if 0) or indicates that at least one of the stacks (PC, loop, counter, status) has overflowed, or the PC or status stack has underflowed (if 1). Note that STKOVERFLOW is only cleared on reset. Loop stack underflow is not detected because it occurs only as a result of a POP LOOP operation.

Stack status conditions can cause a STACK interrupt. The stack interrupt always is generated by a stack overflow condition, but also can be generated by ORing together the stack overflow status (STKOVERFLOW) bit and stack high/low level status (PCSTKLVL) bit. The level bit is set when:

- The PC stack is pushed and the resulting level is at or above the high watermark.
- The PC stack is popped and the resulting level is at or below the low watermark.

This spill-fill mode (using the stack's status to generate a stack interrupt) is disabled on reset. Bits in the ICNTL register control whether the DSP generates this interrupt based on stack status. Table A-5 on page A-16 lists the bits in the ICNTL register. The bits in ICNTL that enable the STACK interrupt are:

- **Global interrupt enable.** Bit 5 (GIE) globally disables (if 0) or enables (if 1) unmasked interrupts
- **PC stack interrupt enable.** Bit 10 (PCSTKE) directs the DSP to disable (if 0) or enable (if 1) spill-fill mode—ORing of stack status—to generate the STACK interrupt.

When switching on spill-fill mode, a spurious (low) stack level interrupt may occur (depending on the level of the stack). In this case, the interrupt handler should push some values on the stack to raise the level above the low-level threshold.

Values move onto (PUSH) or off (POP) the stacks through implicit and explicit operations. Implicit stack operations are stack accesses that the DSP performs while executing a branch instruction (CALL/return, DO/UNTIL) or while responding to an interrupt. Explicit stack operations are stack accesses that the DSP performs while executing the stack instructions (PUSH, POP).

As shown in Figure 3-4 on page 3-37, the source for the pushed values and destination for the pop value differs depending on whether the stack operations is implicit or explicit.

In implicit stack operations, the DSP places values on the stacks from registers (PC, CNTR, ASTAT, MSTAT) and from calculated addresses (end-of-loop, PC+1). For example, a CALL/return instruction directs the DSP to branch execution to the called subroutine and push the return address (PC+1) onto the PC stack. The matching return from subroutine instruction (RTS) causes the DSP to pop the return address off of the PC stack and branch execution to the address following the CALL.

A second instruction that makes the DSP perform implicit stack operations is the Do/Until instruction. It takes the following steps to set up a DO/UNTIL loop:

- Load the loop count into the CNTR register
- Initiate the loop with a DO/UNTIL instruction
- Terminate the loop with an end-of-loop label

When executing a DO/UNTIL instruction, the DSP performs the following implicit stack operations:

- Pushes the loop count from the CNTR register onto the counter stack
- Pushes the start-of-loop address from the PC onto the loop start stack
- Pushes the end-of-loop address from the end-of-loop label onto the loop-end stack

When the count in the top location of the counter stack expires, the loop terminates, and the DSP pops the three loop stacks, resuming execution at the address after the end of the loop. The count is decremented on the stack, *not* in the CNTR register.

A third condition/instruction that makes the DSP perform implicit stack operations is an interrupt/return instruction. When interrupted, the DSP pushes the PC onto the PC stack, pushes the ASTAT and MSTAT registers onto the status stack, and branches execution to the interrupt service routine's location (vector). At the end of the routine, the return from interrupt instruction directs the DSP to pop these stacks and branch execution to the instruction after the interrupt (PC+1).

In explicit stack operations, a program's access to the stacks goes through a set of registers: STACKP, STACKA, LPSTACKP, LPSTACKA, CNTR, ASTAT, and MSTAT. A POP instruction retrieves the value or address from the corresponding stack (PC, Loop, or Sts) and places that value in the corresponding register (as shown in Figure 3-4 on page 3-37). A PUSH instruction takes the value or address from the register and puts it on the corresponding stack. Programs should use explicit stack operations for stack maintenance, such as managing the stacks when exiting a DO/UNTIL loop with a conditional JUMP.

Conditional Sequencing

The sequencer supports conditional execution with conditional logic that appears in Figure 3-4 on page 3-37. This logic evaluates conditions for conditional (IF) instructions and loop (DO/UNTIL) terminations. The conditions are based on information from the arithmetic status (ASTAT) register, the condition code (CCODE) register, the flag inputs, and the loop counter. For more information on arithmetic status, see "Using Computational Status" on page 2-18.

Each condition that the DSP evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in Table 3-15 on page 3-42. For most conditions, the sequencer can test both true and false states. For example, the sequencer can evaluate ALU equal-to-zero (EQ) and ALU not-equal-to-zero (NE).

To test conditions that do not appear in Table 3-15 on page 3-42, a program can use the test bit (`TSTBIT`) instruction to test bit values loaded from status registers. For more information, see the *ADSP-219x DSP Instruction Set Reference*.

Table 3-15. IF Condition and DO/UNTIL Termination Logic

Syntax	Status Condition	True If:	Do/Until	If cond
EQ	Equal Zero	AZ = 1	✗	✓
NE	Not Equal Zero	AZ = 0	✗	✓
LT	Less Than Zero	AN .XOR. AV = 1	✗	✓
GE	Greater Than or Equal Zero	AN .XOR. AV = 0	✗	✓
LE	Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1	✗	✓
GT	Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0	✗	✓
AC	ALU Carry	AC = 1	✗	✓
Not AC	Not ALU Carry	AC = 0	✗	✓
AV	ALU Overflow	AV = 1	✗	✓
Not AV	Not ALU Overflow	AV = 0	✗	✓
MV	MAC Overflow	MV = 1	✗	✓
Not MV	Not MAC Overflow	MV = 0	✗	✓
SWCOND	Compares value in CCODE register with following DSP conditions: PF0-13 inputs Hi, AS, SV	CCODE=SWCOND	✗	✓

Table 3-15. IF Condition and DO/UNTIL Termination Logic (Cont'd)

Syntax	Status Condition	True If:	Do/Until	If cond
Not SWCOND	Compares value in CCODE register with following DSP conditions: PF0-13 inputs Lo, Not AS, Not SV	CCODE= Not SWCOND	✗	✓
CE	Counter Expired	loop counter = 0	✓	✗
Not CE	Counter Not Expired	loop counter = Not 0	✗	✓
Forever	Always (Do)		✓	✗
True	Always (If)		✗	✓

The two conditions that do not have complements are `CE`/`NOT CE` (loop counter expired/not expired) and `TRUE`/`FOREVER`. The context of these condition codes determines their interpretation. Programs should use `TRUE` and `NOT CE` in conditional (`IF`) instructions. Programs should use `FOREVER` and `CE` to specify loop (`DO`/`UNTIL`) termination. A `DO FOREVER` instruction executes a loop indefinitely, until an interrupt, jump, or reset intervenes.

There are some restrictions on how programs may use conditions in `DO`/`UNTIL` loops. For more information, see "Restrictions on Ending Loops" on page 3-24.

Sequencer Instruction Summary

Table 3-16 on page 3-45 lists the program sequencer instructions and how they relate to SSTAT flags. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In Table 3-16 on page 3-45, note the meaning of the following symbols:

- **Reladdr#** indicates a PC-relative address of #number of bits
- **Addr24** indicates an absolute 24-bit address
- **Ireg** indicates an Index (I) register in either DAG
- **Imm4** indicates an immediate 4-bit value
- **Addr24** indicates an absolute 24-bit address
- * indicates the flag may be set or cleared, depending on results of instruction
- – indicates no effect

Table 3-16. Sequencer Instruction Summary

Instruction	SSTAT Status Flags						
	LE	LF	PE	PF	PL	SE	SO
Do <Reladdr12> Until [CE, Forever];	*	*	–	–	–	–	*
[If Cond] Jump <Reladdr13> [(DB)];	–	–	–	–	–	–	–
Call <Reladdr16> [(DB)];	–	–	*	*	*	–	*
Jump <Reladdr16> [(DB)];	–	–	–	–	–	–	–
[If Cond] Lcall <Addr24>;	–	–	*	*	*	–	*
[If Cond] Ljump <Addr24>;	–	–	–	–	–	–	–
[If Cond] Call <Ireg> [(DB)];	–	–	*	*	*	–	*
[If Cond] Jump <Ireg> [(DB)];	–	–	–	–	–	–	–
[If Cond] Rti [(DB)];	–	–	*	*	*	*	–
[If Cond] Rts [(DB)];	–	–	*	*	*	–	–
Push |PC, Loop, Sts|;	*	*	*	*	*	*	*
Pop |PC, Loop, Sts|;	*	*	*	*	*	*	*
Flush Cache;	–	–	–	–	–	–	–
Setint <Imm4>;	–	–	*	*	*	*	*
Clrint <Imm4>;	–	–	–	–	–	–	–
Nop;	–	–	–	–	–	–	–
Idle;	–	–	–	–	–	–	–
Ena | MM, AS, OL, BR, SR, BSR, INT | ;	–	–	–	–	–	–	–
Dis | MM, AS, OL, BR, SR, BSR, INT | ;	–	–	–	–	–	–	–

Abbreviations for SSTAT Flags:
LE = LPSTKEMPTY
LF = LPSTKFULL
PE = PCSTKEMPTY
PF = PCSTKFULL
PL = PCSTKLVL
SE = STSSTKEMPTY
SO = STKOVERFLOW

4 DATA ADDRESS GENERATORS (DAGS)

This chapter provides the following sections:

Overview

The DSP's Data Address Generators (DAGs) generate addresses for data moves to and from Data Memory (DM) and Program Memory (PM). By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The DAG architecture, which appears in Figure 4-1 on page 4-3, supports several functions that minimize overhead in data access routines. These functions include:

- **Supply address and post-modify**—provides an address during a data move and auto-increments the stored address for the next move.
- **Supply pre-modified address**—provides a modified address during a data move without incrementing the stored address.

- **Modify address**—increments the stored address without performing a data move.

- **Bit-reverse address**—provides a bit-reversed address during a data move without reversing the stored address.

The ADSP-2191 has a unified memory, so Program Memory and Data Memory distinction differ from previous ADSP-218x DSPs. For information on the unified memory, see "Overview" on page 5-1.

As shown in Figure 4-1 on page 4-3, each DAG has five types of registers. These registers hold the values that the DAG uses for generating addresses. The types of registers are:

- **Index registers (I0-I3 for DAG1 and I4-I7 for DAG2).** An Index register holds an address and acts as a pointer to memory. For example, the DAG interprets `DM(I0)` and `PM(I4)` syntax in an instruction as addresses.

- **Modify registers (M0-M3 for DAG1 and M4-M7 for DAG2).** A Modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For example, the `dm(I0+=M1)` instruction directs the DAG to output the address in register `I0` then modify the contents of `I0` using the `M1` register.

- **Length and Base registers (L0-L3 and B0-B3 for DAG1 and L4-L7 and B4-B7 for DAG2).** Length registers and Base registers set up the range of addresses and the starting address for a circular buffer. For more information on circular buffers, see "Addressing Circular Buffers" on page 4-12.

- **DAG Memory Page registers (DMPG1 for DAG1 and DMPG2 for DAG2).** Page registers set the upper eight bits address for DAG memory accesses; the 16-bit Index registers and Base registers hold the lower 16 bits. For more information on about DAG page registers and addresses from the DAGs, see "Data Memory Page Registers (DMPGx)" on page 4-7.

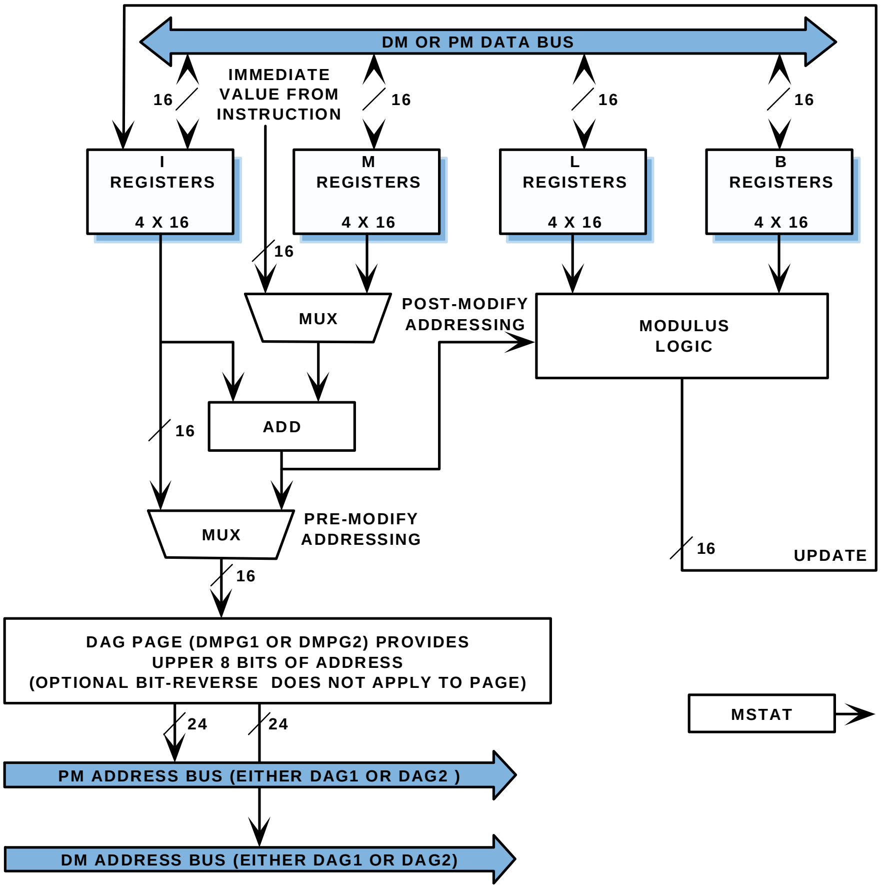

DM OR PM DATA BUS
IMMEDIATE VALUE FROM INSTRUCTION
16
I REGISTERS 4 X 16
M REGISTERS 4 X 16
L REGISTERS 4 X 16
B REGISTERS 4 X 16
MUX
POST-MODIFY ADDRESSING
MODULUS LOGIC
ADD
PRE-MODIFY ADDRESSING
UPDATE
DAG PAGE (DMPG1 OR DMPG2) PROVIDES UPPER 8 BITS OF ADDRESS (OPTIONAL BIT-REVERSE DOES NOT APPLY TO PAGE)
24
MSTAT
PM ADDRESS BUS (EITHER DAG1 OR DAG2)
DM ADDRESS BUS (EITHER DAG1 OR DAG2)

Figure 4-1. Data Address Generator (DAG) Block Diagram

Do not assume that the L registers are automatically initialized to zero for linear addressing. The I, M, L, and B registers contain random values following DSP reset. For each I register used, programs must initialize the corresponding L registers to the appropriate value—either 0 for linear addressing or the buffer length for circular buffer addressing.

On previous 16-bit, fixed-point DSPs (ADSP-218x family), the DAG registers are 14-bits wide, instead of 16-bits wide as on the ADSP-219x DSPs. Because the ADSP-219x DAG registers are 16-bits wide, the DAGs do not need to perform the zero padding on I and L register writes to memory or the sign extension on M register writes to memory that is required for previous ADSP-218x family DSPs.

Setting DAG Modes

The MSTAT register controls the operating mode of the DAGs. Figure A-2 on page A-10 lists all the bits in MSTAT. The following bits in Mode Status (MSTAT) register control Data Address Generator modes:

- **Bit-reverse addressing enable.** Bit 1 (BIT_REV) enables bit-reversed addressing (if 1) or disables bit-reversed addressing (if 0) for DAG1 Index (I0-I3) registers.
- **Secondary registers for DAG.** Bit 6 (SEC_DAG) selects the corresponding secondary register set (if 1) or selects the corresponding primary register set—the set that is available at reset—(if 0).

Secondary (Alternate) DAG Registers

Each DAG has an secondary register set. To facilitate fast context switching, the DSP includes secondary register sets for data, results, and data address generator registers. The SEC_DAG bit in the MSTAT register controls when secondary DAG registers become accessible. While inaccessible, the contents of secondary registers are not affected by DSP operations. Figure 4-2 on page 4-5 shows the DAG's primary and secondary register sets.

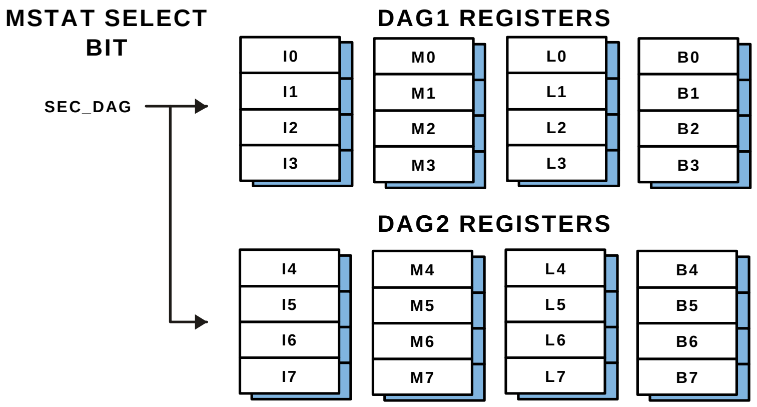


Figure 4-2. Data Address Generator Primary and Alternate Registers

The secondary register sets for the DAGs are described in this section. For more information on secondary data and results registers, see "Secondary (Alternate) Data Registers" on page 2-63.

There are no secondary Data Memory Page (DMPGx) registers. Changing between primary and secondary DAG registers does not affect DMPGx register settings.

System power-up and reset enable the primary set of DAG address registers. To enable or disable the secondary address registers, programs set or clear the SEC_DAG bit in MSTAT. The instruction set provides three methods for swapping the active set. Each method incurs a latency, which is the delay between the time the instruction affecting the change executes until the time the change takes effect and is available to other instructions. Table A-3 on page A-6 shows the latencies associated with each method.

When switching between primary and secondary DAG registers, the program needs to account for the latency associated with the method used. For example, after the MSTAT = data12; instruction, a minimum of three cycles of latency occur before the mode change takes effect. For this method, the program must issue at least three instructions after MSTAT = 0x20; before attempting to use the other set of DAG registers.

The ENA/DIS mode instructions are more efficient for enabling and disabling DSP modes because these instructions incur no cycles of effect latency. For example:

```
CCODE = 0x9; NOP;
IF SWCOND JUMP do_data;          /* Jump to do_data */
do_data:
   ENA SEC_REG;                  /* Switch to 2nd Dregs */
   ENA SEC_DAG;                  /* Switch to 2nd DAGs */
   AX0 = DM(buffer);             /* if buffer empty, */
   AR = PASS AX0;                /* go right to fill */
   IF EQ JUMP fill;              /* and get new data */
   RTI;
fill:                            /* fill routine */
   NOP;
buffer:                         /* buffer data */
   NOP;
```

On previous 16-bit, fixed-point DSPs (ADSP-218x family), there are no secondary DAG registers.

Bit-Reverse Addressing Mode

The BIT_REV bit in the MSTAT register enables bit-reverse addressing mode—outputting addresses in bit-reversed order. When BIT_REV is set (1), the DAG bit-reverses 16-bit addresses output from DAG1 Index registers—I0, I1, I2, and I3. Bit-reverse addressing mode affects post-modify operations. For more information, see "Addressing with Bit-Reversed Addresses" on page 4-16.

Data Memory Page Registers (DMPGx)

The DAGs and their associated page registers generate 24-bit addresses for accessing the data needed by instructions. For data accesses, the DSP's unified memory space is organized into 256 pages, with 64K locations per page. The page registers provide the eight MSBs of the 24-bit address, specifying the page on which the data is located. The DAGs provide the sixteen LSBs of the 24-bit address, specifying the exact location of the data on the page.

- The Data Memory Page (DMPG1) register is associated with DAG1 (registers I0–I3) indirect memory accesses and immediate addressing.
- The DMPG2 page register is associated with DAG2 (registers I4–I7) indirect memory accesses.

At power up, the DSP initializes both page registers to 0x0. Initializing page registers only is necessary when the data is located on a page other than the current page. Programs should set the corresponding page register when initializing a DAG index register to set up a data buffer.

For example,

```
DMPG1 = 0x12;           /* set page register or the syntax */
                        /* "DMPG1 = page(data_buffer);     */
                        /* for relative addressing         */
I2 = 0x3456;            /* init data buffer; 24b addr=0x123456 */
L2 = 0;                 /* define linear buffer */
M2 = 1;                 /* increment address by one */
```

```
                        /* two stall cycles inserted here */
DM(I2 += M2) = AX0;     /* write data to buffer and update I2 */
```

DAG register (DMPGx, Ix, Mx, Lx, Bx) loads can incur up to two stall cycles when a memory access based on the initialized register immediately follows the initialization.

To avoid stall cycles, programs could perform the memory access sequence as follows:

```
I2 = 0x3456;       /* init data buffer; 24b addr=0x123456 */
L2 = 0;            /* define linear buffer */
M2 = 1;            /* increment address by one */
DMPG1 = 0x12;      /* set page register or use the syntax: */
                   /* ”DMPG1 = page(data_buffer);” */
                   /* for relative addressing */AX0 = 0xAAAA; */
AR = AX0 - 1;
DM(I2 += M2) = AR;  /* write data to buffer and update I2 */
```

Typically, programs load both page registers with the same page value (0-255), but programs can increase memory flexibility by loading each with a different page value. For example, by loading the page registers with different page values, programs could perform high-speed data transfers between pages.

Changing the contents of the DAG page registers is not automatic and requires explicit programming.

Using DAG Status

As described in “Addressing Circular Buffers” on page 4-12, the DAGs can provide addressing for a constrained range of addresses, repeatedly cycling through this data (or buffer). A buffer overflow (or wrap around) occurs each time the DAG circles past the buffer’s base address.

Unlike the computational units and program sequencer, the DAGs do not generate status information. So, the DAGs do not provide buffer overflow information when executing circular buffer addressing. If a program requires status information for the circular buffer overflow condition, the program should implement an address range checking routine to trap this condition.

DAG Operations

The DSP's DAGs perform several types of operations to generate data addresses. As shown in Figure 4-1 on page 4-3, the DAG registers and the MSTAT register control DAG operations. The following sections provide details on DAG operations:

- "Addressing with DAGs" on page 4-9
- "Addressing Circular Buffers" on page 4-12
- "Addressing with Bit-Reversed Addresses" on page 4-16
- "Modifying DAG Registers" on page 4-20

An important item to note from Figure 4-1 on page 4-3 is that each DAG automatically uses its Data Memory Page (DMPGx) register to include the page number as part of the output address. By including the page, DAGs can generate addresses for the DSP's entire memory map. For details on these address adjustments, see "Data Memory Page Registers (DMPGx)" on page 4-7.

Addressing with DAGs

The DAGs support two types of modified addressing—generating an address that is incremented by a value or a register. In pre-modify addressing, the DAG adds an offset (modifier), either an M register or an immediate value, to an I register and outputs the resulting address.

Pre-modify addressing does not change (or update) the I register. The other type of modified addressing is post-modify addressing. In post-modify addressing, the DAG outputs the I register value unchanged, then the DAG adds an M register or immediate value, updating the I register value. Figure 4-3 on page 4-10 compares pre- and post-modify addressing.

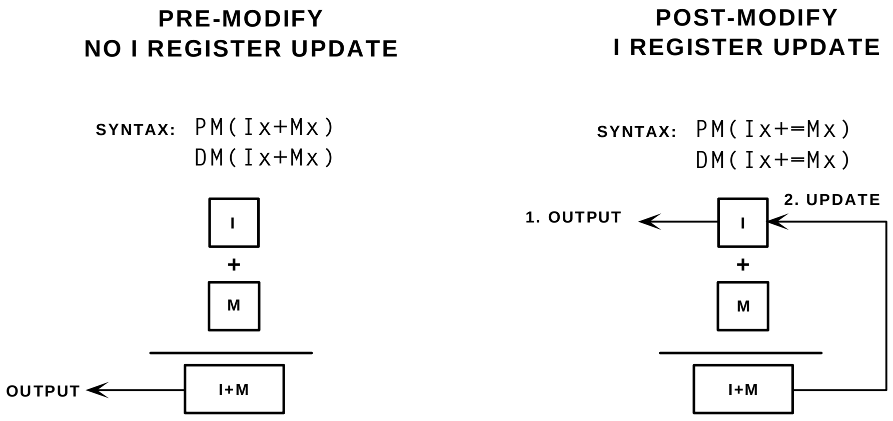


Figure 4-3. Pre-Modify and Post-Modify Operations

The difference between pre-modify and post-modify instructions in the DSP's assembly syntax is the operator that appears between the index and modify registers in the instruction. If the operator between the I and M registers is += (plus-equals), the instruction is a post-modify operation. If the operator between the I and M registers is + (plus), the instruction is a pre-modify without update operation. The following instruction accesses the program memory location indicated by the value in I7 and writes the value I7 plus M6 to the I7 register:

```
AX0 = PM(I7+=M6);   /* Post-modify addressing with update */
```

By comparison, the following instruction accesses the program memory location indicated by the value I7 plus M6 and does not change the value in I7:

```
AX0 = PM(I7+M6);   /* Pre-modify addressing without update */
```

Modify (M) registers can work with any index (I) register in the same DAG (DAG1 or DAG2). For a list of I and M registers and their DAGs, see Figure 4-2 on page 4-5.

On previous 16-bit, fixed-point DSPs (ADSP-2180 family), the assembly syntax uses a comma between the DAG registers (I,M indicates post-modify) to select the DAG operation. While the legacy support in the ADSP-219x assembler permits this syntax, updating ported code to use the ADSP-219x syntax (I+M for pre-modify and I+=M for post-modify) is advised.

Instructions can use a signed 8-bit number (immediate value), instead of an M register, as the modifier. For all single data access operations, modify values can be from an M register or an 8-bit immediate value. The following example instruction accepts up to 8-bit modifiers:

```
AX0 = DM(I1+0x40;                /* DM address = I1+0x40 */
```

Instructions that combine DAG addressing with computations do not accept immediate values for the modifier. In these instructions (multifunction computations), the modify value must come from an M register:

```
AR = AX0+AY0,PM(I4+=m5) = AR;   /* PM address = I4, I4=I4+M5 */
```

Note that pre- and post-modify addressing operations do not change the memory page of the address. For more information, see "Data Memory Page Registers (DMPGx)" on page 4-7.

Addressing Circular Buffers

The DAGs support addressing circular buffers—a range of addresses containing data that the DAG steps through repeatedly, "wrapping around" to repeat stepping through the range of addresses in a circular pattern. To address a circular buffer, the DAG steps the index pointer (I register) through the buffer, post-modifying and updating the index on each access with a positive or negative modify value (M register or immediate value). If the index pointer falls outside the buffer, the DAG subtracts or adds the length of the buffer from or to the value, wrapping the index pointer back to the start of the buffer. The DAG's support for circular buffer addressing appears in Figure 4-1 on page 4-3, and an example of circular buffer addressing appears in Figure 4-4 on page 4-14.

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 a circular buffer.

ⓘ Circular buffering may only use post-modify addressing. The DAG's architecture, as shown in Figure 4-1 on page 4-3, cannot support pre-modify addressing for circular buffering, because circular buffering requires that the index be updated on each access.

⃠ Do not place the index pointer for a circular buffer such that it crosses a memory page boundary during post-modify addressing. All memory locations in a circular buffer must reside on the same memory page. For more information on the DSP's memory map, see "Memory" on page 5-1.

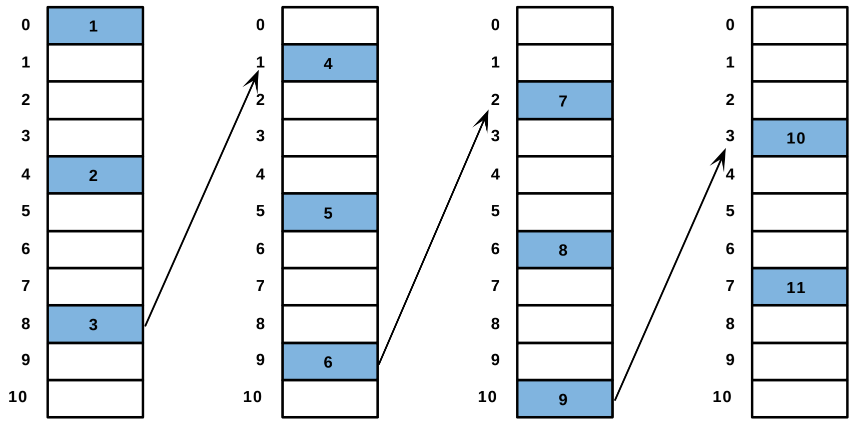
As shown in Figure 4-4 on page 4-14, programs use the following steps to set up a circular buffer:

1. Load the memory page address into the selected DAG's `DMPGx` register. This operation is needed only once per page change in a program.
2. Load the starting address within the buffer into an I register in the selected DAG.
3. Load the modify value (step size) into an M register in the corresponding DAG as the I register. For corresponding registers list, see Figure 4-2 on page 4-5.
4. Load the buffer's length into the L register that corresponds to the I register. For example, `L0` corresponds to `I0`.
5. Load the buffer's base address into the B register that corresponds to the I register. For example, `B0` corresponds to `I0`.

ⓘ The DAG B registers are system control registers. To load these registers, use the `Reg()` instruction.

```
.Section/DM seg_data;
.VAR coeff_buffer[11] = 0,1,2,3,4,5,6,7,8,9,10;
.Section/PM seg_code;
DMPG1 = Page(coeff_buffer);/* Set the memory page */
I0 = coeff_buffer;         /* Set the current addr */
M1 = 4;                    /* Set the modify value */
L0 = Length(coeff_buffer); /* If L = 0 buffer is linear */
AX0 = I0;                  /* Copy the base addr into AX0 */
Reg(B0) = AX0;             /* Set the buffer's base addr */
AR = AX1 And AY0;
AR = DM(I0 += M1);         /* Read 1st buffer location */

CNTR = 11; Do my_cir_buffer Until CE;
                           /* sets up a loop accessing the buffer */
AX0 = DM(I0,M1);           /* access using post modify addressing */
Nop;                       /* other instructions in the loop */
my_cir_buffer: Nop;        /* end of my_cir_buffer loop */
```



THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
NOTE THAT "0" ABOVE IS ADDRESS DM(0X1000). THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Figure 4-4. Circular Data Buffers

After this setup, the DAGs use the modulus logic in Figure 4-1 on page 4-3 to process circular buffer addressing.

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 value is outside the buffer length, the DAG subtracts (positive) or adds (negative) the L register value before writing the updated index value to the I register.

In equation form, these post-modify and wrap around operations work as follows:

If M is positive:

> **Inew** = **Iold** + **M** if Iold + M < Buffer base + length (end of buffer)
>
> **Inew** = **Iold** + **M** – **L** if Iold + M Š Buffer base + length (end of buffer)

If M is negative:

> **Inew** = **Iold** + **M** if Iold + M Š Buffer base (start of buffer)
>
> **Inew** = **Iold** + **M** + **L** if Iold + M < Buffer base (start of buffer)

The DAGs use all types of DAG registers for addressing circular buffers. These registers operate as follows for circular buffering:

- 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. The M register can be any M register in the same DAG as the I register. The modify value also can be an imme-

diate value instead of an M register. The size of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.

- The length (L) register sets the size of the circular buffer and the address range that the DAG circulates the I register through. L is positive and cannot have a value greater than 216 – 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 that the DAG compares the modified I value with after each access.

On previous 16-bit, fixed-point DSPs (ADSP-218x family), the DAGs do not have B registers. When porting code that uses circular buffer addressing, add the instructions needed for loading the ADSP-219x B register that is associated with the corresponding circular buffer.

Addressing with Bit-Reversed Addresses

Programs need bit-reversed addressing for some algorithms (particularly FFT calculations) to obtain results in sequential order. To meet the needs of these algorithms, the DAG's bit-reverse addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order.

Bit-reversed address output is available on DAG1, while DAG2 always outputs its address bits in normal, big endian format. Because the two DAGs operate independently, programs can use them in tandem, with one generating sequentially ordered addresses and the other generating bit-reversed addresses, to perform memory reads and writes of the same data.

To use bit-reversed addressing, programs set the `BIT_REV` bit in `MSTAT` (`ENA BIT_REV`). When enabled, DAG1 outputs all addresses generated by its index registers (`I0–I3`) in bit-reversed order. The reversal applies only to the address value DAG1 outputs, not to the address value stored in the index register, so the address value is stored in big endian format. Bit-reversed mode remains in effect until disabled (`DIS BIT_REV`).

Bit reversal operates on the binary number that represents the position of a sample within an array of samples. Using 3-bit addresses, Table 4-1 on page 4-17 shows the position of each sample within an array before and after the bit-reverse operation. Sample `0x4` occupies position `b#100` in sequential order and position `b#001` in bit-reversed order. Bit reversing transposes the bits of a binary number about its midpoint, so `b#001` becomes `b#100`, `b#011` becomes `b#110`, and so on. Some numbers, like `b#000`, `b#111`, and `b#101`, remain unchanged and retain their original position within the array.

Table 4-1. 8-Point Array Sequence Before and After Bit Reversal

Sequential Order		Bit Reversed Order	
Sample (hexadecimal)	Binary	Binary	Sample (hexadecimal)
0x0	b#000	b#000	0x0
0x1	b#001	b#100	0x4
0x2	b#010	b#010	0x2
0x3	b#011	b#110	0x6
0x4	b#100	b#001	0x1
0x5	b#101	b#101	0x5
0x6	b#110	b#011	0x3
0x7	b#111	b#111	0x7

Bit-reversing the samples in a sequentially ordered array scrambles their positions within the array. Bit-reversing the samples in a scrambled array restores their sequential order within the array.

In full 16-bit reversed addressing, bits 7 and 8 of the 16-bit address are the pivot points for the reversal. (Table 4-2)

Table 4-2. 16-Bit Reversed Addressing

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

The Fast Fourier Transform (FFT) algorithm is a special case for bit-reversal. FFT operations often need only a few address bits reversed. For example, a 16-point sequence requires four reversed bits, and a 1024-bit sequence requires ten reversed bits. Programs can bit-reverse address values less than 16-bits—which reverses a specified number of LSBs only. Bit-reversing less than the full 16-bit index register value requires that the program adds the correct modify value to the index pointer after each memory access to generate the correct bit-reversed addresses.

To set up bit-reversed addressing for address values fewer than 16 bits, determine:

1. The **number of bits to reverse** (`N`)—permits calculating the modify value
2. The **starting address of the linear data buffer**—this address must be zero or an integer multiple of the number of bits to reverse (starting address = `0, N, 2N, …`)

3. The **initialization value for the index register**—the bit-reversed value of the first bit-reversed address the DAG outputs
4. The **modify register value** for updating (correcting) the index pointer after each memory access—calculated from the formula: `Mreg = 2`$^{(16-N)}$

The following example, sets up bit-reversed addressing that reverses the eight address LSBs (`N = 8`) of a data buffer with a starting address of `0x0020` (`4N`). Following the described steps, the factors to determine are:

1. The **number of bits to reverse** (`N`)—eight bits (from description)
2. The **starting address of the linear data buffer**—`0x0020` (`4N`) (from description)
3. The **initialization value for the index register**—this is the first bit-reversed address DAG1 outputs (`0x0004`) with bits 15–0 reversed: `0x2000`. (Table 4-3)

Table 4-3. Index Register Initialization Value

0x0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0x2000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

4. The **modify register value** for updating (correcting) the index pointer after each memory access—this is `2`$^{16-N}$, which evaluates to `2`^{8} or `0x0100`.

Listing 4-1 implements this example in assembly code.

Listing 4-1. Bit-reversed addressing, 8 LSBs

```
br_adds: I4=read_in;          /* DAG2 pointer to input samples */
   I0=0x0200;                 /* Base addr of bit_rev output */
   M4=1;                      /* DAG2 increment by 1 */
   M0=0x0100;                /* DAG1 increment for 8-bit rev. */
   L4=0;                     /* Linear data buffer */
   L0=0;                     /* Linear data buffer */
   CNTR=8;                   /* 8 samples */
   ENA BIT_REV;              /* Enable DAG1 bit reverse mode */
   DO brev UNTIL CE;
      AY1=DM(I4+=M4);        /* Read sequentially */
      brev: DM(I0+=M0)=AY1;  /* Write nonsequentially */
   DIS BIT_REV;              /* Disable DAG1 bit reverse mode */
   RTS;                      /* Return to calling routine */
read_in:                     /* input buf, could be .extern */
   NOP;
```

Modifying DAG Registers

The DAGs support an operation that modifies an address value in an index register without outputting an address. The operation, address modify, is useful for maintaining pointers.

The `MODIFY` instruction modifies addresses in any DAG index register (`I0-I7`) without accessing memory. If the I register's corresponding B and L registers are set up for circular buffering, a `MODIFY` instruction performs the specified buffer wrap around (if needed). The syntax for `MODIFY` is similar to post-modify addressing (index+=modifier). `MODIFY` accepts a signed 8-bit immediate values or an M register as the modifier.

The following example adds 4 to `I1` and updates `I1` with the new value:

```
MODIFY(I1+=4);
```

DAG Register Transfer Restrictions

DAG I, M, and L registers are part of the DSP's register group 1 (`Reg1`), register group 2 (`Reg2`), and register group 3 (`Reg3`) register sets; the B registers are in register memory space. Programs may load the DAG registers from memory, from another data register, or with an immediate value. Programs may store DAG registers' contents to memory or to another data register.

While instructions to load and use DAG registers may be sequential, the DAGs insert stall cycles for sequences of instructions that cause instruction pipeline conflicts. The two types of conflicts are:

- Using an I register (or its corresponding L or B registers) within two cycles of loading the I register (or its corresponding L or B registers)
- Using an M register within two cycles of loading the M register

The following code examples and comments demonstrate the conditions under which the DAG inserts stall cycles. These examples also show how to avoid these stall conditions.

```
/* The following sequence of loading and using the DAG */
/* registers does NOT force the DAG to insert stall cycles. */
I0 = 0x1000;
M0 = 1;
L0 = 0xF;
REG(B0) = AX0;
AR = AX0 +AY0;
MR = MX0 * MY0 (SS);
AX1 = DM(I0+=M0);
```

```
/*    This sequence of loading and using the DAG registers */
/*    FORCES the DAG to insert two stall cycles. */
M0 = 1;
L0 = 0xF;
REG(B0) = AX0;
I0 = 0x1000;
AX1 = DM(I0+=M0);      /* DAG inserts two stall cycles here
                       /* until I0 can be used */

/*    This sequence of loading and using the DAG registers */
/*    FORCES the DAG to insert two stall cycles. */
I0 = 0x1000;
L0 = 0xF;
REG(B0) = AX0;
M0 = 1;
AX1 = DM(I0+=M0);      /* DAG inserts two stall cycles here
                       /* until M0 can be used */

/*    This sequence of loading and using the DAG registers */
/*    FORCES the DAG to insert one stall cycle. */
M0 = 1;
L0 = 0xF;
I0 = 0x1000;
REG(B0) = AX0;
AR = AX0 + AY0;
AX1 = DM(I0+=M0);      /* DAG inserts one stall cycle here
                       /* until I0 (corresponds to B0) can be used
*/
```

DAG Instruction Summary

Table 4-4 on page 4-23 lists the DAG instructions. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In Table 4-4 on page 4-23, note the meaning of the following symbols:

- **Dreg, Dreg1, Dreg2** indicate any register file location (register group)
- **Reg1, Reg2, Reg3, or Reg** indicate register group 1, 2, 3, or any register
- **Ia and Mb** indicate DAG1 I and M registers
- **Ic and Md** indicate DAG2 I and M registers
- **Ireg and Mreg** indicate I and M registers in either DAG
- **Imm# and Data#** indicate immediate values or data of the # of bits

Table 4-4. DAG Instruction Summary

Instruction
|DM(Ia += Mb), DM(Ic += Md)| = Reg;
Reg = |DM(Ia += Mb), DM(Ic += Md)|;
|DM(Ia + Mb), DM(Ic + Md)| = Reg;
Reg = |DM (Ia + Mb), DM (Ic + Md)|;
|PM(Ia += Mb), PM(Ic += Md)| = Reg;
Reg = |PM(Ia += Mb), PM(Ic += Md)|;
|PM(Ia + Mb), PM(Ic + Md)| = Reg;
Reg = |PM(Ia + Mb), PM(Ic + Md)|;
DM(Ireg1 += Mreg1) = |Ireg2, Mreg2, Lreg2|, |Ireg2, Mreg2, Lreg2| = Ireg1;
Dreg = DM(Ireg += <Imm8>);

Table 4-4. DAG Instruction Summary (Cont'd)

Instruction
DM(Ireg += <Imm8>) = Dreg;
Dreg = DM(Ireg + <Imm8>);
DM(Ireg + <Imm8>) = Dreg;
|DM(Ia += Mb), DM (Ic += Md)| = <Data16>;
|PM (Ia += Mb), PM (Ic += Md)| = <Data24>:24;
|Modify (Ia += Mb), Modify (Ic += Md)|;
Modify (Ireg += <Imm8>);

5 MEMORY

This chapter provides the following sections:

Overview

The ADSP-2191 contains a large internal memory and provides access to external memory through the DSP's external port. This chapter describes the internal memory and how to use it. For information on configuring, connecting, and timing accesses to external memory, see "Interfacing to External Memory" on page 7-15. There are 64K words of internal SRAM memory on the **ADSP-2191**, 32K words on the **ADSP-2195** and 16K words on the **ADSP-2196**. Memory is divided into 16-bit blocks for data storage and 24-bit blocks for data and instruction storage. The ADSP-2191 features two 24-bit blocks and two 16-bit blocks, each 16K. The ADSP-2195 features one 24-bit block and one 16-bit block. The ADSP-2196 has one 24-bit block and one 14-bit block, 8K in size. Additionally, there is a 1K word ROM for boot routines.

Including internal and external memory, the DSP can address 16M words of memory space. External memory connects to the DSP's external port, which extends the DSP's address and data buses off the DSP. The DSP can make 16- or 24-bit accesses to external memory for data or instruc-

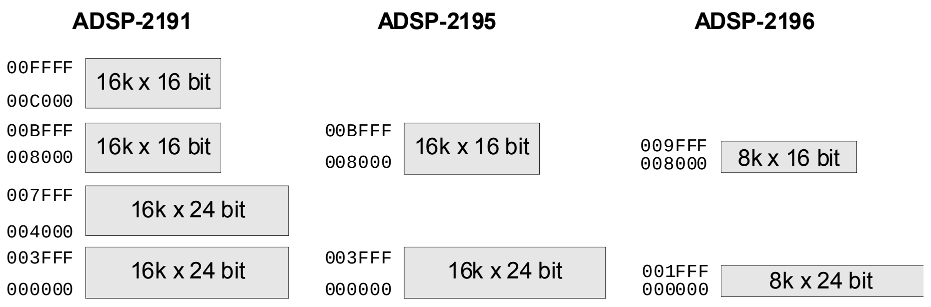


Figure 5-1. On-Chip Memory Layout of ADSP-219x Derivatives

tions. The DSP's external port automatically packs external data into the appropriate word width during data transfer. Table 5-1 on page 5-2 shows the access types and words for DSP external memory accesses.

Table 5-1. Internal-to-External Memory Word Transfers

Word Type	Transfer Type
Instruction	24-bit word transfer[1]
Data	16-bit word transfer

1 Each packed 24-bit word requires two transfers over 16-bit bus.

Most microprocessors use a single address and data bus for memory access. This type of memory architecture is called Von Neumann architecture. But, DSPs require greater data throughput than Von Neumann architecture provides, so many DSPs use memory architectures that have separate buses for program and data storage. The two buses let the DSP get a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

ADSP-219x family DSPs go a step farther by using a modified Harvard architecture. This architecture has program and data buses, but provides a single, unified address space for program and data storage. Although the 16-bit DM bus carries data only, the 24-bit PM bus handles instructions or data, allowing dual-data access.

Unlike ADSP-218x DSPs, ADSP-219x DSPs have unified memory address space. 24-bit and 16-bit memories can be accessed by both the PM and the DM bus system. Although it is not meaningful to talk about “Program Memory (PM)” and “Data Memory (DM)” in this architecture, the terms PM and DM are still used to distinguish between the two bus systems.

DSP core and DMA-capable peripherals share accesses to internal memory. Each block of memory can be accessed by the DSP core and DMA-capable peripherals in every cycle, but a DMA transfer is held off if contending with the DSP core for access.

A memory access conflict can occur when the processor core attempts two accesses to the same internal memory block in the same cycle. When this conflict happens, an extra cycle is incurred. The DM bus access completes first and the PM bus access completes in the following (extra) cycle.

During a single-cycle, dual-data access, the processor core uses the independent PM and DM buses to simultaneously access data from both memory blocks. Though dual-data accesses provide greater data throughput, there are some limitations on how programs may use them. The limitations on single-cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.

  If the core tries to access two words from the same memory block (over the same bus) for a single instruction, an extra cycle is needed. For more information on how the buses access these blocks, see Figure 5-2 on page 5-5.

- The data access execution may not conflict with an instruction fetch operation.
- If the cache contains the conflicting instruction, the data access completes in a single-cycle and the sequencer uses the cached instruction. If the conflicting instruction is not in the cache, an extra cycle is needed to complete the data access and cache the conflicting instruction. For more information, see "Instruction Cache" on page 3-9.

Efficient memory usage relies on how the program and data are arranged in memory and varies with how the program accesses the data. For the most efficient (single-cycle) accesses, use the above guidelines for arranging data in memory.

Internal Address and Data Buses

As shown in Figure 5-2 on page 5-5, the DSP has two internal buses connected to its internal memory, the PM bus and DM bus. The I/O processor—*which is the global term for the DMA controllers, DMA channel arbitration, and peripheral-to-bus connections*—also is connected to the internal memory and external port. The PM bus, DM bus, and I/O processor (for DMAs) share two memory ports; one for each block. Memory accesses from the DSP's core (computational units, data address generators, or program sequencer) use the PM or DM buses. The I/O processor also uses the DM bus for non-DMA memory accesses, such as host port direct reads and writes, but uses a separate connection to the memory's ports for DMA transfers. Using this separate connection and cycle-stealing DMA, the I/O processor can provide data transfers between internal memory and the DSP's communication ports (external port, host port, serial ports, SPI ports, and UART port) without hindering the DSP core's access to memory.

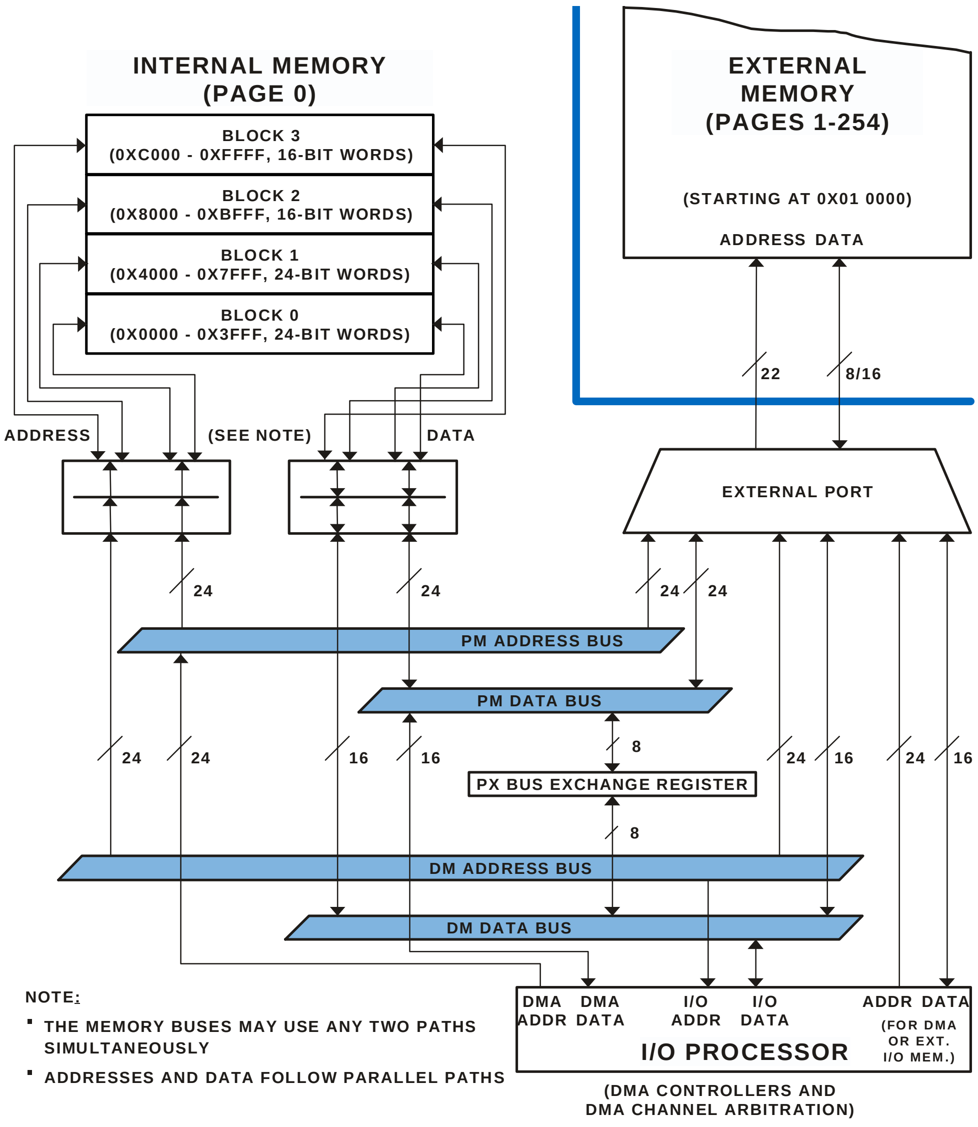

INTERNAL MEMORY (PAGE 0)
BLOCK 3 (0XC000 - 0XFFFF, 16-BIT WORDS)
BLOCK 2 (0X8000 - 0XBFFF, 16-BIT WORDS)
BLOCK 1 (0X4000 - 0X7FFF, 24-BIT WORDS)
BLOCK 0 (0X0000 - 0X3FFF, 24-BIT WORDS)
EXTERNAL MEMORY (PAGES 1-254)
(STARTING AT 0X01 0000)
ADDRESS DATA
22
8/16
ADDRESS
(SEE NOTE)
DATA
EXTERNAL PORT
24
24
24
24
PM ADDRESS BUS
PM DATA BUS
24
24
16
16
8
PX BUS EXCHANGE REGISTER
8
24
16
24
16
DM ADDRESS BUS
DM DATA BUS
NOTE:
THE MEMORY BUSES MAY USE ANY TWO PATHS SIMULTANEOUSLY
ADDRESSES AND DATA FOLLOW PARALLEL PATHS
DMA ADDR
DMA DATA
I/O ADDR
I/O DATA
ADDR DATA
(FOR DMA OR EXT. I/O MEM.)
I/O PROCESSOR
(DMA CONTROLLERS AND DMA CHANNEL ARBITRATION)

Figure 5-2. ADSP-2191 Memory and Internal Buses Block Diagram

While the DSP's internal memory is divided into **blocks**, the DSP's external memory spaces is divided into **banks**. External memory banks have associated memory select ($\overline{\texttt{MSx}}$) pins and may be configured for size, clock ratio, and access waitstates. For more information, see "External Memory Space" on page 5-13.

The DSP core's PM bus and DM bus and I/O processor can try to access internal memory space or external memory space in the same cycle. The DSP has an arbitration system to handle this conflicting access. Arbitration is fixed at the following priority: (highest priority) DM bus, PM bus, and (lowest priority) I/O processor. Also, I/O processor accesses may not be sequential (beyond each burst access), so the DSP core's buses are never held off for more than four cycles.

External Address and Data Buses

Figure 5-2 on page 5-5 also shows that the PM buses, DM buses, and I/O processor have access to the external bus (pins `DATA15-0`, `ADDR21-0`) through the DSP's external port. The external port provides access to system (off-chip) memory and peripherals. This port also lets the DSP access shared memory if connected in a multi-DSP system.

Addresses for the PM and DM buses come from the DSP's program sequencer and Data Address Generators (DAGs). The program sequencer and DAGs supply 24-bit addresses for locations throughout the DSP's memory spaces. The DAGs supply addresses for data reads and writes on both the PM and DM address buses, while the program sequencer uses only the PM address bus for sequencing execution.

The external address bus is 22 bits wide on the ADSP-2191(LQFP or PBGA 144-lead packages), so the upper two bits of address do not get generated off-chip.

For memory accesses by different functional blocks of the DSP, the upper eight bits of the address—the page number—come from different page registers. The Data Address Generators—DAG1 and DAG2—each are associated with a Data Memory Page (DMPG1, DMPG2) register, the program sequencer has an Indirect Jump Page register (IJPG) for indirect jumps, and I/O memory uses the I/O Memory Page (IOPG) register. For more information on address generation, see "Program Sequencer" on page 3-1 or "Data Address Generators (DAGs)" on page 4-1.

Because the DSP's blocks of internal memory are of differing widths, placing 16-bit data in 24-bit blocks leaves some space unchanged. On-chip 16-bit memory blocks cannot store instructions. For more information on how the DSP works with memory words, see "Internal Memory Space" on page 5-12.

The PM data bus is 24 bits wide, and the DM data bus is 16 bits wide. Both data buses can handle data words (16-bit), but only the PM data bus carries instruction words (24-bit).

At the processor's external port, the DSP multiplexes the three memory buses (PM, DM, and I/O) to create a single off-chip data bus (DATA15-0) and address bus (ADDR21-0).

The external port interface supports 8-bit and 16-bit memories. Both types may store data and instructions.

Internal Data Bus Exchange

The internal data buses let programs transfer the contents of one register to another or to any internal memory location in a single cycle. As shown in Figure 5-3 on page 5-8, the PM Bus Exchange (PX) register permits data to flow between the PM and DM data buses. The PX register holds the lower eight bits during transfers between the PM and DM buses. The alignment of PX register to the buses appears in Figure 5-3 on page 5-8.

The PX register is a register group 3 (REG3) register and is accessible for register-to-register transfers.

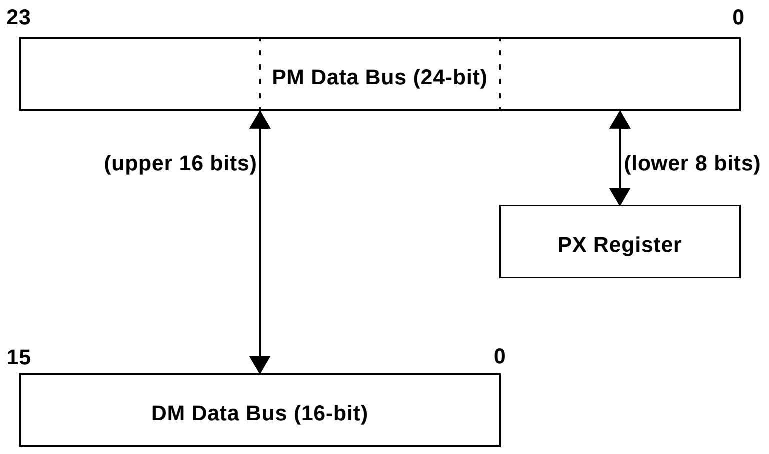


Figure 5-3. PM Bus Exchange (PX) Registers

For transferring data from the PMD bus, the `PX` register is:

1. Loaded automatically whenever data (not an instruction) is read from program memory to any register. For example:

   ```
   AX0 = PM(I4,M4);
   ```

   In this example, the upper 16 bits of a 24-bit program memory word are loaded into `AX0` and the lower eight bits are automatically loaded into `PX`.

2. Read out automatically as the lower eight bits when data is written to program memory. For example:

   ```
   PM(I4,M4) = AX0;
   ```

   In this example, the 16 bits of `AX0` are stored into the upper 16 bits of a 24-bit program memory word. The eight bits of `PX` are automatically stored to the eight lower bits of the memory word.

For transferring data from the DMD bus, the PX register may be:

- Loaded with a data move instruction, explicitly specifying the PX register as the destination. The lower eight bits of the data value are used and the upper eight are discarded.

  ```
  PX = AX0;
  ```

- Read with a data move instruction, explicitly specifying the PX register as a source. The upper eight bits of the value read from the register are all zeroes.

  ```
  AX0 = PX;
  ```

If the PM bus is used to read from 16-bit memory, the PX register is filled with zeros. Whenever any register is written out to program memory, the source register supplies the upper 16 bits. The contents of the PX register are automatically added as the lower eight bits. If these lower eight bits of data to be transferred to program memory (through the PMD bus) are important, programs should load the PX register from DMD bus before the program memory write operation. PM bus reads from 16-bit memories clear the PX register; this includes PM reads of off-chip address space if the E_DFS bit is cleared. DM bus transfers never use the PX register.

ADSP-2191 DSP Memory Map

The ADSP-2191 memory map appears in Figure 5-4 and has multiple memory spaces: internal memory space, external memory space, system control register memory space, I/O memory space, and boot memory space. register to the buses appears in This section provides the following topics:

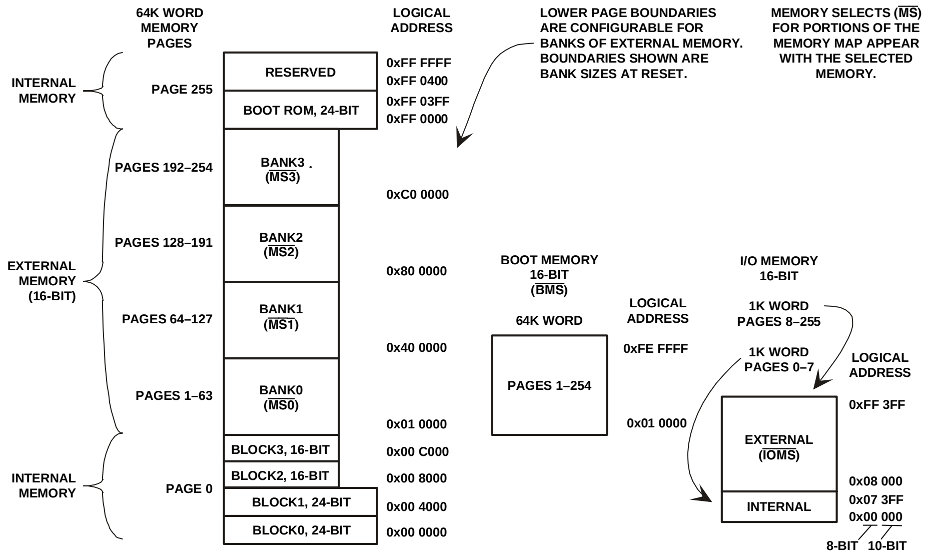


Figure 5-4. ADSP-2191 Memory Map

This section provides the following topics:

- "Overview" on page 5-11
- "Internal Memory Space" on page 5-12
- "External Memory Space" on page 5-13
- "System Control Registers" on page 5-15
- "I/O Memory Space" on page 5-16
- "Boot Memory Space" on page 5-16
- "Shadow Write FIFO" on page 5-17

Overview

These spaces have the following definitions:

- **Internal memory space.** The internal RAM space ranges from address `0x00 0000` through `0x00 FFFF`. The internal (boot kernel) ROM space ranges from address `0xFF 0000` through `0xFF 0400`. Internal memory space refers to the DSP's on-chip SRAM.

- **External memory space.** This space ranges from address `0x01 0000` through `0xFE FFFF`. External memory space refers to off-chip memory that is accessed through data move instructions and is attached to the DSP's external address (`ADDR21-0`) and data (`DATA15-0`) buses. During accesses to external memory space, the DSP generates memory select ($\overline{\text{MS3-0}}$) signals for the memory bank that corresponds to the address.

- **System control registers.** This space is separate from other memory spaces and does not appear in Figure 5-4. This space contains ungrouped registers; not part of a register group. For more information, see "System Control Registers" on page 5-15.

- **I/O memory space.** This space is separate from other memory spaces and has an address range from address `0x00:0000` through `0xFF:03FF`. During accesses to off-chip I/O memory space, the DSP generates an I/O Memory Select ($\overline{\text{IOMS}}$) signal. For more information, see "I/O Memory Space" on page 5-16.

- **Boot memory space.** This space is separate from other memory spaces and has an address range from address `0x01 0000` through `0xFE FFFF`. During accesses to boot memory space, the DSP generates a Boot Memory Select ($\overline{\text{BMS}}$) signal. For more information, see "Boot Memory Space" on page 5-16.

Internal Memory Space

The DSP's internal memory space contains up to 64K words of memory, which occupy Page 0 on the DSP's memory map. The memory map is a unified, continuous address range, but some features of the DSP's architecture lead to these block and page distinctions within the map. These distinctions include:

- **Internal memory block width.** 24-bit blocks can contain instructions and data. 16-bit blocks can contain data only.
- **Internal bus width.** The PM data bus is 24 bits wide, and the DM data bus is 16 bits wide. While either bus can access either internal memory block for data, only the PM bus can fetch instructions.
- **Data Address Generators.** DAG1 and DAG2 each are associated with a DAG page (`DMPG1`, `DMPG2`) register and generate addresses for Block 1 and Block 2. Both DAGs can generate external memory addresses.
- **Page size.** Architectural constraints (which are described in the Program Sequencer and Data Address Generators chapters) lead to 64K-word page segmentation of memory—a 16-bit address range per page. To move beyond a page range requires changing a value in a page register. These registers hold the upper eight bits of the 24-bit address. There are page registers associated with internal/external/boot memory space and I/O memory space. These registers include: `DMPGx`, `IJPG`, and `IOPG`.

To execute programs and use data in internal memory, the ADSP-2191 operates similarly to previous ADSP-218x DSPs. For internal memory operations, paging is not required, and the page registers remain at their reset values (Page 0).

The DSP's memory architecture permits either bus to access either internal memory block and also permits dual accesses—a single cycle operation where each bus accesses a block of memory. To arbitrate simultaneous accesses, the memory interface:

- Processes a memory read before memory write
- Processes a DM bus access before a PM bus access

Because the internal PM and DM buses are multiplexed at the DSP's external port, external memory accesses differ slightly from internal memory accesses. For more information, see "External Memory Space" on page 5-13.

Also on-chip, the DSP has an internal boot kernel ROM on memory Page 255. Programs should treat this area as reserved and should not access this area at runtime.

External Memory Space

The DSP's external memory space can address four banks of memory, which contain Page 1 through Page 254 on the DSP's memory map. Programs can configure the number of 64K-word pages per bank, but the addresses for each page are fixed and are part of the unified, continuous address range. For more information on accessing pages through page registers, see "Internal Memory Space" on page 5-12.

Though external memory is part of the same unified address and page register system as internal memory, the DSP configures and controls access to internal and external memory differently. Items that are unique to external memory accesses include:

- **Memory bank and space selects** ($\overline{\texttt{MS3-0}}$, $\overline{\texttt{BMS}}$, $\overline{\texttt{IOMS}}$)**.** The DSP automatically activates an external memory bank or space's select line for each access. For external memory accesses, the DSP activates the bank's select line (Bank 0=$\overline{\texttt{MS0}}$ through Bank 3= $\overline{\texttt{MS3}}$) based on the memory page of the access.

- **Waitstates.** The DSP can apply a selectable number of waitstates for accesses to each external memory bank or space and supports several waitstate modes.

- **Memory bank starting page (bank size).** The DSP can select the starting memory page for each external memory bank, allowing the size of each bank to be configured.

- **Bus arbitration.** The DSP must be bus master to access external memory. To grant bus mastership to other devices, the DSP has bus request ($\overline{\texttt{BR}}$) and bus grant ($\overline{\texttt{BG}}$) pins. If the DSP is stalled, waiting to regain bus mastership, the DSP signals this state with the bus grant hung ($\overline{\texttt{BGH}}$) pin.

- **External memory access latencies.** The DSP's core and peripherals pass data to each other over bus interfaces. Accesses over these interfaces (for example, a core read or write access to external memory) involve some interface latencies. These latencies vary, depending on the type of access. For a list of external memory access latencies, see Table 7-10 on page 7-26.

To execute programs and use data in external memory, the ADSP-2191 operates differently from previous ADSP-218x DSPs—paging is required, the interface has more latencies, and (depending on external bus configuration) packing may be required. For more information on these latencies, see "Memory Interface Timing" on page 7-24.

In the case of an external memory access, the core behaves differently during a read or a write. During a read, the core stalls until the data is returned from the PM or DM bus. During a write access, two contexts should be considered (1) Any previous write access has been completed (2) A previous write access is still pending in the external access bridge.

The second case is the consequence of the first. In context 1, the core is stalled only one cycle (by the core memory controller when the external access is detected), and then relaxed so that even if the external access did not complete the transfer, the core can still run until another external access is requested. This "posted write" may be a problem for the application standpoint (the core is not aware that the write is not completed).

Thus, in the case of consecutive writes, a single write followed by a single write, the first write is posted and the second write is held until the pending write completion, at which time it is posted.

To prevent consecutive external accesses from stalling the core, as a workaround, the `E_WPF` bit of the internal I/O mapped register (`E_STAT`) flags any pending write so that the core has the possibility to check the last write transaction status.

System Control Registers

The DSP has a separate memory space for system control registers. These registers support DSP core operations. The registers in this space include the DAG Base (`Bx`) registers and the Cache Control (`CACTL`) register. For more information, see "Length (Lx) Registers and Base (Bx) Registers" on page A-21 and "Cache Control (CACTL) Register" on page A-19. To access system control registers, programs use system control register read/write instructions (`Reg()`).

I/O Memory Space

The DSP has an I/O memory space for internal I/O memory-mapped registers and external I/O memory-mapped devices. Similar to—but entirely separate from—internal and external memory, the addressing of I/O memory is divided into 1K-word pages with Pages 0–7 on-chip and Pages 8–255 off-chip. Programs select an I/O memory page with the `IOPG` register. To access I/O memory, programs use the I/O memory read/write instructions (`IO()`).

The on-chip I/O memory contains memory-mapped registers for control, status, and data buffers of DSP peripherals (external port, host port, serial ports, serial peripheral interface ports, and UART port) and peripheral DMA.

The off-chip I/O memory is for external memory-mapped devices that use the I/O memory interface to communicate with the DSP. If off-chip, the peripherals are attached to the DSP's external address (`ADDR21-0`) and data (`DATA15-0`) buses. During accesses to off-chip I/O memory space, the DSP generates an I/O Memory Select ($\overline{\texttt{IOMS}}$) signal.

The **I/O processor**—*which is the global term for the DMA controllers, DMA channel arbitration, and peripheral-to-bus connections*—and **I/O memory**—*which contains the control, status, and buffer registers for the I/O processor*—are very different things. The I/O processor does not use the `IOPG` register for DMA, instead the I/O processor uses DMA page information from a DMA's descriptor. Also, the I/O processor cannot perform DMA to I/O memory; only the DSP core or a host may read or write I/O memory.

Boot Memory Space

The DSP has a separate external memory space for mapping a boot ROM or FLASH and booting the DSP from this device. This space is separate from other memory spaces and has an address range from address

0x01 0000 through 0xFE FFFF. Boot memory space refers to off-chip ROM memory that is accessed when the DSP boots from ROM and is attached to the DSP's external address (ADDR21-0) and data (DATA15-0 or DATA7-0) buses. During accesses to boot memory space, the DSP generates a boot memory select ($\overline{\text{BMS}}$) signal.

Although the most common usage for boot memory space is boot loading at system restart, this memory space also can be accessed at runtime. For more information, see "Using Boot Memory Space" on page 7-14.

Shadow Write FIFO

Because the DSP's internal memory must operate at high speeds, writes to the memory do not go directly into the internal memory. Instead, writes go to a two-deep FIFO called the shadow write FIFO.

When an internal memory write cycle occurs, the DSP loads the data in the FIFO from the previous write into memory, and the new data goes into the FIFO. This operation is transparent, because any reads of the last two locations written are intercepted and routed to the FIFO.

Because the ADSP-2191's shadow write FIFO automatically pushes the write to internal memory as soon as the write does not compete with a read, this FIFO's operation is completely transparent to programs, except in software reset/restart situations. To ensure correct operation after a software reset, software must perform two "dummy" writes (repeat last write per block) to internal memory before writing the software reset bit.

Data Move Instruction Summary

Table 5-2 on page 5-18 lists the data move instructions. For more information on assembly language syntax, see the ADSP-219x *DSP Instruction Set Reference*. In Table 5-2 on page 5-18, note the meaning of the following symbols:

- **Dreg, Dreg1, Dreg2** indicate any register file location (register group)
- **Reg1, Reg2, Reg3, or Reg** indicate register group 1, 2, 3, or any register
- **Ia and Mb** indicate DAG1 I and M registers
- **Ic and Md** indicate DAG2 I and M registers
- **Ireg and Mreg** indicate I and M registers in either DAG
- **Imm# and Data#** indicate immediate values or data of the # of bits

Table 5-2. Data/Register Move Instruction Summary

Instruction
Reg = Reg;
|DM(<Addr16>), PM(<Addr16>)| = |Dreg, Ireg, Mreg|;
|Dreg, Ireg, Mreg| = |DM(<Addr16>), PM(<Addr16>)|;
|<Dreg>, <Reg1>, <Reg2>| = <Data16>;
Reg3 = <Data12>;
Io(<Addr10>) = Dreg;
Dreg = Io (<Addr10>);
Reg(<Addr8>) = Dreg;
Dreg = Reg(<Addr8>);

6 I/O PROCESSOR

The ADSP-2191 I/O processor manages interaction between the on-chip peripherals and the DSP core. Direct Memory Access (DMA) enables high data throughput between the memory and peripherals. An enhanced interrupt controller guarantees fast signaling of interrupt events from the peripherals to the core.

This chapter provides the following sections:

System Interrupt Controller

ADSP-2191 DSPs enhance the interrupt capabilities of the ADSP-219x core with an additional system interrupt controller. For information on core interrupt controller, see "Interrupts and Sequencing" on page 3-25.

Figure 6-1 illustrates how the system interrupt controller connects to the core.

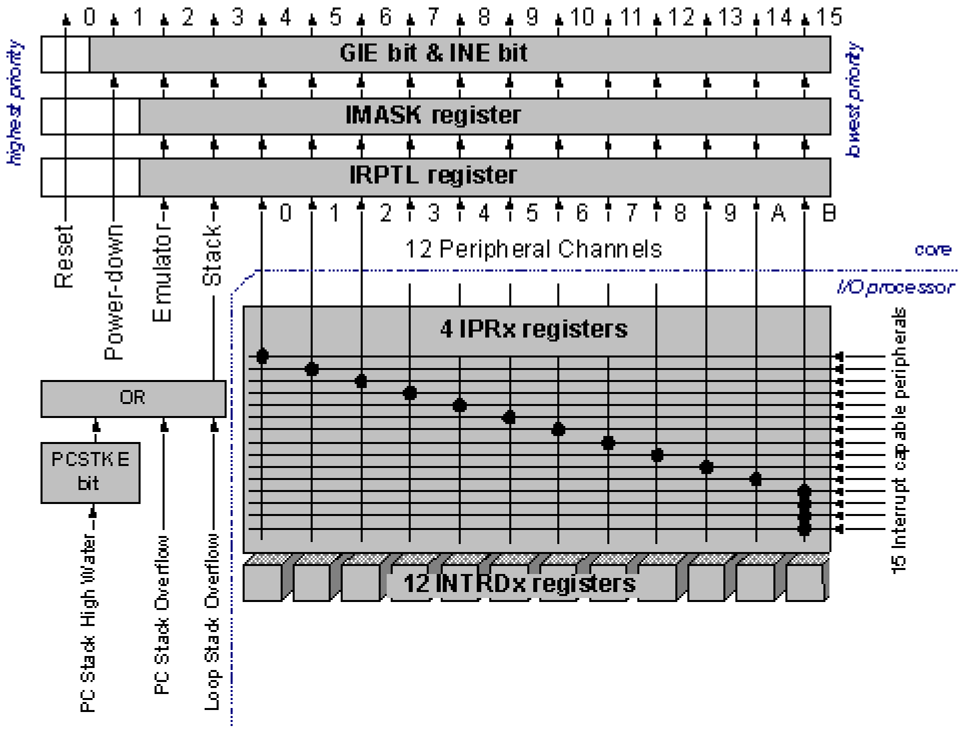
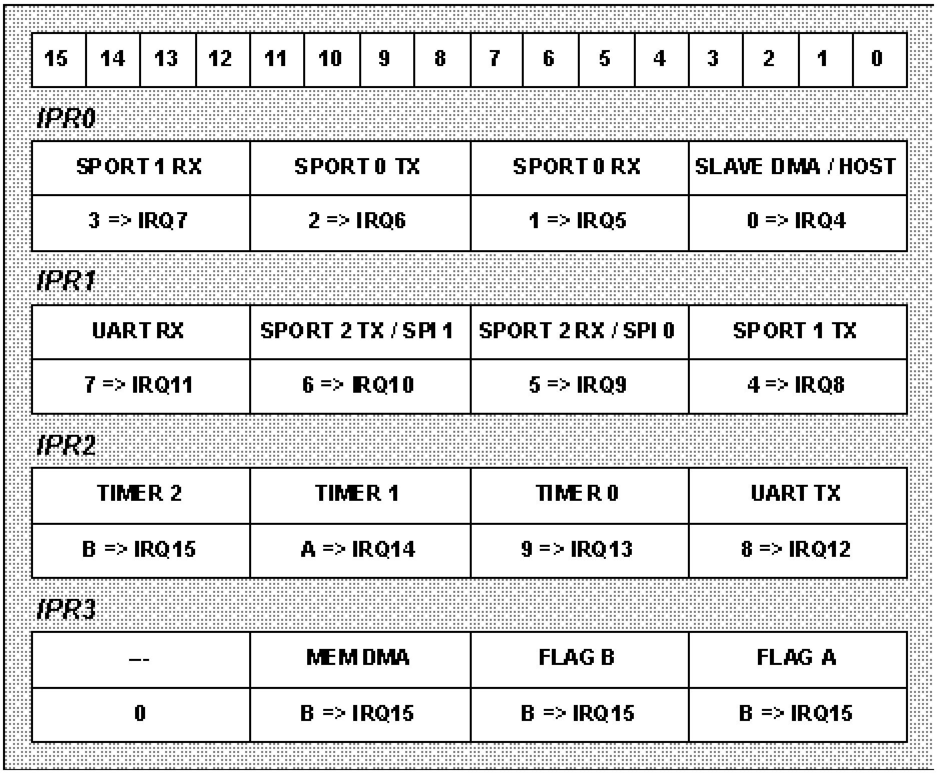


Figure 6-1. Interrupt Controller Overview

The ADSP-219x core provides 16 interrupt channels. The four channels with the highest priority are dedicated to special core purposes. The other 12 channels can be used for software interrupt schemes. More likely, the DSP peripherals utilize these channels.

The interrupt signals of the individual peripherals are not connected directly to the core's inputs; they are routed through a crossbar unit inside the I/O processor. The crossbar is controlled by the four Interrupt Priority (`IPRx`) registers. Each of the 15 peripheral interrupts owns four bits that map the interrupt source to one of the core channels. Since the core channels have fixed priorities, this crossbar enables free assignment of interrupt priorities.



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

IPR0

SPORT 1 RX	SPORT 0 TX	SPORT 0 RX	SLAVE DMA / HOST
3 => IRQ7	2 => IRQ6	1 => IRQ5	0 => IRQ4

IPR1

UART RX	SPORT 2 TX / SPI 1	SPORT 2 RX / SPI 0	SPORT 1 TX
7 => IRQ11	6 => IRQ10	5 => IRQ9	4 => IRQ8

IPR2

TIMER 2	TIMER 1	TIMER 0	UART TX
B => IRQ15	A => IRQ14	9 => IRQ13	8 => IRQ12

IPR3

--	MEM DMA	FLAG B	FLAG A
0	B => IRQ15	B => IRQ15	B => IRQ15

Figure 6-2. Default Mapping of IPRx Registers After Reset

Figure 6-2 shows the default mapping of the `IPRx` registers after a reset.

Multiple peripheral interrupts may share a single core interrupt channel. In this case, one interrupt routine services all the requests. The 12 Interrupt Source (`INTRDx`) registers can be used by the service routine to identify which interrupt sources are signaling. Refer to "System Interrupt Controller Registers" on page B-21 for more information.

Configuring System Interrupts

Most of the DSP peripherals have their own local interrupt enable and latch bits. Therefore, every enabled interrupt must be set up in the peripheral in the I/O processor and in the core.

Usually, the following steps are recommended:

1. Clear pending interrupt requests
2. Configure and enable interrupt in peripheral
3. Optionally, configure the DMA's interrupt behavior
4. Program the Interrupt Priority (`IPRx`) registers
5. Set the proper bits in the `IMASK` register to unmask core channels
6. Program the `ICNTL` register
7. Execute the `ENA INT;` instruction

Interrupt Setup Examples

The following brief examples illustrate how to set up interrupts.

The first example makes the Timer0 interrupt, which has an ID of 9, have a priority of `1`. This priority of `1` is bit 5 of the `IMASK` and `IRPTL` registers.

```
IOPG = Interrupt_Controller_Page;
AR = 0xBB1B;
IO(IPR2) = ar;          /* assign Timer 0 int priority to 1 */
AR = 0xBBBB;
IO(IPR0) = AR;          /* assign all other int priorities to 11 */
IO(IPR1) = AR;
IO(IPR3) = AR;
IMASK = 0x0020;         /* unmask core channel 5 */
```

The second example makes the Timer0 interrupt, which has an ID of 9, have a priority of `5`. This priority of `5` is bit 9 in the `IMASK` and `IRPTL` registers.

```
IOPG = Interrupt_Controller_Page;
AR = 0xBB5B;
IO(IPR2) = AR;
AR = 0xBBBB;
IO(IPR0) = AR;
IO(IPR1) = AR;
IO(IPR3) = AR;
IMASK = 0x0200;     /* unmask core channel 9 */
```

The final example sets up an SPI interrupt.

```
Program_SPI0_Interrupt:
IOPG = 0;
AR = IO(SYSCR);       /* Map Interrupt Vector Table to Page 0*/
AR = SETBIT 4 OF AR;
IO(SYSCR) = AR;

DIS INT;            /* Disable all interrupts */
IRPTL = 0x0;        /* Clear all interrupts */
ICNTL = 0x0;        /* Interrupt nesting disable */
IMASK = 0;          /* Mask all interrupts */

/* Set up Interrupt Priorities */
IOPG = Interrupt_Controller_Page;
AR = 0xBB1B;        /* Assign SPI0 priority of 1 */
IO(IPR1) = AR;
AR = 0xBBBB;     /* Assign the remainder with lowest priority */
IO(IPR0) = AR;
IO(IPR2) = AR;
IO(IPR3) = AR;
```

```
AY0 = IMASK;
AY1 = 0x0020;            /* Unmask SPI Interrupt */
AR = AY0 OR AY1;
IMASK = AR;

ENA INT;                /* Globally Enable Interrupts */
```

Servicing System Interrupts

Core interrupts and software interrupts are fully controlled by the Interrupt Latch (IRPTL) register. The individual interrupt latch bits are cleared automatically when the service routine executes the RTI instruction.

In the case of system interrupts, the service routine must clear the interrupt request explicitly, because there is no feedback path from the IRPTL registers back to the signaling peripheral. Otherwise, the peripheral would set the according bit in the IRPTL register again, as soon as the service routine terminates.

To prevent a subsequent event from being ignored, clear the peripheral's interrupt request at the very beginning of the service routine. Because of potential system latencies, the interrupt request must be cleared several cycles before the RTI instruction executes.

The following example illustrates a typical case of an interrupt routine that services a request from Timer 0, assuming Timer 0 is assigned to the core channel number 5.

```
.section / code IVint5;
ENA SR;                  /* enable secondary register file */
AY1 = iopg;              /* save I/O page register */
IOPG = Timer_Page;
AX0 = 0x0001;            /* Clear interrupt request */
IO(GSR0) = AX0;

                         /* do everything else here */
```

```
IOPG = AY1;                 /* restore I/O page register */
RTI;                        /* return from interrupt */
                            /* and disable secondary registers */
```

DMA Controller

The DSP's I/O processor manages Direct Memory Access (DMA) of DSP memory through the external, host, serial, SPI, and UART ports. Each DMA operation transfers an entire block of data. By managing DMA, the I/O processor lets programs move data as a background task while using the processor core for other DSP operations. The I/O processor's architecture, which appears in Figure 6-3 on page 6-9, supports a number of DMA operations. These operations include the following transfer types:

- Memory ↔ Memory or memory-mapped peripherals
- Memory ↔ Host processor
- Memory ↔ Serial port I/O
- Memory ↔ Serial Peripheral Interface (SPI) port I/O
- Memory ↔ UART port I/O

This chapter describes the I/O processor and how the I/O processor controls external, host, serial, SPI, and UART port DMA operations. For information on connecting external devices to these ports, see "External Port" on page 7-1, "Host Port" on page 8-1, "Serial Ports (SPORTs)" on page 9-1, "Serial Peripheral Interface (SPI) Ports" on page 10-1, or "UART Port" on page 11-1.

The ADSP-2191's I/O processor uses a distributed DMA control architecture. Each DMA channel on the chip has a controller to handle its transfers. Each of these channel controllers is similar, but each has some minor difference to accommodate the peripheral port being served by the channel.

The common features of all DMA channels are that they use a linked list of "descriptors" to define each DMA transfer, and the DMA transactions take place across an internal DMA bus. DMA-capable peripherals arbitrate for access to the DMA bus, so they can move data to and from memory.

Each channel's DMA controller moves 16-bit or 24-bit data without DSP-core processor intervention. When data is ready to be moved, the DMA channel requests the DMA bus and conducts the desired transaction.

To further minimize loading on the processor core, the I/O processor supports chained DMA operations. When using chained DMA, a program can set up a DMA transfer to automatically set up and start the next DMA transfer after the current one completes.

Figure 6-3 on page 6-9 shows the DSP's I/O processor, related ports, and buses. Software accesses the registers shown in this figure using an I/O memory read or write (`Io()`) instruction. The port, buffer, and DMA status registers configure the ports and show port status. The DMA descriptor registers configure and control DMA transfers.The data buffer registers hold data passing to and from each port. These data buffer registers include:

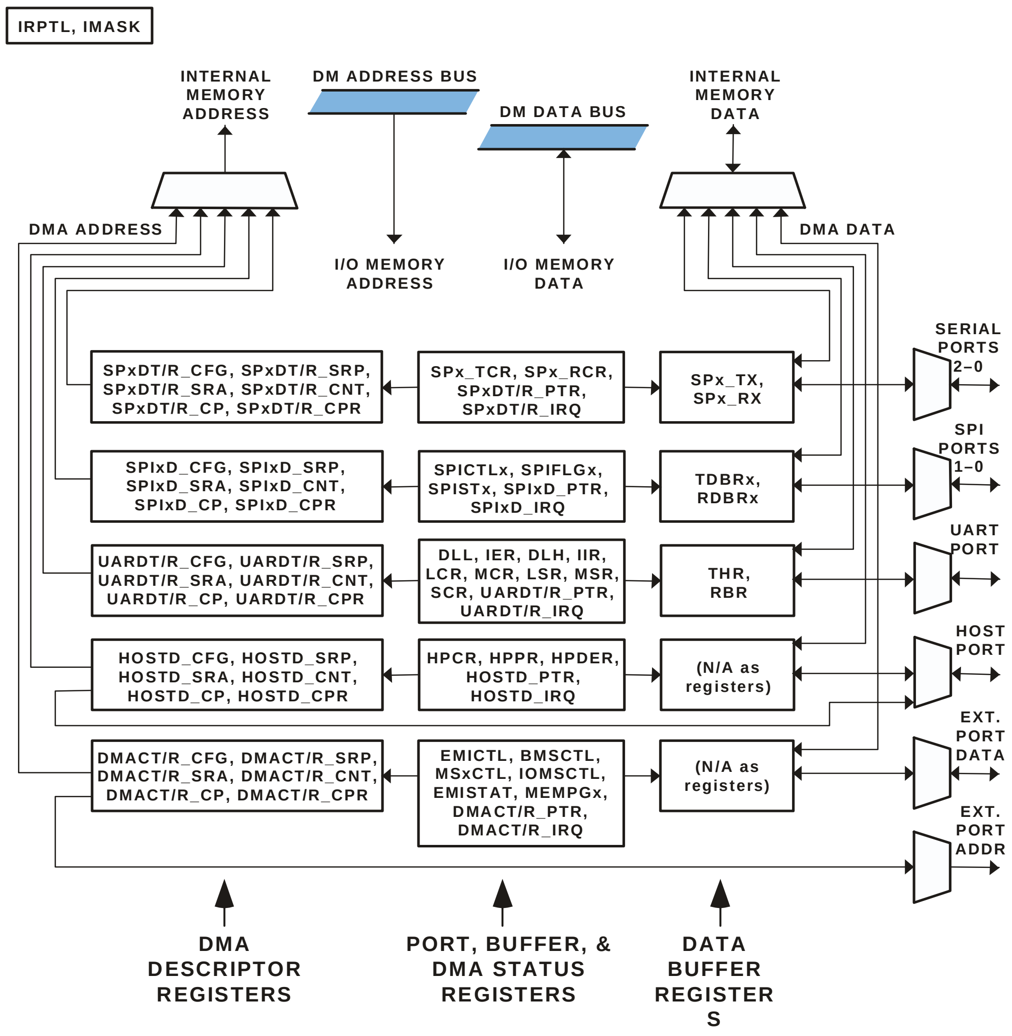

IRPTL, IMASK
INTERNAL MEMORY ADDRESS
DM ADDRESS BUS
DM DATA BUS
INTERNAL MEMORY DATA
DMA ADDRESS
DMA DATA
I/O MEMORY ADDRESS
I/O MEMORY DATA
SERIAL PORTS 2–0
SPI PORTS 1–0
UART PORT
HOST PORT
EXT. PORT DATA
EXT. PORT ADDR
SPxDT/R_CFG, SPxDT/R_SRP, SPxDT/R_SRA, SPxDT/R_CNT, SPxDT/R_CP, SPxDT/R_CPR
SPx_TCR, SPx_RCR, SPxDT/R_PTR, SPxDT/R_IRQ
SPx_TX, SPx_RX
SPIxD_CFG, SPIxD_SRP, SPIxD_SRA, SPIxD_CNT, SPIxD_CP, SPIxD_CPR
SPICTLx, SPIFLGx, SPISTx, SPIxD_PTR, SPIxD_IRQ
TDBRx, RDBRx
UARDT/R_CFG, UARDT/R_SRP, UARDT/R_SRA, UARDT/R_CNT, UARDT/R_CP, UARDT/R_CPR
DLL, IER, DLH, IIR, LCR, MCR, LSR, MSR, SCR, UARDT/R_PTR, UARDT/R_IRQ
THR, RBR
HOSTD_CFG, HOSTD_SRP, HOSTD_SRA, HOSTD_CNT, HOSTD_CP, HOSTD_CPR
HPCR, HPPR, HPDER, HOSTD_PTR, HOSTD_IRQ
(N/A as registers)
DMACT/R_CFG, DMACT/R_SRP, DMACT/R_SRA, DMACT/R_CNT, DMACT/R_CP, DMACT/R_CPR
EMICTL, BMSCTL, MSxCTL, IOMSCTL, EMISTAT, MEMPGx, DMACT/R_PTR, DMACT/R_IRQ
(N/A as registers)
DMA DESCRIPTOR REGISTERS
PORT, BUFFER, & DMA STATUS REGISTERS
DATA BUFFER REGISTER S

Figure 6-3. I/O Processor Block Diagram

- **Serial Port Receive Buffer registers** (SPx_RX). These receive buffers for the serial ports have FIFOs for receiving data when connected to another serial device. For I/O data, the FIFOs are two-levels deep. For DMA data, the FIFOs are eight-levels deep.
- **Serial Port Transmit Buffer registers** (SPx_TX). These transmit buffers for the serial ports have FIFOs for transmitting data when connected to another serial device. For I/O data, the FIFOs are two-level deep. For DMA data, the FIFOs are eight-levels deep.
- **SPI Port Receive Buffer registers** (RDBRx). These receive buffers for the SPI ports have four-level deep FIFOs for receiving data when connected to another SPI device.
- **SPI Port Transmit Buffer registers** (TDBRx). These transmit buffers for the SPI ports have four-level deep FIFOs for transmitting data when connected to another SPI device.
- **UART Port Receive Buffer registers** (RBR). These receive buffers for the UART ports have two-level deep FIFOs for receiving data when connected to another UART device.
- **UART Port Transmit Buffer registers** (THR). These transmit buffers for the UART ports have two-level deep FIFOs for transmitting data when connected to another UART device.

The DMA channels for the external port (memDMA) and host port each have four-level deep FIFOs, but these FIFOs are not visible as registers. The transmit and receive channels of each port share the port's FIFOs.

The Port, Buffer, and DMA Status Registers column in Figure 6-3 on page 6-9 shows the control registers for the ports and DMA channels. For more information on these registers, see the corresponding chapter of this text or "ADSP-2191 DSP I/O Registers" on page B-1.

The DMA Descriptor Registers column in Figure 6-3 on page 6-9 shows the descriptor registers for each DMA channel. These configure DMA channels and set up DMA transfers. For detailed information on descriptor registers, see "Setting Peripheral DMA Modes" on page 6-17.

Descriptor-Based DMA Transfers

DMA transfers on the ADSP-2191 can be descriptor-based or autobuffer-based—autobuffering only is available on SPORT, SPI, and UART DMA channels. Descriptor-based DMA has many more features, which requires more setup overhead, but descriptor-based DMA permits chaining varied DMA transfers together. Autobuffer-based DMA is much simpler, requiring minimal initial setup. Autobuffering also has the advantage of not requiring added setup overhead for repeated transfers.

Comparing DMA of the ADSP-2191 with the DMA of previous ADSP-218x DSPs, host port channel DMA is similar to IDMA, and memDMA channel DMA is similar to BDMA.

Descriptor-based DMA is the default method for describing DMA transfers on the ADSP-2191. Each descriptor contains all the information on a particular data transfer operation and contains the pointer to the next descriptor. When a transfer is complete, the DMA channel fetches the next descriptor's information then begins that transfer. The structure of a DMA descriptor appears in the Order of DMA Descriptor column of Table 6-2 on page 6-18 and consists of five register positions HEAD through HEAD+4

DMA descriptors either are *active*—have been loaded by the peripheral's DMA controller into registers on Pages 0–7 of internal I/O memory and are being used for an active DMA transfer—or are *inactive*—have not yet been loaded by a DMA controller.

Inactive DMA descriptors are stored in internal data memory (Page 0). For address information on descriptor registers, see "ADSP-2191 DSP I/O Registers" on page B-1. While descriptors are inactive, the DSP or host sets up descriptors as needed.

DMA descriptors become *active* as each DMA controller fetches its descriptor information from internal I/O memory before beginning a DMA transfer. The dynamic fetching of a descriptor is controlled by the DMA ownership (`DOWN`) bit in the descriptor. Before loading the descriptor from I/O memory, the DMA controller checks the `DOWN` bit to determine if the descriptor is configured and ready. If `DOWN` is set, the DMA controller loads the remaining words of the descriptor. If the descriptor is not ready then the DMA controller waits until the `DOWN` bit is set. Setting the descriptor ready (`DR`) bit triggers the DMA controller to load the descriptor from the descriptor registers. Then, the DMA controller uses the descriptor information to carry out the required DMA transfer.

The following steps illustrate the typical process for software setting up a DMA descriptor for descriptor-based DMA. Note that steps 2 and 4 only apply for standalone transfers or the first descriptor in a series of chained descriptors.

1. Software writes the descriptor's HEAD+1 (Start Page), HEAD+2 (Start Address), HEAD+3 (DMA Count), HEAD+4 (Next Descriptor Pointer), and HEAD (DMA Configuration) to consecutive locations in data memory.

   For this write, the descriptor's `DOWN` bit (in HEAD) must be set (=1), indicating that the DMA controller "owns" the descriptor. After completing the transfer, the DMA controller clears (=0) the `DOWN` bit, returning descriptor ownership to the DSP or host.

   If a standalone transfer, note that the HEAD+4 (Next Descriptor Pointer) pointer must point to a memory location containing the data 0x0—this pointer *should not* point to address 0x0.

2. Software writes the address of HEAD to the DMA channel's Next Descriptor Pointer register I/O memory.

   This step only is needed if this descriptor is a standalone transfer or the first in a chained series of transfers.

3. Software sets (=1) the DMA channel's descriptor ready (`DR`) bit in the channel's Descriptor Ready register in I/O memory, directing the channel's DMA controller to load the descriptor.

   The channel's DMA controller responds by loading the descriptor from data memory into the channel's DMA control registers in I/O memory.

4. Software sets (=1) the DMA channel's DMA enable (`DEN`) bit in the channel's DMA Configuration register in I/O memory.

   This final write to the descriptor is only needed if this descriptor is a standalone transfer or the first in a chained series of transfers.

After loading the descriptor and detecting that the DMA transfer is enabled, the channel's DMA controller sets to work on the data transfer. The DMA channel arbitrates for the internal DMA bus as required and (when it gets bus access) performs the transfer. On each peripheral clock cycle, the DMA channel updates the status of the DMA transfer in the channel's DMA status registers.

When the DMA transfer is completed (DMA count has decremented to zero), the DMA controller writes the descriptor's HEAD value (now with count of 0 and ownership of 0) to the descriptor's HEAD location in data memory to indicate the final status of the transfer. If enabled in the descriptor's configuration, the channel also generates a DMA transfer complete interrupt; for more information, see "Interrupts from DMA Transfers" on page 6-15. Next, the channel's DMA controller fetches the next descriptor HEAD from the location indicated by the channel's Next Descriptor Pointer register. If the location contains a descriptor HEAD and the channel is configured for chained DMA, the process repeats. If the location contains 0x0, the process stops, because this value disables the DMA channel's DEN bit.

It is important to note that each descriptor-based transfer requires the overhead of five additional reads and one write transaction to load the descriptor and start the transfer. This overhead is inefficient for very small transfer sizes. This overhead also occurs between chained transfers (loading the next descriptor) and creates a possibility for overflow situations.

Autobuffer-Based DMA Transfers

ADSP-2191 DSP DMA transfers can be autobuffer-based or descriptor-based—autobuffering not available on the memDMA DMA channels. Autobuffering has the same setup overhead for the first transfer as descriptor-based DMA. Unlike descriptor-based DMA, autobuffer does not require loading descriptors from internal data memory for each repeated transfer. The DMA setup occurs once, and the transfer (once started) iterates repeatedly without re-loading DMA descriptors.

The steps for using autobuffering and the response from the DMA controller in autobuffering are the same as in descriptor-based DMA, except that on completing the transfer the DMA controller re-uses the setup values instead of fetching the next descriptor. This effectively creates a circular buffer that continues to transfer data until disabled by clearing (=0) the DMA channel's DEN bit.

When autobuffering, some bits in the DMA Configuration register in I/O memory become read/write, instead of their read-only state when in descriptor-base DMA mode.

If enabled, the DMA controller generates interrupts at the halfway and completion points in the transfer. For more information, see "Interrupts from DMA Transfers" on page 6-15. Note that the corresponding bit in the IMASK register must be set to unmask the interrupt.

The following steps illustrate the typical process for software setting up a DMA descriptor for autobuffer-based DMA. Do not set the channel's `DEN` bit until the last step.

1. Software writes the descriptor's HEAD register in I/O memory, only setting (=1) the DMA channel's `DAUTO` bit.
2. Software writes the descriptor's HEAD+1, HEAD+2, and HEAD+3 registers in I/O memory.
3. Software writes the descriptor's HEAD register in I/O memory, configuring the DMA transfer and setting the `DEN` bit.

   This final write to the descriptor starts the autobuffering transfer.

Interrupts from DMA Transfers

The ADSP-2191's DMA channels can produce two types of interrupts: a completion interrupt and a port-specific DMA error interrupt.

DMA interrupt status is distributed because the DMA channels' operation is recorded in two ways. The status is recorded in the channel's DMA Configuration (`xxxx_CFG`) register and in the channel's DMA Interrupt Status (`xxxx_IRQ`) register when an interrupt occurs; these registers are in I/O memory.

The channel's `xxxx_IRQ` register is a sticky two-bit register that records that a DMA interrupt has occurred. These bits stay set until cleared (W1C) through a software write to them. This software write is required to clear the interrupt.

The channel's `xxxx_CFG` register records a more dynamic status of the DMA interrupts. Because DMA operation typically continues after an interrupt, the status available in the `xxxx_CFG` register must be used carefully. At the end of a transfer, the DMA controller writes the channel's `xxxx_CFG` register in I/O memory, then loads the next descriptor. If the transfer ends between the interrupt occurrence and the software polling the xxxx_CFG register, the software reads the status for the previous transfer as the status for the current transfer.

To avoid mismatched status, the software must conduct a full descriptor cleanup after most interrupts. This cleanup implies both checking the status of the current xxxx_CFG register and checking the status of recently completed descriptors in memory to determine the transfer with the error.

The channel's DMA controller generates a DMA complete interrupt at the end of a transfer. When a transfer completes, the DMA controller clears (=0) the DOWN bit in the descriptor's HEAD in data memory (returning descriptor ownership to the DSP or host) and sets (=1) the DS bit (indicating DMA status as complete).

The channel's DMA controller generates a port specific DMA error interrupt for errors such as receive overrun, framing errors, and others. For these port specific errors, the DMA controller logs the status in bits 11–9 of the channel's xxxx_CFG register. For more information on these bits, see "Host Port DMA Settings" on page 6-22, "SPI Port DMA Settings" on page 6-23, and "UART Port DMA Settings" on page 6-25.

These port specific bits are sticky and are only cleared at the start of the next transfer. These bits only can indicate that a port DMA error has occurred in the transfer, but cannot identify the exact word.

For information on enabling DMA interrupts, see "Setting Peripheral DMA Modes" on page 6-17.

Setting Peripheral DMA Modes

Each of the ADSP-2191's I/O ports has one or more DMA channels. The DMA controller setup and operation for each channel is almost identical. This section describes the settings that are common to all channels. For more information on settings that are unique to a particular DMA channel, see the following sections:

- "DMA Channels" on page 6-17
- "MemDMA DMA Settings" on page 6-21
- "Host Port DMA Settings" on page 6-22
- "Serial Port DMA Settings" on page 6-23
- "SPI Port DMA Settings" on page 6-23
- "UART Port DMA Settings" on page 6-25

DMA Channels

The ADSP-2191's DMA channels are listed in order of arbitration priority in Table 6-1 on page 6-17. This table also indicates the channel abbreviation that prefixes each channel's register names.

Table 6-1. DMA Channel Descriptions

DMA Channel Abbreviation	DMA Channel Description	DMA Channel Arbitration Priority
SP0DR	Serial Port (SPORT) 0 Receive	0
SP1DR	Serial Port (SPORT) 1 Receive	1
SP2DR	Serial Port (SPORT) 2 Receive	2
SP0DT	Serial Port (SPORT) 0 Transmit	3
SP1DT	Serial Port (SPORT) 1 Transmit	4

Table 6-1. DMA Channel Descriptions (Cont'd)

DMA Channel Abbreviation	DMA Channel Description	DMA Channel Arbitration Priority
SP2DT	Serial Port (SPORT) 2 Transmit	5
SPI0D	Serial Peripheral Interface (SPI) Port 0 (R/T)	6
SPI1D	Serial Peripheral Interface (SPI) Port 1 (R/T)	7
UARDR	UART Port Receive	8
UARDT	UART Port Transmit	9
HOSTD	Host Port (R/T)	10
DMACR	MemDMA Receive	11
DMACW	MemDMA Transmit	12

Each DMA channel listed in Table 6-1 on page 6-17 has the registers listed in Table 6-2 on page 6-18. The xxxx in the Table 6-2 on page 6-18 register names are place holders for the DMA channel abbreviations. The following registers control the operating mode of a peripheral's DMA controller.

Table 6-2. DMA Register Descriptions

DMA Register Name (in I/O Memory)	DMA Register Description	Order of DMA Descriptor (in Data Memory)
xxxx_PTR	Current Pointer. Contains the 16-bit address of the memory location that the DMA controller is reading (for transmit) or writing (for receive)	
xxxx_CFG	DMA Configuration. Contains the DMA configuration for the transfer (see bit descriptions on page 6-19)	HEAD
The xxxx in the register name corresponds to the DMA channels that are listed in Table 6-1 on page 6-17. The empty descriptor positions indicate registers that are not loaded from the DMA descriptor.		

Table 6-2. DMA Register Descriptions (Cont'd)

DMA Register Name (in I/O Memory)	DMA Register Description	Order of DMA Descriptor (in Data Memory)
xxxx_SRP	Start Page. Contains the Memory Space (MS) bit (bit 8, 0=memory, 1=boot) and transfer memory page (MP) bits (bits 7–0);	HEAD+1
xxxx_SRA	Start Address. Contains the 16-bit starting memory address of transfer	HEAD+2
xxxx_CNT	DMA Count. Contains the 16-bit number of words in the transfer	HEAD+3
xxxx_CP	Next Descriptor Pointer. Contains the 16-bit memory address of the Head of the next DMA descriptor	HEAD+4
xxxx_CPR	Descriptor Ready. Contains the Descriptor Ready (DR) bit (bit 0)	
xxxx_IRQ	DMA Interrupt Status. Contains the DMA Complete Interrupt Pending (DCOMI) bit (bit 0) and DMA Error Interrupt Pending (DERI) bit (bit 1); 1=pending interrupt, 0=no interrupt	

The xxxx in the register name corresponds to the DMA channels that are listed in Table 6-1 on page 6-17.
The empty descriptor positions indicate registers that are not loaded from the DMA descriptor.

Each DMA channel's xxxx_CFG register contains the following bits. Note that some bits are read-only in registers and only can be loaded when the DMA controller loads the xxxx_CFG register on descriptor load from data memory (see "Descriptor-Based DMA Transfers" on page 6-11). Also, a number of bits are read-only on channels where they are not supported (such as the DAUTO bit on the memDMA channel):

- **DMA Enable.** xxxx_CFG bit 0 (DEN). This bit directs the channel's DMA controller to start (if set, =1) or stop (if cleared, =0) the DMA transfer defined by the DMA descriptor. (read/write)

- **DMA Transfer Direction Select.** xxxx_CFG bit 1 (TRAN). This bit selects the transfer direction as memory write (if set, =1) or memory read (if cleared, =0). (read-only; applies on all I/O channels)

On the MemDMA channel, a memory write (transmit) uses the start address as the destination, and a memory read (receive) uses the start address as the source.

- **DMA Interrupt on Completion Enable.** `xxxx_CFG` bit 2 (`DCOME`). This bit enables (if set, =1) or disables (if cleared, =0) the channel's DMA complete interrupt. (read-only for descriptor-based DMAs)
- **DMA Data Type Select.** `xxxx_CFG` bit 3 (`DTYPE`). This bit—on parallel I/O channels—selects the data format as 24-bit (if set, =1) or 16-bit (if cleared, =0). (read-only; only applies on parallel I/O channels)
- **DMA Autobuffer/Descriptor Mode Select.** `xxxx_CFG` bit 4 (`DAUTO`). This bit—on channels that support autobuffer mode—selects autobuffer mode DMA (if set, =1) or descriptor-based DMA (if cleared, =0). (read-only; only applies on autobuffer mode channels)
- **DMA Buffer & Status Flush.** `xxxx_CFG` bit 7 (`FLSH`). Setting (writing 1) this bit flushes the channel's DMA buffer and clears (=0) the channel's `FS` and `FLSH` bits. This bit must be explicitly cleared (W1C); writing 0 to this bit has no effect. (read/write; only write when `DEN`=0)
- **DMA Interrupt on Error Enable.** `xxxx_CFG` bit 8 (`DERE`). This bit enables (if set, =1) or disables (if cleared, =0) the channel's DMA error interrupt. (read-only)
- **DMA FIFO Buffer Status.** `xxxx_CFG` bits 13-12 (`FS`). These bits indicate the status of the channel's buffer as: 00=empty, 01=partially full, 10=partially empty, or 11=full. (read-only)

- **DMA Completion Status.** `xxxx_CFG` bit 14 (`DS`). This bit indicates whether the DMA transfer completed successfully (=0) or with an error (=1). (read-only)
- **DMA Ownership Status.** `xxxx_CFG` bits 15 (`DOWN`). This bit indicates the current "owner" of the DMA descriptor as: 1=DMA controller or 0=DSP/host. (read-only)

Although some channels have preset directions for transmit or receive, the `TRAN` bit must be set or cleared appropriately to match the direction of the DMA transfer.

Some bus master settings can lock out DMA requests. For more information, see "Bus Master Settings" on page 7-7.

MemDMA DMA Settings

There are two MemDMA channels—one for transmit and one for receive. These channels handle memory-to-memory DMA transfers. The transmit channel provides internal memory to external memory transfers, and the receive channel provides external memory to internal memory transfers. These channels have the following DMA configuration differences from other DMA channels:

- These DMA channels support descriptor mode DMA (they do not support autobuffer mode), so this channel's `DAUTO` bit is ignored.
- Even though each of these DMA channels has a preset direction (transmit or receive), the channels' `TRAN` bits must be set or cleared appropriately.
- These DMA channels serve a parallel I/O port, so these channels' `DTYPE` bits are used.

For information on these channels' other settings, see Table 6-1 on page 6-17, Table 6-2 on page 6-18, and the xxxx_CFG register discussion on page 6-19. For information on using these DMA channels, see "Using MemDMA DMA" on page 6-27.

Host Port DMA Settings

There is one Host port DMA channel. It handles transmit or receive transfers between the Host port and memory. This channel has the following DMA configuration differences from other DMA channels:

- This DMA channel supports descriptor mode DMA and autobuffer mode, so this channel's DAUTO bit is used.
- This DMA channel serves a parallel I/O port, so this channel's DTYPE bits are used.
- The Host Port DMA Configuration (HOSTD_CFG) register has a bit that differs from the other channel's configuration registers:

  DMA Transfer Ready Status. HOSTD_CFG bit 9 (DRDY) This bit—only on the host port DMA channel—indicates that the host port DMA transfer is ready (if set, =1) or is not ready (if cleared, =0). (read/write)

For information on this channel's other settings, see Table 6-1 on page 6-17, Table 6-2 on page 6-18, and the xxxx_CFG register discussion on page 6-19. For information on using this DMA channels, see "Using Host Port DMA" on page 6-28.

Serial Port DMA Settings

There are six serial port channels—one per port for transmit and one per port for receive. The transmit channels provide memory to SPORT transfers, and the receive channels provide SPORT memory transfers. These channels have the following DMA configuration differences from other DMA channels:

- These DMA channels support descriptor mode DMA and autobuffer mode, so these channels' `DAUTO` bit is used.
- Even though each of these DMA channels has a preset direction (transmit or receive), the channels' `TRAN` bits must be set or cleared appropriately.
- These DMA channels serve a serial I/O port, so these channels' `DTYPE` bits are ignored.

For information on these channels' other settings, see Table 6-1 on page 6-17, Table 6-2 on page 6-18, and the `xxxx_CFG` register discussion on page 6-19. For information on using these DMA channels, see "Using Serial Port (SPORT) DMA" on page 6-30.

SPI Port DMA Settings

There are two Serial Peripheral Interface (SPI) port channels—one per port. These channels can be set to transmit or receive. A transmit channel provides memory to SPI port transfers, and a receive channel provides SPI port to memory transfers. These channels have the following DMA configuration differences from other DMA channels:

- These DMA channels support descriptor mode DMA and autobuffer mode, so these channels' `DAUTO` bits are used.
- These DMA channels serve a serial I/O port, so these channels' `DTYPE` bits are ignored.

- The SPI DMA Configuration (SPIxD_CFG) registers have bits that differ from the other channel's configuration registers:

  DMA SPI Receive Busy (Overflow Error) Status. SPIxD_CFG bit 9 (RBSY) This bit—only on an SPI port DMA channel with TRAN=1—indicates that the SPI port buffer has overflowed (if set, =1) or has not overflowed (if cleared, =0). (read-only)

  DMA SPI Transmit (Underflow) Error Status. SPIxD_CFG bit 10 (TXE) This bit—only on an SPI port DMA channel with TRAN=0—indicates that the SPI port buffer has underflowed (if set, =1) or has not underflowed (if cleared, =0). (read-only)

  DMA SPI Mode Fault (Multi-master Error) Status. SPIxD_CFG bit 11 (MODF) This bit indicates that another SPI master has aborted (if set, =1) or has not aborted (if cleared, =0) the current DMA transfer. (read-only)

For information on these channels other settings, see Table 6-1 on page 6-17, Table 6-2 on page 6-18, and the xxxx_CFG register discussion on page 6-19. For information on using these DMA channels, see "Using SPI Port DMA" on page 6-33.

UART Port DMA Settings

There are two UART port channels—one for transmit and one for receive. The transmit channel provides memory to UART transfers, and the receive channel provides UART to memory transfers. These channels have the following DMA configuration differences from other DMA channels:

- These DMA channels support descriptor mode DMA and autobuffer mode, so these channels' `DAUTO` bits are used.
- Even though each of these DMA channels has a preset direction (transmit or receive), the channel's `TRAN` bit must be set or cleared appropriately.
- These DMA channels serve a serial I/O port, so these channels' `DTYPE` bits are ignored.

- The UART port's DMA channels' configuration (UARD_CFG) register have bits that differ from the other channel's configuration registers:

  DMA UART Receive Overflow Error Status. UARD_CFG bit 9 (UAROE). This bit—only on an UART port DMA channel with TRAN=0—indicates that the receive buffer has (if set, =1) or has not (if cleared, =0) overflowed in the current DMA transfer. (read-only)

  DMA UART Receive Parity Error Status. UARD_CFG bit 10 (UARPE). This bit—only on an UART port DMA channel with TRAN=0—indicates that a parity error has (if set, =1) or has not (if cleared, =0) occurred in the current DMA transfer. (read-only)

  DMA UART Receive Framing Error Status. UARD_CFG bit 11 (UARFE). This bit—only on an UART port DMA channel with TRAN=0—indicates that a framing error has (if set, =1) or has not (if cleared, =0) occurred in the current DMA transfer. (read-only)

For information on these channels' other settings, see Table 6-1 on page 6-17, Table 6-2 on page 6-18, and the xxxx_CFG register discussion on page 6-19. For information on using these DMA channels, see "Using UART Port DMA" on page 6-39.

Working with Peripheral DMA Modes

With some minor differences, the DMA control for all ADSP-2191 DMA channels is identical. For a discussion of the DMA process and how to set it up, see "Descriptor-Based DMA Transfers" on page 6-11 and "Setting Peripheral DMA Modes" on page 6-17. This section provides detailed information on using each DMA-capable port.

This section provides the following topics:

Using MemDMA DMA

The MemDMA channels move 16- or 24-bit data between memory locations. These transfers include internal-to-external, external-to-internal, internal-to-internal, and external-to-external memory transfers. MemDMA can perform DMA transfers between internal, external, or boot memory spaces, but cannot DMA to or from I/O memory space.

There are two "halves" to the MemDMA (memory DMA) port: a dedicated "read" channel and a dedicated "write" channel. MemDMA first reads and stores data in an internal four-level deep FIFO, then (when the FIFO is full) MemDMA writes the FIFO's contents to the memory destination. When the remaining words of a transfer are less than four, the FIFO effectively becomes a single word buffer, which the MemDMA channels alternatively read and write.

Because the halves of MemDMA share their FIFO buffer, the read and write MemDMA channels must be configured for the same DMA transfer count. Failure to follow this restriction causes the MemDMA transfer to hang. When hung this way, the MemDMA channel releases the internal DMA bus, but does not complete the DMA transfer. Disabling the DMA and performing a buffer clear operation is required to clear this hang condition.

Using Host Port DMA

The host port DMA channel moves 16- or 24-bit data between DSP and host memory locations. This channel performs DMA transfers between the host port and internal, external, or boot memory spaces.

When host port DMA is enabled, the host should not send any address and should not initiate a memory or boot transfer with an address cycle while the host port is active. If the host does an address cycle, the host port ignores the address.

The data strobes sent by the host must be in line with the DMA direction parameter set in the DMA configuration register. If the direction is “0”, memory reads, the host must perform read cycles (read strobe); if the direction is “1”, the host must perform write cycles (write strobes). If the wrong strobe is used, it has no effect on the sequencing of the host port and DMA logic.

Data strobes clock the advancement of the packing/unpacking logic. The host port keeps track of the start and end of a packet from the start of a DMA transfer.

Either the ADSP-2191 or host may configure the DMA descriptor in DSP data memory and write the host port DMA control registers for starting, monitoring, and controlling the transfer.

During a host port DMA access, the DMA controller loads the DMA descriptor from data memory. The host is required to strobe out read data or strobe in write data while the host port automatically increments the DSP memory address.

In the case where both DSP core and host processor attempt to access the host port interface at the same time, there are some restrictions on host port DMA channel and host port interface operations. These restrictions include:

- If the external host attempts to latch a memory address (host port active) on the host port bus while the host port DMA is enabled, the address cycle is ignored and the address discarded.
- If a host port qualified data strobe is asserted, but does not correspond to the current DMA setting (assert write strobe while the DMA is enabled in read mode for example), the strobe is ignored.
- If the host makes a memory/boot access while DMA descriptors are changing, the host port acknowledge (HACK) takes longer to be asserted. To avoid this, use autobuffer-based DMA instead of descriptor-based DMA.
- If the host makes an I/O memory access while DMA is enabled, this access can be destructive in some cases. Destructive cases are: I/O read with the DMA in read mode, I/O write outside of a packet boundary of the host DMA data stream.

The DSP core should be held off from write-accessing the host port DMA controller and host port interface I/O space while it is in use by the host. To avoid a race condition, a high-level synchronization protocol should be employed if both DSP and external host agent are likely to use the host port DMA controller and host port interface at the same time. For this, system software can implement a DMA ownership bit using one of the semaphore registers. This technique lets the processors, host or DSP core determine if another process is already using the DMA controller.

Using Serial Port (SPORT) DMA

The SPORT DMA channels move data between the serial ports and memory locations. Although the SPORT DMA transfers to and from memory are always performed with 16-bit words, the serial ports can handle word sizes from 3 to 16 bits. No packing of smaller words into the 16-bit DMA transfer word are performed by the SPORT. Each SPORT has one channel for receiving data and one for transmitting data.

The SPORT DMA channels are assigned higher priority than all other DMA channels (e.g., higher than SPI ports, UART port, MemDMA, and host port channels), because the SPORTS have a relatively low service rate and are unable to hold off incoming data. Having higher priority causes the SPORT DMA transfers to be performed first when multiple DMA requests occur in the same cycle.

Descriptor-Based SPORT DMA

Once a DMA descriptor block has been properly generated, the SPORT DMA controller set up, and the DMA enabled (for details, see "Descriptor-Based DMA Transfers" on page 6-11), the SPORT loads the first descriptor block and begins to perform the first DMA transfer.

During the DMA transfer, data words received in the receive DMA FIFO are automatically transferred to the data buffer in internal memory. When the serial port is ready to transmit data, a word is automatically transferred from memory to the transmit DMA FIFO.

Note that the SPORT DMA controller extends the depth of the receive buffer when receive DMA is enabled from two words to eight words. This buffer extension lets the receive DMA controller correctly operate with long memory arbitration latencies in systems where many DMA peripherals are functioning at once. Similarly, the SPORT DMA controller extends the depth of the transmit buffer when transmit DMA is enabled from two words to eight words.

DMA operation continues until the entire transfer is complete—when the word count register reaches zero. When the word count register of an active DMA channel reaches zero, the DMA controller generates the DMA complete interrupt (if enabled in the DCOME bit of the descriptor).

Also on completion of the DMA, the DMA controller writes status and returns ownership of the descriptor of the just completed DMA operation to the DSP or host processor by writing the DMA configuration location of the descriptor. The DMA controller then continues to load the next descriptor in the linked list if the DMA configuration location of the next descriptor has the DOWN bit set and DEN bit set.

If a DMA overflow or underflow error occurs during a transfer, the DMA channel's controller sets the corresponding error status bit. Errors do not terminate the transfer. Error status is summarized in the SPORT Status Register (the TUVF and ROVF bits). Based on this information, software can make a decision to terminate the transfer by clearing (=0) the channel's DEN bit. If enabled with the DERE bit, this error also can generate an interrupt, setting the DERI bit.

If an error occurs, software should flush the channel's FIFO by setting (=1) the channel's FLSH bit. This bit should be set following any DMA termination due to an error condition. This bit has write-one-to-clear characteristic. This bit may also be used by a descriptor block load to initialize a DMA FIFO to a cleared condition prior to starting a DMA transfer. The DMA extended buffer not only is cleared, but the SPORT transmit double buffer and receive double buffers also are cleared.

Autobuffer-Based SPORT DMA

Autobuffer mode removes the overhead of the descriptor-based method when simply circular buffer type transfers are required. This mode provides compatibility with previous ADSP-218x SPORT autobuffering mode. For more information, see "Autobuffer-Based DMA Transfers" on page 6-14.

SPORT DMA Data Packed/Unpacked Enable

SPORT DMA supports packed and unpacked data. If in packed mode, the SPORT expects that the data contained by the DMA buffers corresponds only to the enabled SPORT channels. If an MCM frame contains ten enabled channels, the SPORT expects that the DMA buffer contains ten consecutive words for each of the frames. The DMA buffer size only can be as small as the number of the enabled channels, hence reducing the DMA traffic.

Note that one can not change the total number of the enabled channels without changing DMA buffer size. No mid-frame reconfiguration is allowed. DMA data packed mode is the only type of SPORT operation supported in non-DMA mode

If in unpacked mode, the DMA data is assumed to be unpacked. The DMA buffer is expected to have a word for each of the channels in the window (whether enabled or not). The DMA buffer size must be equal to the size of the window. If Channels 1 and 10 are enabled and the window size is 16, the DMA buffer size would have to be 16 words with the data to be transmitted/received placed at address 1 and 10 of the buffer. The content of the rest of the DMA buffer is ignored. The data is considered "unpacked" because the DMA buffer contains "extra" words. The purpose of this mode is to simplify the programming model of the SPORT MCM. For instance, this mode has no restrictions in terms of changing the number of enabled channels mid-frame (unlike in Data Packed mode above).

Software should set up the MCM Channel Select registers prior to enabling TX/RX DMA operation, because SPORT FIFO operation begins immediately after TX/RX DMA is enabled and depends on the values of the MCM Channel Select registers.

Using SPI Port DMA

The SPI has a single DMA controller, which supports either an SPI transmit channel or a receive channel, but not both simultaneously. When configured as a transmit channel, the received data is ignored. When configured as a receive channel, what is transmitted is irrelevant. A four-level deep FIFO is included to improve throughput of the DMA data.

ⓘ When changing the direction for SPI port DMA (from TX to RX or vice versa), the program must conclude the DMA in one direction, disable the channel, then start the next DMA in the other direction. TX and RX SPI DMA sequences cannot be chained with descriptors.

SPI DMA in Master Mode

When enabled as a master and the DMA controller is used to transmit or receive data, the SPI interface operates as follows:

1. The core writes to the `SPICTL` and `SPIBAUD` registers, enabling the device as a master and configuring the SPI system by selecting the appropriate word length, transfer format, baud rate, etc. The `TIMOD` field is configured to select "Transmit or Receive with DMA" mode.

2. The core selects the desired SPI slave(s) by setting one or more of the SPI flag select bits.

3. The core defines one or more DMA transfers by generating one or more DMA descriptors in data memory.

4. The core writes to the SPI DMA Configuration register, enabling the SPI DMA controller and configuring access direction.

5. The DMA controller writes the Head of the descriptor to the SPI DMA Next Descriptor register. To enable a receive operation, it is necessary to set the TRAN bit. In order to be able to set TRAN, it is first necessary to temporarily set the DAUTO bit. This is only necessary for master mode DMA operation.

6. If configured for transmit, as the DMA controller reads data from memory into the SPI DMA buffer, it initiates the transfer on the SPI port. If configured for receive, as the DMA controller reads data from SPI DMA buffer and writes to memory, it initiates the receive transfer.

7. The SPI then generates the programmed clock pulses on SPICLK and simultaneously shifts data out of MOSI and shifts data in from MISO. For transmit transfers, before starting to shift, the value in the DMA buffer is loaded into the shift register. For receive transfers, at the end of the transfer, the value in the shift register is loaded into the DMA buffer.

8. The SPI keeps sending or receiving words until the SPI DMA Word Count register transitions from 1 to 0.

For transmit DMA operations, if the DMA controller is unable to keep up with the transmit stream, perhaps because another DMA controller has been granted access to memory, the transmit port operates according to the state of the SZ bit. If SZ=1 and the DMA buffer is empty, the device repeatedly transmits 0s on the MOSI pin. If SZ=0 and the DMA buffer 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. The data in RDBR is not intended to be used, and the RXS and RBSY bits should be ignored. The RBSY overrun condition can not generate an error interrupt in this mode.

For receive DMA operations, if the DMA controller is unable to keep up with the receive data stream, the receive buffer operates according to the state of the GM bit. If GM=1 and the DMA buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the DMA buffer. If GM=0 and the DMA buffer is full, the incoming data is discarded, and the RDBR register is not updated. While performing a receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ=1, the device repeatedly transmits 0s on the MOSI pin. If SZ=0, it repeatedly transmits the contents of the TDBR register. The TXE underflow condition cannot generate an error interrupt in this mode.

Writes to the TDBR register during an active SPI transmit DMA operation should not occur because DMA data is overwritten. Writes to the TDBR register during an active SPI receive DMA operation are allowed. Reads from the RBDR register are allowed at any time. Interrupts are generated based on DMA events and are configured in the SPI DMA Configuration Word of the DMA descriptor.

For a transmit DMA operation to start, the transmit buffer must initially be empty (TXS=0). This is normally the case, but means that the TDBR register should not be used for any purpose other than SPI transfers. TDBR should not be used as a “scratch” register for temporary data storage. Writing to TDBR sets the TXS bit.

SPI DMA in Slave Mode

When enabled as a slave and the DMA controller is used to transmit or receive data, the start of a transfer is triggered by a transition of the SPISSx signal to the active-low state or by the first active edge of SCKx. The following steps illustrate the SPI receive DMA sequence in an SPI slave:

1. The core writes to the SPICTL 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 “Transmit or Receive with DMA” mode.

2. The core defines a DMA receive transfer by generating a receive DMA descriptor in data memory.

3. The core writes to the SPI DMA Configuration register, enabling the SPI DMA controller and configuring a receive access. The head of the descriptor is written to the SPI DMA Next Descriptor (SPIxD_CP) register.
4. Once the slave-select input is active, the slave starts receiving data on active SCKx edges.
5. Reception continues until SPI DMA Word Count (SPIxD_CNT) register transitions from 1 to 0.
6. The core could continue by queuing up the next DMA descriptor.

For receive DMA operations, if the DMA controller is unable to keep up with the receive data stream, the receive buffer operates according to the state of the GM bit. If GM=1 and the DMA buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the DMA buffer. If GM=0 and the DMA buffer is full, the incoming data is discarded. While performing receive DMA, the transmit buffer is assumed to be empty. If SZ=1, the device repeatedly transmits 0s on the MISO pin. If SZ=0, it repeatedly transmits the contents of the TDBR register. The following steps illustrate the SPI transmit DMA sequence in an SPI slave:

1. The core writes to the SPICTL register to define the mode of the serial link to be the same as the mode set-up in the SPI master. The TIMOD field is configured to select "Transmit or Receive with DMA" mode.
2. The core defines a DMA receive work unit by generating a receive DMA descriptor in data memory.
3. The core writes to the SPI DMA Configuration (SPIxD_CFG) register, enabling the SPI DMA controller and configuring a transmit operation. The head of the DMA descriptor is written to the SPI DMA Next Descriptor (SPIxD_CP) register.

4. Once the slave-select input is active, the slave starts transmitting data on active SCKx edges.

5. Transmission continues until the SPI DMA Word Count (SPIxD_CNT) register transitions to 0.

6. The core could continue by queuing up the next DMA descriptor.

For transmit DMA operations, if the DMA controller 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 buffer is empty, the device repeatedly transmits 0s on the MISO pin. If SZ=0 and the DMA buffer 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. The data in RDBR is not intended to be used, and the RXS and RBSY bits should be ignored. The RBSY overrun condition can not generate an error interrupt in this mode.

Writes to the TDBR register during an active SPI transmit DMA operation should not occur. Writes to the TDBR register during an active SPI receive DMA operation are allowed. Reads from the RBDR register are allowed at any time. Interrupts are generated based on DMA events and are configured in the SPI DMA Configuration Word of the DMA descriptor.

In order for a transmit DMA operation to execute properly, it is necessary for the transmit buffer to initially be empty (TXS=0). This is normally the case, but means that the TDBR register should not be used for any purpose other than SPI transfers. TDBR should not be used as a “scratch” register for temporary data storage. Writing to TDBR sets the TXS bit.

SPI DMA Errors

SPI DMA provides SPI-specific DMA error modes.

Mode-Fault Error (MODF). The MODF bit is set in the SPIST register when the SPISSx input pin of a device enabled as a master is driven low by another device in the system. This occurs in multiple master systems when another device is also trying to be the master. This contention between two drivers can potentially cause damage to the driving pins. To enable this feature, the PSSE bit in SPICTL must be set. As soon as this error is detected, the following actions take place:

1. The MSTR control bit in SPICTL is cleared, configuring the SPI interface as a slave.
2. The SPE control bit in SPICTL is cleared, disabling the SPI system.
3. The MODF status bit in SPIST is set.
4. An SPI interrupt is generated.

These conditions persist until the MODF bit is cleared, which is accomplished by a write-1 (W1C) software operation. Until the MODF bit is cleared, the SPI can not 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 SPISSx input pin should be checked to make sure the pin is high; otherwise, once SPE and MSTR are set, another mode-fault condition occurs again immediately.

As a result of SPE and MSTR being cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled, but the slave-select output pins revert to being controlled by the programmable flag registers. This change could lead to contention on the slave-select lines if these lines are still being driven by the DSP.

To assure that the slave-select output drivers are disabled once a MODF error occurs, configure the programmable flag registers appropriately. When enabling the MODF feature, configure all the PFx pins that serve as slave-selects as inputs. Accomplish this configuration by writing to the DIR register prior to configuring the SPI port. If configured this way, when the MODF error occurs, the slave-selects are automatically reconfigured as PFx pins, disabling the slave-select output drivers.

Transmission Error (TXE). This error bit is set in the SPI Port Status (SPISTx) register when all the conditions of transmission are met and there is no new data in SPI Transmit Buffer (TDBRx) register. In this case, what is transmitted depends on the state of the SZ bit in the SPICTL register. The TXE bit is cleared by a write-1 (W1C) software operation.

Reception Error (RBSY). The RBSY flag is set in the SPI Port Status (SPISTx) register when a new transfer has completed before the previous data could be read from the SPI Receive Buffer (RDBRx) register. This bit indicates that a new word was received while the receive buffer was full. The RBSY bit is cleared by a software write-1 (W1C) operation. The state of the GM bit in the SPICTL register determines whether the RDBRx register is updated with the newly received data.

Transmit Collision Error (TXCOL). The TXCOL flag is set in the SPI Port Status (SPISTx) register when a write to the TDBR register coincides with the load of the shift register. The write to TDBR can be direct or through DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted; in this case, the data which is in TDBRx may not match what was transmitted. Note that this bit is never set when the SPI is configured as a slave with CPHA=0; the collision error may occur, but it can not be detected. In any case, this error can easily be avoided by proper software control. The TXCOL bit is cleared by a software write-1 (W1C) operation.

Using UART Port DMA

The UART may be used in either a programmed I/O mode or in a DMA mode of operation. The I/O mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention because the DMA controller moves the data.

Separate receive and transmit DMA channels move data between the peripheral and memory in this mode. The processor is freed of the task of moving data and just sets up the appropriate transfers through DMA descriptors.

The UART has two interrupt outputs referred to as the receive (RX) and transmit (TX) interrupts. This nomenclature is somewhat misleading because in I/O mode all UART interrupts are grouped together as a single interrupt (the RX). Also note that in DMA mode, the break and modem status interrupts are not available.

In I/O mode, the RX interrupt is generated for all cases:

- RBR full
- Receive overrun error
- Receive parity error
- Receive framing error
- Break interrupt (RXSIN held low)
- Modem status interrupt (changes to DCD, RI, DSR, or CTS)
- THR empty

In DMA mode, the RX interrupt is generated when:

- RX work block complete
- Receive overrun error
- Receive parity error
- Receive framing error
- RX `DEN` bit error

In DMA mode, the TX interrupt is generated when:

- TX work block complete
- TX DEN bit error

The DMA completion interrupt is enabled by the DCOME bit of the DMA Configuration register. The error types of interrupts are enabled by the DERE bit of this register.

Boot Mode DMA Transfers

The ADSP-2191 uses DMA for external port booting only. This section provides a brief description of the DMA processes involved in booting. For a description of the booting process for all peripherals, see "Booting the Processor ("Boot Loading")" on page 14-16.

After reading the header for external port booting, the loader kernel polls the DMA ownership bit within the configuration word to determine completion of DMA. The loader kernel parses the header and sets up another DMA descriptor to load in the actual data following this header. While this DMA is in progress, the Boot ROM routine polls the DMA ownership bit to determine whether the DMA has completed or not.

This process repeats for all the blocks that need to be transferred. The last block to be read/initialized is the "final DM" block. This final block does not use a DMA descriptor, rather it is a direct core accesses. The interrupt service routine performs some housecleaning, transfers program control to location 0x0000, and begins running.

Code Example: Internal Memory DMA

This example demonstrates multiple internal to internal DMA transfers within the memory of the ADSP-2191. The example uses two methods to check for DMA completion:

- Interrupts—at the end of the first transfer a DMA completion interrupt is generated.
- Ownership bit of the configuration word—the second DMA polls this bit in memory to see if it is cleared. At the end of the transfer, the DMA engine writes a 0 (zero) to this bit in memory in order to transfer the control of DMA descriptor block to the DSP.

```
#include "def2191.h"
#define N 20

.section/dm data1;
.var SOURCE[N] =  0x1111, 0x2222, 0x3333, 0x4444, 0x5555,
                  0x6666, 0x7777, 0x8888, 0x9999, 0xaaaa,
                  0xbbbb, 0xcccc, 0xdddd, 0xeeee, 0xffff,
                  0xbade, 0xdeed, 0xfeed, 0xbead, 0xcafe;
.VAR DESTINATION2[N/2];

/*   Config     Start     Start    DMA    Next descriptor */
     word       page     address   count      pointer     */
     ------     -----    -------   -----  --------------- */

.var WR_DMA_WORD_CONFIG[5] =
     0x8007,    0x0000,   0x0000,    N,       0x0000;

.var RD_DMA_WORD_CONFIG[5] =
     0x8001,    0x0000,   0x0000,    N,       0x0000;

.var WR_DMA_WORD_CONFIG2[5] =
     0x0003,    0x0000,   0x0000,    N/2,    0x0000;

.var RD_DMA_WORD_CONFIG2[5]  =
     0x0001,    0x0000,   0x0000,    N/2,    0x0000;

.var end_dma = 0x0;
/*            to stop the DMA, point Next Address Pointer */
/*            to a buffer that contains ZERO */

.section/pm data2;
.var DESTINATION[N];
.var SOURCE2[N/2] =   0x1111, 0x2222, 0x3333, 0x4444, 0x5555,
                      0x6666, 0x7777, 0x8888, 0x9999, 0xaaaa;

.section/pm IVreset;
JUMP start;

/*  INTERRUPT SERVICE ROUTINE  */

.section/pm IVint4;
iopg = Memory_DMA_Controller_Page;
AX0 = 0x1;
IO(DMACW_IRQ) = AX0;
     /* writing a 1 to this register clears the interrupt */

     /* write the Configuration words for the 2nd transfer, */
     /* setting the Ownership and DMA enable bits */
AX0 = 0x8003;
AX1 = 0x8001;
DM(WR_DMA_WORD_CONFIG2) = AX0;
DM(RD_DMA_WORD_CONFIG2) = AX1;

AX0 = 0x1;
IO(DMACW_CPR) = AX0;
```

```
   /* Set the descriptor ready bit in both Write and Read chans
*/

IO(DMACR_CPR) = AX0;
   /* signal to the DMA engine that the down bit has been set */

RTI;     /* Return from interrupt */

    /* MAIN PROGRAM */

.section/pm  program;
start:

/* INTERRUPT PRIORITY CONFIGURATION */

IOPG = Interrupt_Controller_Page;
AX0 = 0xB0BB;
IO(IPR3) = AX0;
  /* assign DSP's interrupt priority 4 to Memory DMA port */
AX0 = 0xBBBB;
IO(IPR0) = AX0;
  /* set all other interrupts to the lowest priority */
IO(IPR1) = AX0;
IO(IPR2) = AX0;

ICNTL = 0X0;    /* Disable nesting */
IRPTL = 0X0;    /* Clear pending interrupts */
imask = 0x0010;    /* unmask interrupt 4 */

/*
Setting up the 1st set of descriptor blocks in memory
*/

/*
```

```
Write channel
*/

AX0 = WR_DMA_WORD_CONFIG2;
AX1 = DESTINATION;
DM(WR_DMA_WORD_CONFIG + 2) = AX1;
           /* write start address word (start of buffer) */
DM(WR_DMA_WORD_CONFIG + 4) = AX0;
           /* write next descriptor pointer word */

/*
Read channel
*/

AX1 = SOURCE;
AR = RD_DMA_WORD_CONFIG2;

DM(RD_DMA_WORD_CONFIG + 2) = AX1;
    /* write start address word (start of buffer) */

DM(RD_DMA_WORD_CONFIG + 4) = AR;
    /* write next descriptor pointer word */

/*
Set up the 2nd set of descriptor blocks in memory
*/

/*
Write channel
*/

AX0 = END_DMA;
AX1 = DESTINATION2;
DM(WR_DMA_WORD_CONFIG2 + 2) = AX1;     /* start address word */
DM(WR_DMA_WORD_CONFIG2 + 4) = ax0;  /* nxt descriptor ptr word*/

/*
Read channel
*/

AX1 = SOURCE2;
DM(RD_DMA_WORD_CONFIG2+2) = AX1;      /* startaddressword */
DM(RD_DMA_WORD_CONFIG2 + 4) = AX0;   /* next desc ptr word */

/*
Write to the DMA engine
*/

/* The following IO writes are necessary to kick off the DMA
engine for the first transfer. Subsequent chained DMA transfers
will only need to have the Ownership and DMA Enable bits set in
their respective configuration words in memory. Note that for
subsequent transfers if the ownership bit is not set, the
Descriptor Ready bits will need to be set again once the owner-
ship bit is set. */

IOPG = Memory_DMA_Controller_Page;
AX0 = WR_DMA_WORD_CONFIG;
IO(DMACW_CP) = AX0;
/* Load the address of the First Write Channel work unit */
AX1 = RD_DMA_WORD_CONFIG;
IO(DMACR_CP) = AX1;
   /* Load the address of the Read Channel work unit */

AX0 = 0x1;
IO(DMACW_CPR) = AX0;
   /* Set the descriptor ready bit in both Write and Read chans
*/
```

```
IO(DMACR_CPR) = AX0;
IO(DMACW_CFG) = AX0;

/* enable DMA in both channels, this enable plus the setting of
the descriptor ready bits will cause the DMA engine to fetch the
descriptor words from memory to its space in IO and begin the
transfer */

IO(DMACR_CFG) = AX0;

ENA INT;      /* enable global interrupts */
IDLE;
   /* wait for the DMA interrupt, which will be generated */
   /* once the 1st transfer completes */

/* loop here to check bit 15 (ownership bit) of the config regis-
ter in DM to see if DMA completed, On completion the DMA engine
will write a 0 to this bit */

do test_ownership until forever;
   AR = DM(WR_DMA_WORD_CONFIG2);
   AR = TSTBIT 15 OF AR;
test_ownership: if EQ jump dma_done;
   /* the explicit jump ends the infinite loop */

dma_done:
POP LOOP;
/* this instruction is necessary to recover the loop stack after
exiting an infinite loop if the label DMA_DONE is not the next
sequential instruction, after popping the loop stack another jump
to the next instruction after the loop may be needed.*/

NOP;
IDLE;
```

7 EXTERNAL PORT

This chapter provides the following sections:

Overview

The DSP's external port extends the DSP's address and data buses off-chip. Using these buses and external control lines, systems can interface the DSP with external memory or memory-mapped peripherals. This chapter describes configuring, connecting, and timing accesses to external memory or memory-mapped peripherals. For information describing the DSP's memory and how to use it, see "Memory" on page 5-1.

The external port connections appear in Figure 7-1 on page 7-2. The main sections of this chapter describe how to use the interfaces that are available through the external port.

There is a 4:1 conflict resolution ratio at the external port interface (three internal buses to one external bus), a 2:1 clock ratio between the DSP's internal clock and the peripheral clock (when HCLK=½ CCLK), and a packing delay of one cycle per word to

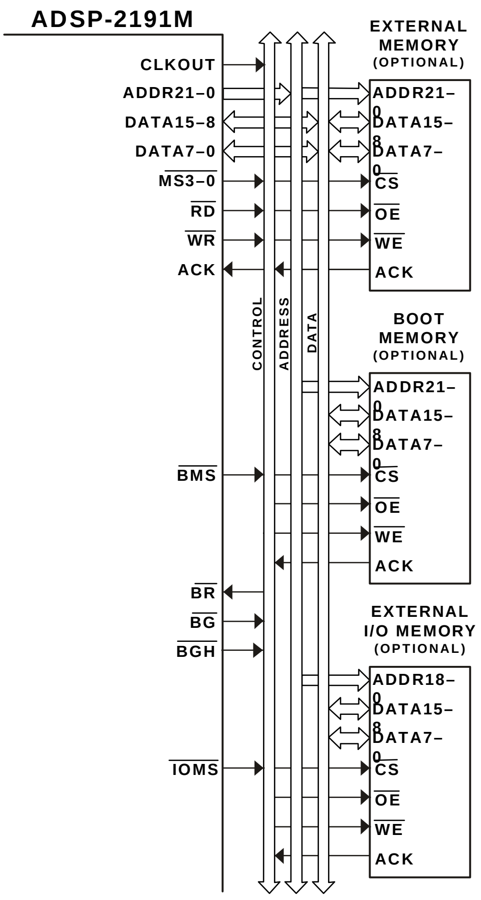


Figure 7-1. ADSP-2191 System—External Port Interfaces

unpack instructions. Systems that fetch instructions or data through the external port must tolerate latency on these accesses. For more information, see “Memory Interface Timing” on page 7-24.

Setting External Port Modes

The E_STAT, EMICTL, MSxCTL, BMSCTL, IOMSCTL, and MEMPGx registers control the operating mode of the DSP's memory. The settings for these modes are covered in the following sections:

- "Memory Bank and Memory Space Settings" on page 7-3
- "External Bus Settings" on page 7-5
- "Bus Master Settings" on page 7-7
- "Boot Memory Space Settings" on page 7-7

Memory Bank and Memory Space Settings

Each bank of external memory has a configurable setting for read waitstate count, write waitstate count, waitstate mode select, clock divider, and write hold cycle. Boot memory space and I/O memory space also have these settings. These features come from the following bits in the MSxCTL, BMSCTL, and IOMSCTL registers:

Read Waitstate Count. MSxCTL, BMSCTL, IOMSCTL bits 2-0 (E_RWC)
Write Waitstate Count. MSxCTL, BMSCTL, IOMSCTL bits 5-3 (E_WWC). These bits direct the DSP to apply 0 to 7 waitstates (EMICLK clock cycles), before completing the read or write access to the corresponding memory bank or memory space.

- **Waitstate Mode Select.** MSxCTL, BMSCTL, IOMSCTL bits 7-6 (E_WMS). These bits direct the DSP to use the following waitstate mode for the corresponding memory bank or memory space: external ACK only (if 00), internal waitstates only (if 01), both ACK and waitstates (if 10), either ACK or waitstates (if 11).

- **Clock Divider Select.** MSxCTL, BMSCTL, IOMSCTL bits 10-8 (E_CDS). These bits set the memory bank or space clock rate (EMICLK) at a ratio of the peripheral clock rate (HCLK) for accesses to the corresponding memory bank or memory space. The possible EMICLK:HCLK ratios are as follows: 1:1 (if 000), 1:2 (if 001), 1:4 (if 010), 1:8 (if 011), 1:16 (if 100), or 1:32 (if 101)

- **Write Hold Enable.** MSxCTL, BMSCTL, IOMSCTL bit 11 (E_WHE). This bit directs (if 1) the DSP to extend the write data hold time by one cycle following de-asserting of the $\overline{\text{WR}}$ strobe for the corresponding memory bank or memory space, providing more data hold time for slow devices. When disabled (if 0), the write data hold time is not extended.

The size of each bank of external memory is configurable. As shown in the ADSP-2191 memory map in Figure 5-4 on page 5-10, the default settings for bank size place 64 memory pages on each bank. The configurable number of pages per bank is set in the following registers/bits:

- **Bank 0 Lower Page Boundary.** MEMPG10 bits 7-0 (E_MS0_PG)
  Bank 1 Lower Page Boundary. MEMPG10 bits 15-8 (E_MS1_PG)
  Bank 2 Lower Page Boundary. MEMPG32 bits 7-0 (E_MS2_PG)
  Bank 3 Lower Page Boundary. MEMPG32 bits 15-8 (E_MS3_PG).

These bits select external memory bank sizes by selecting the starting page boundary for each memory bank. Each register holds the 8-bit page number of the lowest page on the bank.

External Bus Settings

The external port configuration includes settings for $\overline{\text{RD}}$/$\overline{\text{WR}}$ strobe polarity, external memory format, and external bus master access. The features come from the following bits in the EMICTL and E_STAT registers:

- **External Bus Width Select.** EMICTL bit 3 (E_BWS) selects the bus width for the external bus as 16 bits (if 1) or 8 bits (if 0). The external port bases packing operations on the data format selection and external bus width. This bus width applies to external memory space, boot memory space, and external I/O memory space.

- **Write Strobe Sense Logic Select.** EMICTL bit 4 (E_WLS)
  Read Strobe Sense Logic Select. EMICTL bit 5 (E_RLS)
  These bits direct the DSP to use active low (negative logic, if 1) or active high (positive logic, if 0) for the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ pins for accesses to external memory.

- **PM and DM Data Format Select.** E_STAT bit 3 (E_DFS) selects whether user PM and DM data requests from the core are treated as 24 bit or 16 bit when they are forwarded to the external interface for external memory transfers. The E_DFS bit effectively normalizes the word size and allows programs to use the same program address for accessing data regardless of whether it is in PM (24 bit) or DM (16 bit). The external interface packs the 16 or 24 bit data in external memory, depending on whether it is configured for 8 or 16 bit external memories. Instruction fetches are not affected by the E_DFS bit.

- **Access Split Enable.** EMICTL bit 6 (E_ASE) enables (if 1) splitting DMA transfers to or from external memory. If split is enabled, other DMA capable peripherals (for example, from or to SPORT, SPI, UART, or host) can perform DMA of internal memory while the external port is waiting to read or write DMA data in external memory. When disabled (if 0), other peripherals must wait for external port DMA transfers to complete (releasing its hold on DMA mastership), before getting access to internal memory for DMA.

- **CMS Output Enable.** `MSxCTL`, `BMSCTL`, `IOMSCTL` bit 15 (`E_COE`) enables (if 1) ORing of the corresponding memory bank's or memory space's select line with other (also enabled) selects, producing a composite memory select output. When disabled (if 0), the memory bank's or memory space's select line is not used to generate a $\overline{\texttt{CMS}}$ output.

The `E_COE` bit is a reserved bit on the ADSP-2191 (144-lead LQFP or mini-BGA packages), because the $\overline{\texttt{CMS}}$ pin is not available on this DSP.

Bus Master Settings

The external port permits external processors to gain control of the external bus using the $\overline{\text{BR}}$, $\overline{\text{BG}}$, and $\overline{\text{BGH}}$ pins. The configurable features for these pins come from the following bits in the `EMICTL` register:

- **Bus Lock.** `EMICTL` bit 0 (`E_BL`) locks out (if 1) response to external bus request ($\overline{\text{BR}}$) signals, locking the DSP as bus master. When disabled (if 0), the DSP responds to bus requests. This bit also locks out bus requests for DMA.

- **External Bus and DMA Request Holdoff Enable.** `EMICTL` bit 1 (`E_BHE`) holds off (if 1) response to external bus request ($\overline{\text{BR}}$) signals and DMA requests for 16 I/O clock cycles, delaying loss of bus mastership. When disabled (if 0), the DSP responds to bus requests without delay.

- **Access Control Registers Lock.** `EMICTL` bit 2 (`E_CRL`) locks out (if 1) write access to the `MSxCTL`, `BMSCTL`, and `IOMSCTL` registers, making their `E_RWC`, `E_WWC`, `E_WMS`, `E_CDS`, `E_WHE`, and `E_COE` settings read only. When disabled (if 0), the DSP can read or write the `MSx-CTL`, `BMSCTL`, and `IOMSCTL` registers.

Boot Memory Space Settings

The external port permits accessing boot memory space at runtime (after the DSP boots). When any of these modes are enabled, the DSP uses the $\overline{\text{BMS}}$ pin (instead of the $\overline{\text{MSx}}$ pins) for off-chip memory accesses, selecting boot memory space (instead of an external memory bank). The configurable features for boot memory format come from the following bits in the `E_STAT` register:

- **PM Instruction from Boot Space Enable.** `E_STAT` bit 0 (`E_PI_BE`) enables (if 1) access to boot memory space with the $\overline{\mathtt{BMS}}$ select line for fetching instructions or disables (if 0) boot memory space access. If disabled, the DSP applies normal usage of $\overline{\mathtt{MSx}}$ chip select lines for fetching instruction from external memory.
- **PM Data from Boot Space Enable.** `E_STAT` bit 1 (`E_PD_BE`) enables (if 1) access to boot memory space with the $\overline{\mathtt{BMS}}$ select line for accessing data over the PM bus or disables (if 0) boot memory space access. If disabled, the DSP applies normal usage of $\overline{\mathtt{MSx}}$ chip select lines for accessing data over the PM bus from external memory.
- **DM Data from Boot Space Enable.** `E_STAT` bit 2 (`E_DD_BE`) enables (if 1) access to boot memory space with the $\overline{\mathtt{BMS}}$ select line for accessing data over the DM bus or disables (if 0) boot memory space access. If disabled, the DSP applies normal usage of $\overline{\mathtt{MSx}}$ chip select lines for accessing data over the DM bus from external memory.

Working with External Port Modes

The external port provides many operating modes for using the DSP's external memory space, boot memory space, and I/O memory space. Techniques for using these modes are described in the following sections.

- "Using Memory Bank/Space Waitstates Modes" on page 7-9
- "Using Memory Bank/Space Clock Modes" on page 7-10
- "Using External Memory Banks and Pages" on page 7-11
- "Using Memory Access Status" on page 7-12
- "Using Bus Master Modes" on page 7-13
- "Using Boot Memory Space" on page 7-14

Using Memory Bank/Space Waitstates Modes

The DSP has a number of modes for accessing external memory space. The External Waitstate Mode Select (E_WMS) fields in the MSxCTL, BMSCTL, and IOMSCTL registers select how the DSP uses waitstates and the acknowledge (ACK) pin to access each external memory bank, boot memory, and I/O memory. The waitstate modes appear in Table 7-1 on page 7-10.

The DSP applies waitstates to each external memory access depending on the bank's and/or spaces's external waitstate mode (E_WMS). The External Read/Write Waitstates Count (E_R/WWC) fields in the MSxCTL, BMSCTL, and IOMSCTL registers set the number of waitstates for each bank and/or space as 000 = 0 waitstates to 111 = 7 waitstates.

For additional hold time on write data, systems can enable the Write Hold Enable (E_WHE) bit. Enabling E_WHE causes the DSP to leave the address and data unchanged for one additional cycle after the write strobe is

Table 7-1. External Memory Interface Waitstate Modes

E_WMS	External Memory Interface Waitstate Mode
00	ACK mode—DSP $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKOUT's edge—accesses require external acknowledge (ACK), allowing a de-asserted ACK to extend the access time. Note that there are two waitstates (at minimum) when using ACK mode.
01	Wait mode—DSP $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKOUT's edge—reads use the waitstate count setting from E_RWC (for reads) and writes use the waitstate count setting from E_WWC (for writes).
10	Both mode—DSP $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKOUT's edge—reads use the waitstate count setting from E_RWC (for reads) and writes use the waitstate count setting from E_WWC (for writes) and require external acknowledge (ACK), allowing both the waitstate count and a de-asserted ACK to extend the access time.
11	Either mode—DSP $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKOUT's edge—reads use the waitstate count setting from E_RWC (for reads) and E_WWC (for writes) or respond to external acknowledge (ACK), allowing either completion of the wait-state count or a de-asserted ACK to limit the access time.

de-asserted. This hold cycle provides additional address and data hold times for slow devices. For more information, see the E_WHE description on page 7-4.

The DSP applies hold time cycles regardless of the waitstate mode (E_WMS). For example, the Both mode (ACK plus waitstate mode) also could have an associated hold cycle.

Using Memory Bank/Space Clock Modes

The DSP provides additional clock ratio selections for each external memory bank, boot memory space, and external I/O memory space. These clock ratios let system designers accommodate access to slow devices without slowing the DSP core or other memory banks/spaces. Both address setup and strobe delay may be controlled by adjusting EMICLK. The clock ratio selections appear in Table 7-2 on page 7-11.

Table 7-2. External Memory Interface Clock Ratio Selections

E_CDS	Clock Divider Select Ratio (HCLK-to-EMICLK for Bank/Space)
000	1:1
001	1:2
010	1:4
011	1:8
100	1:16
101	1:32

Using External Memory Banks and Pages

At reset, the DSP's external memory space is configured with four banks of memory, each with 63 or 64 pages. After reset, systems should program the correct lower page boundary into each bank's `E_MSx_PG` bits, unless the default settings are appropriate for the system. Mapping peripherals into different banks lets systems accommodate I/O devices with different timing requirements, because each bank has an associated waitstate mode and clock mode setting. For more information, see "Using Memory Bank/Space Waitstates Modes" on page 7-9 and "Using Memory Bank/Space Clock Modes" on page 7-10.

As shown in Figure 5-4 on page 5-10, Bank 0 starts at address 0x1,0000 in external memory, and the Banks 1, 2, and 3 follow. Whenever the DSP generates an address that is located within one of the four banks, the DSP asserts the corresponding memory select line ($\overline{\texttt{MS3-0}}$).

Using Memory Access Status

The E_STAT and EP_STAT registers indicate the status of external port accesses to external memory. The following bits in the E_STAT and EP_STAT registers indicate memory access status:

- **External Write Pending Flag.** E_STAT bit 8 (E_WPF) is a read-only bit that indicates whether a write is pending (if 1) or no write is pending (if 0) on the external port.
- **External Bus Busy.** EP_STAT bits 1–0 (E_BSY) are read-only bits that indicate the external bus status as: 00 = not busy, 01 = off-chip master, 10 = on-chip master, or 11 = reserved.
- **External Last Master ID.** EP_STAT bits 6-2 (E_MID) are read-only bits that indicate the ID code for current or last master of the external port interface. A list of these ID codes appears in Table B-14 on page B-107.
- **External Word Packer Status.** EP_STAT bits 8-7 (E_WPS) are read-only bits that indicate the packing status for the external port interface as the packer contains: no bytes (empty if 00), one byte (if 01), two bytes (if 10), or three bytes (if 11).

Because the external memory interface does not hold up the DSP core while waiting for a write complete acknowledge, it's important for systems to check the write pending flag when using slow external memories. For more information, see "Memory Interface Timing" on page 7-24.

Using Bus Master Modes

An ADSP-2191 DSP can relinquish control of its data and address buses to an external device. The external device requests the bus by asserting (low) the bus request ($\overline{\texttt{BR}}$) pin. $\overline{\texttt{BR}}$ is an asynchronous input. If the ADSP-2191 is not performing an external access, it responds to the active $\overline{\texttt{BR}}$ input in the following processor cycle by:

1. Three-stating the data and address buses and the $\overline{\texttt{MSx}}$, $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$ pins
2. Asserting the bus grant ($\overline{\texttt{BG}}$) signal
3. Continuing program execution (until the DSP core requires an external memory access)

In systems that make the DSP a bus slave (active $\overline{\texttt{BR}}$ input), 10 kW pull-up resisters should be placed on the DSP's $\overline{\texttt{MSx}}$, $\overline{\texttt{BMS}}$, $\overline{\texttt{IOMS}}$, $\overline{\texttt{RD}}$, and $\overline{\texttt{WR}}$ pins.

The ADSP-2191 continues to execute instructions from its internal memory while the external bus is granted. The DSP does not halt program execution, until it encounters an instruction that requires an external access. An external access may be either an external memory, external I/O memory, or boot memory access.

Even when the ADSP-2191 halts because the DSP core is held off, the DSP's internal state is not affected by granting the bus. The other peripheral (host port, serial ports, SPI ports, and UART port) remain active during a bus grant, even when DSP core halts.

If the ADSP-2191 is performing an external access when the $\overline{\texttt{BR}}$ signal is asserted, the DSP does not grant the buses until the cycle after the access completes. The entire instruction does not need to be completed when the bus is granted. If a single instruction requires two external accesses, the bus is granted between the two accesses. The second access is performed after $\overline{\texttt{BR}}$ is removed.

When the $\overline{\mathtt{BR}}$ input is released, the ADSP-2191 releases the $\overline{\mathtt{BG}}$ signal, re-enables the output drivers and continues program execution from the point where it stopped. $\overline{\mathtt{BG}}$ is always deasserted in the same cycle that the removal of $\overline{\mathtt{BR}}$ is recognized. Refer to the *ADSP-2191 DSP Microcomputer Data Sheet* for exact timing relationships.

The bus request feature operates at all times, including when the processor is booting and when $\overline{\mathtt{RESET}}$ is active. During $\overline{\mathtt{RESET}}$, $\overline{\mathtt{BG}}$ is asserted in the same cycle that $\overline{\mathtt{BR}}$ is recognized. During booting, the bus is granted after completion of loading of the current byte (including any waitstates). Using bus request during booting is one way to bring the booting operation under control of a host computer.

The ADSP-2191 DSPs also have a Bus Grant Hung ($\overline{\mathtt{BGH}}$) pin, which lets them operate in a multiprocessor system with a minimum number of wasted cycles. The $\overline{\mathtt{BGH}}$ pin asserts when the ADSP-2191 is ready to perform an external memory access but is stopped because the external bus is granted to another device. The other device can release the bus by de-asserting bus request. Once the bus is released, the ADSP-2191 deasserts $\overline{\mathtt{BG}}$ and $\overline{\mathtt{BGH}}$ and executes the external access.

Using Boot Memory Space

As shown in Figure 7-1 on page 7-2, the DSP supports an external boot EPROM mapped to external memory and selected with the $\overline{\mathtt{BMS}}$ pin. The boot EPROM provides one of the methods for automatically loading a program into the internal memory of the DSP after power-up or after a software reset. This process is called booting. For information on boot options and the booting process, see "Boot Mode DMA Transfers" on page 6-41.

Boot memory space also is available at runtime, after booting. Depending on the size of the connected EPROM, the EPROM content is aliased several times in the memory map. Access the EPROM by using the addresses starting from `0x80 0000`. For information on this run-time access, see

"Reading from Boot Memory" on page 7-15 and "Writing to Boot Memory" on page 7-15. For a programming example of this access, see "Code Example: BMS Run-Time Access" on page 7-28.

Reading from Boot Memory

When the DSP boots from an EPROM, the DSP uses the code in the boot ROM kernel to load the program from boot memory space. If further access to boot memory space is needed, the DSP may gain access to the boot memory space after the automatic boot process. To request access to boot memory, the DSP uses the PM instructions from boot memory (E_PI_BE), PM data from boot memory (E_PD_BE), or DM data from boot memory (E_DD_BE) bits in the E_STAT register.

Setting (=1) one of these bits overrides the external memory selects and asserts the DSP's $\overline{\text{BMS}}$ pin for an external memory transfer of the type corresponding to the bit.

Writing to Boot Memory

In systems using write-able EEPROM or FLASH memory for boot memory, programs can write new data to the DSP's boot memory using the same technique as "Reading from Boot Memory" on page 7-15, setting (=1) one of the E_PI_BE, E_PD_BE, or E_DD_BE bits to override the external memory selects and asserts the DSP's $\overline{\text{BMS}}$ pin for an external memory transfer.

Interfacing to External Memory

In addition to its on-chip SRAM, the DSP provides addressing of up to 4M words per bank of off-chip memory through its external port. This external address space includes external memory space—the region for standard addressing of off-chip memory.

This section provides the following topics:

- "Data Alignment—Logical vs. Physical Address" on page 7-16
- "Memory Interface Pins" on page 7-21
- "Memory Interface Timing" on page 7-24

Data Alignment—Logical vs. Physical Address

Data alignment through the external port depends on whether the system uses an 8- or 16-bit data bus. Figure 7-2 on page 7-17 shows the external port's data alignment. Each address in external, boot, and I/O memory corresponds to a 16- or 24-bit location, depending on the interface's configuration. A 16-bit data word occupies two bytes, and a 24-bit instruction word occupies four bytes (with an empty byte). When the system uses an 8-bit bus, two accesses are required for external 16-bit data, and three accesses are required for external instruction fetches or 24-bit data. When the system uses a 16-bit bus, one access is required for external 16-bit data, and two accesses are required for external instruction fetches or 24-bit data. For more information, see "External Bus Settings" on page 7-5.

To make it easier for programs to work with data alignment in external memory that varies with the external data format (16- or 24-bit), data size (16- or 24-bit) and the bus width (8- or 16-bit), the DSP supports logical addressing for programs and physical addressing for connecting devices to the external address bus.

Logical addressing normalizes addresses for 16- and 24-bit data in memory, creating a contiguous address map. The address map does not have a multitude of "holes" when addressing 24-bit data (e.g., an instruction fetch) in external memory.

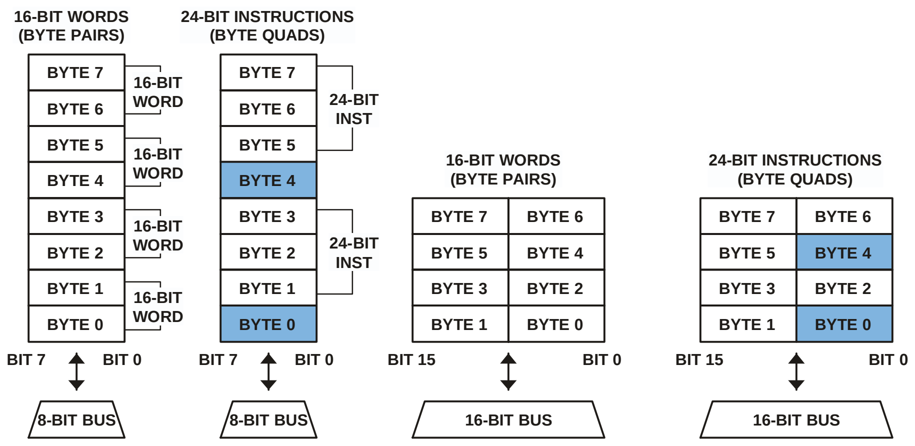


Figure 7-2. External Port Word Alignment

Physical addressing makes every location in external memory space available for addressing external devices using the external address bus. Whether using an 8- or 16-bit bus, the DSP can access each memory with the same granularity as the bus size.

The equation in Figure 5-4 on page 5-10 permits calculating the correlation between physical and logical addresses. The Format and Size factors for this equation appear in Table 7-3 on page 7-18 and Table 7-4 on page 7-18. This is a useful calculation when identifying the physical location for connecting an external device (such as a memory mapped I/O device) and identifying the logical location for addressing that device (such as the device's buffer address).

$$\text{Physical Address} = \text{Format Factor} \times \text{Size Factor} \times \text{Logical Address}$$

Figure 7-3. Physical Address Calculation

Table 7-3. Format Factor Address Multipliers

External Memory Data Format	16-bit	24-bit	16-bit	24-bit	16-bit	24-bit
E_DFS Bit	=0	=1	=0	=1	=0	=1
Transfer Type	24-bit Instr.	24-bit Instr.	24-bit Data	24-bit Data	16-bit Data	16-bit Data
Word Size	24-bit	24-bit	24-bit	24-bit	16-bit	16-bit
Transfer Size	24-bit	24-bit	16-bit[1]	24-bit	16-bit	16-bit[2]
Address Multiplier	x1	x1	x1	x1	x1	x2

1 Note that the transfer size is smaller than the words size, because the external memory format (E_DFS bit) and the transfer type do not match (16- versus 24-bit). In this case, the data is truncated, losing the lower 8 bits.

2 Note that this case has an address multiplier factor of x2, because the external memory format (E_DFS bit) and the transfer type do not match (24- versus 16-bit).

Table 7-4. Size Factor Address Multipliers

External Port Bus Size	16-bit	8-bit	16-bit	8-bit
E_BWS Bit	=1	=0	=1	=0
Transfer Size	24-bit	24-bit	16-bit	16-bit
Address Multiplier	x2	x4	x1	x2

For example, take the following programming and system design task of logically and physically addressing the following data:

- A 24-bit instruction fetch
- At logical address 0x2 0000
- An 24-bit external memory format (`E_STAT` register, `E_DFS` bit = 1)

- An 8-bit wide external bus (EMICTL register, E_BWS bit = 0)
- What is the physical address that the address lines in the system use for accessing this data?

For these parameters, use the physical address calculation as follows:

$$\text{Physical Address} = \text{Format Factor} \times \text{Size Factor} \times \text{Logical Address}$$

$$= 1 \times 4 \times 0x20000 = 0x80000$$

Figure 7-4. Example: Physical Address Calculation

Four useful combinations of external data format, data size, and bus size that cover the most common applications (where data format equals data size) are: 24-bit data over an 8-bit bus, 16-bit data over an 8-bit bus, 24-bit data over a 16-bit bus, and 16-bit data over a 16-bit bus. Table 7-5 on page 7-20, Table 7-6 on page 7-20, Table 7-7 on page 7-20, and Table 7-8 on page 7-21 show how logical and physical addressing compare for these cases.

Because boot memory space (like external memory space) also can contain 16- or 24-bit data and is accessible using the external address bus (with $\overline{\text{BMS}}$), the logical and physical addressing scheme also applies to external boot memory space accesses.

Boot memory space on the ADSP-2191's memory map starts at logical address 0x1 0000. For easy physical mapping when booting from an external EPROM, the ADSP-2191's boot kernel accesses data in the EPROM starting at physical address 0x0 0000.

The boot kernel accomplishes this by accessing logical address 0x80 0000, which (because the ADSP-2191 has 22 address lines)

Table 7-5. Example: 24-bit Format, 24-bit Data, and 8-bit External Bus

Logical Address	24-Bit Data Word	Physical Address	24-Bit Data Word
0x20000	0x123456	0x80000	unused
		0x80001	0x56
		0x80002	0x34
		0x80003	0x12
0x20001	0x789abc	0x80004	unused
		0x80005	0xbc
		0x80006	0x9a
		0x80007	0x78

Table 7-6. Example: 24-bit Format, 24-bit Data, and 16-bit External Bus

Logical Address	24-Bit Data Word	Physical Address	24-Bit Data Word
0x20000	0x123456	0x40000	0x5600
		0x40001	0x1234
0x20001	0x789abc	0x40002	0xbc00
		0x40003	0x789a

Table 7-7. Example: 16-bit Format, 16-bit Data, and 8-bit External Bus

Logical Address	16-Bit Data Word	Physical Address	16-Bit Data Word
0x20000	0x1234	0x40000	0x34
		0x40001	0x12
0x20001	0x5678	0x40002	0x78
		0x40003	0x56

Table 7-8. Example: 16-bit Format, 16-bit Data, and 16-bit External Bus

Logical Address	16-Bit Data Word	Physical Address	16-Bit Data Word
0x20000	0x1234	0x20000	0x1234
0x20001	0x5678	0x20001	0x5678

produces a physical address 0x0 0000. For more information on booting from an EPROM, see "External Memory Interface Booting" on page 14-20.

Because I/O memory space can contain 16-bit data and is accessible using the external address bus (with $\overline{\texttt{IOMS}}$), the logical and physical addressing scheme also applies to external I/O memory space accesses.

Memory Interface Pins

Figure 7-1 on page 7-2 shows how the buses and control signals extend off-chip, connecting to external memory. Table 7-10 on page 7-26 defines the DSP pins used for interfacing to external memory. The DSP's memory control signals permit direct connection to fast static RAM devices. Memory mapped peripherals and slower memories also can connect to the DSP using a user-defined combination of programmable waitstates and hardware acknowledge signals.

External memory can hold instructions and data. The external data bus (`DATA15-0`) must be 16 bits wide to transfer 16-bit data without data packing. In an 8- or 16-bit bus system, the DSP's on-chip external port unpacks incoming data and packs outgoing data. Figure 7-2 on page 7-17 shows how the DSP transfers different data word sizes over the external port.

The ADSP-2191 external memory interface differs from previous ADSP-218x DSPs. Compared to previous ADSP-218x DSPs, the interface uses a unified address space (no program and data memory separation) and supports configurable banks of external memory. The external interface provides glue-less support for many asynchronous and/or synchronous devices, including other DSPs.

Table 7-9. External Memory Interface Signals

Pin	Type	Function
ADDR 21-0	O/T	External Bus Address. The DSP outputs addresses for external memory and peripherals on these pins.
DATA 15-0	I/O/T	External Bus Data. The DSP inputs and outputs data and instructions on these pins. Pull-up resistors on unused DATA pins are not necessary. Read and write data is sampled by the rising edge of the strobe (RD or WR). In systems using an 8-bit data bus, the upper data pins (DATA 15-8) may serve as additional programmable flag (PF15-8) pins
$\overline{\text{MS3-0}}$ $\overline{\text{BMS}}$ $\overline{\text{IOMS}}$	O/T	Memory Bank/Space Select Lines. These lines are asserted (low) as chip selects for the corresponding banks of external memory. Memory bank size may be defined in the DSP's page boundary registers (MEMPGx). The select lines are asserted for the whole access.
CLKOUT	O/T	Clock output. Output clock signal at core clock rate (CCLK) or half the core clock rate, depending on the core:peripheral clock ratio.
$\overline{\text{RD}}$	O/T	Read strobe. RD indicates that a read of the data bus (DATA15-0) is in progress. As a master, the DSP asserts the strobe after the ADDR21-0 and $\overline{\text{MS3-0}}$/$\overline{\text{BMS}}$/$\overline{\text{IOMS}}$ assert.
$\overline{\text{WR}}$	O/T	Write strobe. WR indicates that a write of the data bus (DATA15-0) is in progress. As a master, the DSP asserts the strobe after the ADDR21-0 and $\overline{\text{MS3-0}}$/$\overline{\text{BMS}}$/$\overline{\text{IOMS}}$ assert.
ACK	I	Memory Acknowledge. External devices can de-assert ACK (low) to add waitstates to an external memory access when the waitstate mode is ACK mode, Both mode, or Either mode. ACK is used by I/O devices, memory controllers, or other peripherals to hold off completion of an external memory access. As a bus master, the DSP samples.
$\overline{\text{BR}}$	I	Bus Request. An external host or other DSP asserts this pin to request bus mastership from the DSP.
I (Input), O (Output), T (Three-state, when the DSP is a bus slave)		

Table 7-9. External Memory Interface Signals (Cont'd)

Pin	Type	Function
$\overline{\text{BG}}$	O	Bus Grant. The DSP asserts this pin to grant bus mastership to an external host or other DSP.
$\overline{\text{BGH}}$	O	Bus Grant Hung. The DSP asserts this pin to signal an external host or other DSP that the DSP core is being held off, waiting for bus mastership.
I (Input), O (Output), T (Three-state, when the DSP is a bus slave)		

On the ADSP-2191, Bank 0 starts at address 0x10000 in external memory and is followed in order by Banks 1, 2, and 3. When the DSP generates an address located within one of the four banks, the DSP asserts the corresponding memory select line, $\overline{\text{MS3-0}}$.

The $\overline{\text{MS3-0}}$ outputs serve as chip selects for memories or other external devices, eliminating the need for external decoding logic.

The $\overline{\text{MS3-0}}$ lines are decoded memory address lines that change at the same time as the other address lines. When no external memory access is occurring, the $\overline{\text{MS3-0}}$ lines are inactive.

Most often, the DSP only asserts the $\overline{\text{BMS}}$ memory select line when the DSP is reading from a boot EPROM. This line allows access to a separate external memory space for booting. For more information on booting from boot memory, see "Boot Mode DMA Transfers" on page 6-41. It is also possible to write to boot memory using $\overline{\text{BMS}}$. For more information, see "Using Boot Memory Space" on page 7-14.

Memory Interface Timing

Memory access timing for external memory space, boot memory space, and I/O memory space is the same. This section describes timing relationships for different types of external port transfers, but does not provide specific timing data. For exact timing specifications, refer to the *ADSP-2191 DSP Microcomputer Datasheet*.

ⓘ This section mentions the DSP's core (CCLK) and peripheral (HCLK) clocks. For information on using these clocks, see "Managing DSP Clocks" on page 14-29.

The DSP can interface to external memories and memory-mapped peripherals that operate asynchronously with respect to the peripheral clock (`HCLK`). In this interface there are latencies—lost core clock cycles—that occur when the DSP accesses external memory. These latencies occur as the external memory interface manages its two-level-deep pipeline and performs synchronization between the core and peripheral clock domains.

The number of latent cycles for an external memory access is influenced by several factors. These factors include the core:peripheral clock ratio, the data transfer size, the external bus size, the access type, the access pattern (single access or sustained accesses), and contention for internal bus access. These factors have the following influence on external memory interface performance:

- Core clock (`CCLK`)-to-peripheral clock (`HCLK`) ratio. The choice is between optimizing core speed or peripheral transfer speed. At the 2:1 ratio, the core can operate at up to 160 MHz, but the peripherals are limited to 80 MHz. At the 1:1 ratio, the core and peripherals can operate at up to 100 MHz.

- Core clock (`CCLK`)-to-memory bank clock (`EMICLK`) ratio. Each memory bank may apply an additional clock divisor to slow memory access and accommodate slow external devices.

- Data transfer size and external bus size. The DSP supports an 8- or 16-bit bus for transferring 16-bit data or 24-bit instructions. The best throughput is 16-bit data over a 16-bit bus, because in this case no packing is required.
- Access type (read and write accesses differ) and access pattern (single access or sustained accesses). These latencies have two sources: synchronization across core and peripheral clock domains and operation of the external memory interface pipeline.

Assessing these factors, there are two types of high performance systems. For a high performance system that requires minimal external memory access, use the 2:1 clock ratio, a 16-bit bus, and do most external memory access as DMA. For a high performance system that requires substantial external memory access, use the 1:1 clock ratio, a 16-bit bus, do as much external memory access as possible using DMA, and minimize single or dual (nonsustained) access.

ⓘ If a high-performance external memory interface is required, the system to avoid (because it does not use the strengths of the part) combines a 2:1 clock ratio, an 8-bit external bus, instruction fetches from external memory (with lots of cache misses), and uses minimal DMA for external memory accesses. This type of system causes unnecessary latency in external memory accesses.

Table 7-10 on page 7-26 shows external memory interface throughput estimates for the DSP operating at maximum core clock versus maximum peripheral clock. Some important conditions to note about the data in Table 7-10 on page 7-26 include:

- Assumes that the core is idle except for the transfers under test.
- Assumes there is no contention for the internal DSP core interface buses.
- Assumes the EMI clock divide is set to 1X and the Read/Write wait count = 0.

- Measures single access times beginning when the request is issued by the hard core and ending when the data is ready in the target memory.
- Includes the cycles to program the DMA descriptors and the cycles for the I/O processor to fetch the descriptors in the DMA single access times.
- Does not include the cycles to program the DMA descriptors or the cycles for the I/O processor to fetch the descriptors in the DMA sustained access times.

Table 7-10. External Memory Interface Performance at Maximum Core and Peripheral Clocks

			HCLK = 1/2 CCLK [1]				HCLK = CCLK [2]			
	DSP Word Size[3]	EMI Bus Size[4]	Single Access[5]		Sustained Accesses[6]		Single Access		Sustained Accesses	
			Cycles[7]	Words[8]	Cycles[7]	Words[8]	Cycles[7]	Words[8]	Cycles[7]	Words[8]
Direct Access										
Fetch	24	16			16	10			9	8.89
Fetch	24	8			20	8			11	7.27
Write	24	16	19	8.42	8	20	11	7.27	4	20.00
Write	24	8	23	6.95	12	13.33	13	6.15	6	13.33
Write	16	16	15	10.66	6	26.66	9	8.89	4	20.00
Write	16	8	19	8.42	8	20	11	7.27	4	20.00
Read	24	16	18	8.88	18	8.88	10	8.00	10	8.00
Read	24	8	22	7.27	22	7.27	12	6.67	12	6.67
Read	16	16	14	11.43	12	13.33	9	8.89	7	11.43
Read	16	8	18	8.88	16	10	11	7.27	9	8.89

Table 7-10. External Memory Interface Performance at Maximum Core and Peripheral Clocks (Cont'd)

			HCLK = 1/2 CCLK [1]				HCLK = CCLK [2]			
	DSP Word Size[3]	EMI Bus Size[4]	Single Access[5]		Sustained Accesses[6]		Single Access		Sustained Accesses	
			Cycles[7]	Words[8]	Cycles[7]	Words[8]	Cycles[7]	Words[8]	Cycles[7]	Words[8]
DMA Access										
Write	24	16	156	1.02	11.6	13.79	119	0.67	6.55	12.21
Write	24	8	160	1	13.6	11.76	121	0.66	7.55	10.60
Write	16	16	151	1.05	9.6	16.66	117	0.68	5.55	14.41
Write	16	8	155	1.03	11.6	13.79	119	0.67	6.55	12.21
Read	24	16	156	1.02	15.5	10.32	120	0.67	8.50	9.41
Read	24	8	160	1	19.5	8.20	122	0.66	10.50	7.62
Read	16	16	151	1.05	12.5	12.8	118	0.68	7.0	11.43
Read	16	8	156	1.02	15.5	10.32	120	0.67	8.5	9.41
Register Access										
Write	16	16	14	11.43	6	26.66	7	11.43	4	20.00
Write	16	8	18	8.88	8	20	9	8.89	4	20.00
Read	16	16	17	9.41	14	11.43	10	8.00	9	8.89
Read	16	8	21	7.62	18	8.88	12	6.67	11	7.27

1 Core clock CCLK = 160 MHz, Peripheral clock HCLK = 80 MHz
2 Core clock CCLK = 80 MHz, Peripheral clock HCLK = 80 MHz
3 *DSP Word Size* column is bits
4 *EMI Bus Size* column is bits
5 *Single Access* column refers to a single external memory read or write separated from the next external memory access by two or three instructions that do not access external memory.
6 *Sustained Accesses* column refers to repeated external memory access instructions
7 *Cycles* column is DSP core clock cycles
8 *Words* column is 1M words per second

Code Example: BMS Run-Time Access

The example in this section shows how to set up the external port on the ADSP-2191 for boot memory space (BMS) accesses.

The ADSP-2191 features an external boot memory space that can be accessed during runtime. When boot space is enabled, the ADSP-2191 uses the $\overline{\texttt{BMS}}$ pin for off chip memory access, selecting boot memory space.

The ADSP-2191 external port supports instruction and data transfers from the core to external memory space and boot space through the external port. The external port also provides access to external DSP memory and boot memory for ADSP-219x peripherals, which support DMA transfers. The external port is configurable for 8- or 16-bit data to provide convenient interfaces to 8- and 16-bit memory devices. Address translation and data packing is provided in hardware to allow easy translation between the core memory types (16- or 24-bit, word addressing) and address space and the external memory configuration (8/16-bit, Byte addressing).

The following listing shows how to set up a program for accessing 24-bit data from an external 8-bit memory device mapped to the ADSP-2191's boot space. After the external interface is configured, a single read is executed.

The External Memory Interface Control register (`EMICTL`) is used to configure the external port for an 8 or 16-bit external data bus. Beside that, the register provides a lock bit to disable write accesses to the external port memory access control registers. Separate register bits are also provided to set the read and write strobe sense for positive logic (bit=0) or negative logic (bit=1). These sense bits are common to all memory spaces. The data bus size and R/W sense bits are not written when the control register is written if the lock bit is set to 1 or an external access is in progress.

In this example, an 8-bit external device is mapped to boot space. The `EMICTL` register is setup for an 8-bit external bus and read/write strobes with negative logic.

```
IOPG = External_Memory_Interface_Page;
AR = 0x0070;
IO(EMICTL) = AR;
```

Because the device is mapped to Boot space, controlled by one of the memory access control registers (`MSxCTL`, `BMSCTL`, or `IOMSCTL`), the code configures the Boot Space Access Control Register (`BMSCTL`). Within the `BMSCTL` are six parameters that can be programmed to customize accesses to Boot memory space. These parameters are read waitstate count, write waitstate count, waitstate mode, base clock divider, write hold mode, and CMS output enable. To allow maximum flexibility, the `BMSCTL` is initialized with maximum waitstates, and base clock divisor.

```
AR = 0x0DFF;
IO(BMSCTL) = AR;
```

With `EMICTL` and `BMSCTL` configured, what remains is to configure the external port for 24-bit data and enable PM data boot space. This results in $\overline{\texttt{BMS}}$ being asserted any time DAG2 is used to access external memory.

```
IOPG = External_Access_Bridge_Page;
AR = 0x000A;
IO(E_STAT) = AR;
```

After the `EMICTL`, `BMSCTL`, and `E_STAT` have been initialized accordingly, it is now possible to use the PM data bus to perform accesses to external boot space.

The following is an example read from 0x80 0009 in external boot memory space.

```
DMPG2= 0x80;            /* Init DAG2 Page Register */
AX0 = 0;
```

```
I4= 0x0009;             /* Initialize DAG2 Pointer */
M4= 1;                  /* Initialize DAG2 Modifier */
REG(B4) = ax0;
L4= 0;                  /* Linear Addressing */
/* Perform External Boot Access */
MR0=PM(I4, M4);
```

Figure 7-5 on page 7-30 is an illustration of an example access:

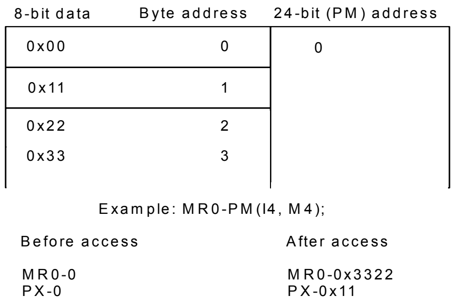


Figure 7-5. Example 8-to-24 Bit Word Packing

Following the access, the PM Bus Exchange (PX) register contains the LSB of the 24-bit data word, while MR0 contains bits 8-24 of the data word.

The following listing shows code for boot memory space initialization and operation in an ADSP-2191 system.

```
/ ***********************************************************

Purpose: This routine contains initialization code and accesses a
24-bit word from 8-bit External Boot memory space.

*********************************************************** */

#include <def2191.h>
```

```
/*
PM Reset interrupt vector code
*/

.SECTION /pm IVreset;
   jump Start;
   nop; nop; nop;

/*
Program memory code
*/

.SECTION /pm program;
Start:
_main:
   call Boot_Mem_Init;    /* Call Boot Memory Init Routine */
   call Boot_Mem_Access;    /* Read from External Boot memory */
   nop;
Loop_forever:
   jump Loop_forever;    /* Loop forever */

.SECTION /pm program;
Boot_Mem_Init:
/* Configure External Memory Interface */
   IOPG = External_Memory_Interface_Page;
   AR = 0x0070;
   IO(EMICTL) = AR;
     /* EMI control Register - Sets up for 8-bit external bus, */
     /* WS = Neg Logic, RS = Neg Logic, Split Enable */

   AR = 0x0DFF;
   IO(BMSCTL) = AR;
     /* Boot Space Access Control Register */
     /* max waitstates in case of slow EPROM */
/* Configure External Access Bridge */
   IOPG = External_Access_Bridge_Page;
   AR = 0x000A;
   IO(E_STAT) = AR;   /* EAB Config/Status Register - P */
                      /* data Boot Space, 24 bit data */
   RTS;

.SECTION /pm program;
Boot_Mem_Access:
   DMPG2= 0x80;       /* Initialize DAG2 Page Register */
   AX0 = 0;
   I4= 0x0009;        /* Initialize DAG2 Pointer */
   M4= 1;      /* Initialize DAG2 Modifier */
   REG(B4) = ax0;
   L4= 0;             /* Configure for Linear Addressing */
/* Perform External Boot Access */
   MR0=PM(I4, M4);    /* Reading from address 0x800009 */
                      /* in the Boot memory */
   RTS;
```

8 HOST PORT

This chapter provides the following sections:

Overview

The host port interface is an 8- or 16-bit asynchronous slave to the off-chip host processor. The primary use of this interface is to provide an external host with direct access to ADSP-2191 memory space, boot space, and I/O space. The ADSP-2191 acts as a slave while supporting and responding to accesses initiated by other host port masters. A host port master can be a microcontroller, FGPA, or another DSP.

This interface includes a DMA controller that eases the transfer of blocks of data between the ADSP-2191 memory/boot space and the external host processor. A functional diagram of the host port is shown in Figure 8-1 on page 8-2.

The host port signals are listed in Table 8-1 on page 8-2.

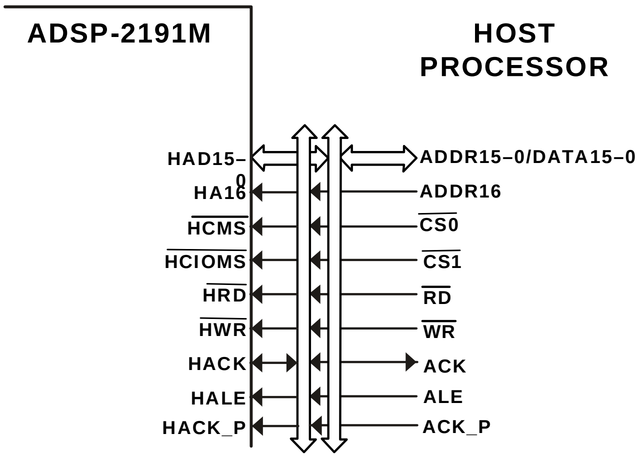


Figure 8-1. Host Port Functional Diagram

Table 8-1. Host Port Signals

Pin Name(s)	Input/Output	Function
HAD15:0	I/O/T	Host port multiplexed address and data bus
HA16	I	Host port MSB address bus
HACK_P	I	Host ACK polarity
HALE	I	Host port address latch strobe or address cycle control
$\overline{\text{HRD}}$	I	Host port read strobe
$\overline{\text{HWR}}$	I	Host port write strobe
HACK	I/O	Host port access ready acknowledge
$\overline{\text{HCIOMS}}$	I	Host port I/O space select
$\overline{\text{HCMS}}$	I	Host port memory select

The external host or the DSP can configure these host port access parameters:

- Memory/boot space map page number
- Memory/boot space data type (internal 16- or 24-bit data access)
- Data type
- External data bus size

The port logic provides address translation and packing/unpacking logic to allow mapping of 8-bit and 16-bit external accesses into 16- or 24-bit internal access data type.

The Host port can function in direct mode or host DMA mode.

- In direct mode, the host must provide an address before initiating the data exchanges for the transaction. The host can access the memory space, the boot space, and the I/O space.
- In host DMA mode, the host does not have to provide an address; the address is supplied by the DMA controller embedded in the host port logic. The host can access the memory and boot space but cannot access the I/O space.

The protocol and use of control lines is configured at reset. Other parameters of the interface such as data type, byte endian-ness, or address page can be programmed by software by the DSP core or the external host processor.

The host may use this port to directly access the entire DSP memory space map, the entire DSP boot space map, and one section of DSP I/O space map (I/O page[1:63]). Since the off-chip host has access to the complete ADSP-2191 on-chip peripheral I/O space (except page 0 space), the host may take control of any of the I/O mapped peripherals from the DSP.

Host port activity may impact DSP performance. The DSP stalls for one cycle when the host port accesses DSP internal memory. The DSP core may also have to wait in case of access conflict through the same interface. For example, if both the DSP and host port try to use the external port to access external space (memory, boot, or I/O), a wait period for the DSP or the host port may occur. Host port access to on-chip or off-chip I/O space can sometimes be accomplished without DSP cycle penalty.

A transaction on the host port is completed when the total number of data bytes, as defined by the data type, have been transferred to the ADSP-2191 internal bus. Depending on the data bus size and the data type, as described in Table 8-2 on page 8-4, a total of one to four host data accesses may be needed to complete the transaction.

Note: The HPI expects 24-bit data to be left aligned.

Table 8-2. Access Cycles for One Host Transaction

Mode	Data Bus Size	Data Type	Complete Transaction	
			Host Address Cycles	Host Data Access Cycles
Direct	8	16	1	2
	8	24	1	3 or (4)
	16	16	1	1
	16	24	1	2
Host DMA	8	16	0	2
	8	24	0	3 or (4)
	16	16	0	1
	16	24	0	2

Host Port Setup Parameters

This section provides the following topics:

Overview

In direct mode, an internal data transaction is composed of an address phase and a data phase, and is triggered by host access to the host port. The data can be 16 bits or 24 bits and is mapped into a packet of one, two, three, or four consecutive host accesses. Before performing a transaction, the host should have a number of parameters configured in I/O mapped registers. The parameters can be set up by the external host or by the DSP core.

Set up the host port memory page register to contain the most significant bits of the address that will be accessed (9 bits of memory page). The data type (16 or 24 bits) is also set up in this register. Also configure the memory space (memory or boot) that will be accessed.

If desired, the bus data width should be modified as indicated in the next section. The data type must be specified in the host port memory map register. The data type configuration bit is used only for a memory/boot transaction; it defines the size of the data entity to be transferred. The data type for an I/O transaction is always fixed at 16 bits. Depending on the data type and the bus data size, packing or unpacking and address translation operations may be involved. Packing logic assembles and disassembles words between the external data path width (8-bit or 16-bit) and the internal data types (16-bit or 24-bit). Table 8-3 on page 8-6 describes the packet size and the number of data phases required to complete a transaction.

Table 8-3. Packet Sizes

Internal Data Type	External Data Bus Width	Packet Size
16	8	2 bytes
16	16	1 16-bit word
24	8	3 or 4 bytes (the mode bit in the host port configuration register enables a packet of 4 bytes.)
24	16	2 16-bit words

Data Bus Width and Address Bus

The host port is set to an 8-bit data path width (default) after hardware reset. The host can change the data path width to 16 bits by writing the proper value to the I/O mapped host port configuration register.

The size of the internal data is defined by the data type: 16 or 24 bits. Logic in the host port as well as the bus protocol handles moving, packing, and unpacking information from/to the external host.

The address that the host provides to the port is always a byte address, regardless of the data width configuration or the data type. `HAD[0]` determines whether odd or even byte. The address is 17 bits wide total: 16 bits multiplexed on the data/address bus, and 1 bit (MSB) on a separate line `HA[16]`. In the case of a data bus width of 16-bits, the value of the LSB address bit is not used ("don't care" during an address cycle).

The address on the DSP address space is generated from the host address bus according to Table 8-4.

Table 8-4. DSP Address Generation

Space	Data Type	DSP Address	DSP Address Size
IO space	16	{0, 0, HA16, HAD[15:1]}	18 bits
Memory/Boot Space	16	{MPAGE[8:1], HA16, HAD[15:1]}	24 bits
	24	{MPAGE[8:0], HA16, HAD[15:2]}	24 bits
	MPAGE is 9 bits from the Direct Page Register (HPMMR[15:7])		

For direct mode, bit 1 of the `HPPR` register sets the data type. For DMA mode, bit 2 of the `HOSTD-CFG` register sets the data type.

Packing Parameters

Data organization and address translations performed within the host port are defined by two packing parameters: data byte endian-ness and data ordering.

Endian-ness:

- Little endian = 0
- Big endian = 1

Data ordering:

- LSB first = 0
- MSB first = 1

The host port logic maps one, two, three, or four consecutive host accesses into a single ADSP-2191 I/O or DMA access. In this way, the host port reads are pre-fetched (one value up to 24-bits is loaded into the pre-fetch read buffer), and writes are posted (one value up to 24-bits is loaded into the write buffer).

Requests for internal access are made relative to the start of a transaction or the end of a transaction. The start of a transaction (or start of a packet) is defined by endian-ness, ordering bits, bus size, data type, and address bits, as summarized in Table 8-5 on page 8-8. The end of the transaction is given by the count of access strobes after the beginning of the packet.

Table 8-5. Start of Transaction Determination

Data Bus Size	**Data Type**	**Start of a transaction if LSB of address HAD[1:0] =**			
		Endian-ness = 0 (little endian)		**Endian-ness = 1 (big endian)**	
		Ordering =1 (MSB first)	**Ordering =0 (LSB first)**	**Ordering =1 (MSB first)**	**Ordering =0 (LSB first)**
8	16	x1	x0	x0	x1
16	16	Every Host Access	Every Host Access	Every Host Access	Every Host Access
8	24	11	00 - 4 bytes 01 - 3 bytes	00	11 - 4 bytes 10 - 3 bytes
16	24	1x	0x	0x	1x

A host port write triggers an internal write if the host access corresponds to the last address associated with the data entity. This is the last access of a data packet.

While assembling a larger word, the host port logic automatically asserts `ACK` for each byte access that does not start a transaction, write or read. For accesses that start a transaction, write or read, `ACK` is returned when the host port is not busy. This occurs when the read data has been loaded into the read buffer and previous write access (if any) has successfully completed to memory.

Control Signals

The host port has two select pins: $\overline{\text{HCMS}}$ and $\overline{\text{HCIOMS}}$, both active-low. With $\overline{\text{HCMS}}$ asserted, an external host can directly access the full ADSP-2191 memory space and the full boot space. In the direct mode, assertion of the $\overline{\text{HCIOMS}}$ pin allows access to all on-chip and off-chip I/O space. Only one select pin can be driven active at a time. When a select is de-asserted, any ongoing access is aborted (completed).

In addition to the two select signals, transactions on the host port are controlled by four signals: HALE, $\overline{\text{HRD}}$, $\overline{\text{HWR}}$ and HACK. The functionality of HALE and HACK, and the polarity of the $\overline{\text{HRD}}$ and $\overline{\text{HWR}}$ signals, are configured at reset by the chip hardware. Their values must be defined during the reset sequence, and kept for ten peripheral cycles after the reset is de-asserted. The values, sensed during the hardware reset sequence, are stored in the host port configuration register as read-only bits.

HACK default mode is programmed by hardware at reset by sensing the values driven on the HACK and HACKP pins. During the rising edge of reset and for ten cycles after the reset, the HACK pin is configured as an input. This may require an external pull-up or pull-down resistor. The sensed value is returned as a default value driven to the HACK pin after reset. This default value is also used to define the HACK functional behavior as described in the next section. The HACK functionality can be further modified by software, either by the host or by the DSP. The HACK functional mode is latched into configuration bits in the host port configuration register.

HACK polarity is defined by the level driven during reset on the HACK_P pin. If high, the HACK is active-high; if the level is low, HACK is active-low.

Address Latch Enable/Address Cycle Control (HALE)

HALE is programmed during hardware reset to function in one of two modes: address latch enable (ALEM) mode or address cycle control (ACCM) mode.

ALEM
If the HALE pin is held low by an off-chip resistor or an external host during the assertion of the RESET pin, the HALE pin functions as an address latch enable. In this mode, HALE is active-high. The host port latches the address from the HAD[16:0] bus at a falling edge transition of the HALE. In ALEM mode, the $\overline{\mathtt{HWR}}$ pin must be held high during address transfers.

Use ALEM when interfacing to micro-controllers with multiplexed address and data pins (for example, the 8051 family). For this type of system, ALEM works well with the micro-controller's multiplexed bus hardware.

ACCM
If the HALE pin is held high during hardware reset, this pin functions as an address cycle control pin. As an address cycle control pin, HALE is active-low. A logic zero on the HALE pin causes a trailing edge transition of the $\overline{\mathtt{HWR}}$ pin to latch an address into the host port. During the address cycle, the HACK is returned to the host in the same way as for a data write cycle. In this mode, HALE can be used like an address or a select line. For HALE to be active, one of the select signals ($\overline{\mathtt{HCMS}}$ or $\overline{\mathtt{HCIOMS}}$) must be active.

The HALE sense bit is readable as part of the host port configuration register. To be properly sampled at initialization, the default value of HALE must be maintained for ten peripheral clock cycles after the reset has been de-asserted.

Use ACCM when interfacing to controllers with separate address and data buses. For this type of system, the address cycle and data cycle can be controlled by software.

$\overline{\mathtt{HRD}}$ and $\overline{\mathtt{HWR}}$ Data Strobes

On a write access, the data bus is sampled by the host port on the trailing edge of the $\overline{\mathtt{HWR}}$ write strobe. On a read access, the host port provides the data after the leading edge of the $\overline{\mathtt{HRD}}$ read strobe. If the host port does not return the data acknowledge to the external host, the host should keep the strobe signal in an active state waiting for the host port data access to complete.

In the ACCM mode, with HALE low, the write strobe trailing edge samples the address value on the HA/HAD bus.

The polarity of the strobes is defined by the default inactive state driven on the pin when the ADSP-2191 is in reset. To be properly sampled at initialization, the default value (inactive state) of strobes ($\overline{\text{HRD}}$, $\overline{\text{HWR}}$) must be maintained for ten peripheral clock cycles after the reset has been de-asserted.

Read and Write Timing Diagrams

Figure 8-2 on page 8-12 shows host port normal read timing. Figure 8-3 on page 8-13, Figure 8-4 on page 8-14, and Figure 8-5 on page 8-15 show host port pre-fetch read timing.

Acknowledge/Ready

The HACK signal can be active-high or active-low, depending on the reset sequence (based on the value driven on the HACK_P pin). HACK polarity is stored in the host port configuration register as a read-only bit.

The HACK default value is based on the value sensed on the HACK pin during the reset sequence.

HACK indicates to the host when to complete an access. For a read transaction, a host can proceed and complete an access when a valid data is present in the read buffer and the host port is not busy doing a write. For a write transaction, a host can complete an access when the write buffer is not full (the host port is not busy doing a write).

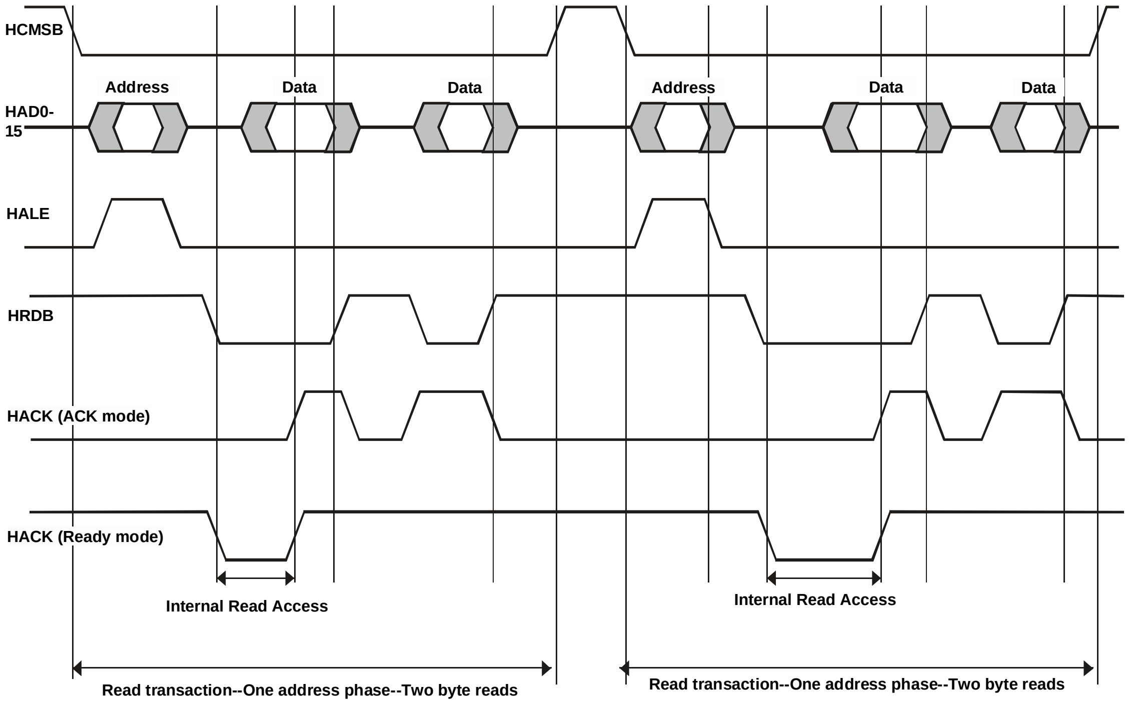


Figure 8-2. Normal Read Triggered from $\overline{\text{HRD}}$ Leading Edge (ALEM)

Two mode bits in the host port configuration register HPCR[7:6] define the functionality of the HACK line. Those two bits are initialized at reset, based on the values driven on the HACK and the HACK_P pins as presented in Table 8-6 on page 8-15. They can be modified after reset by a write access to the host port configuration register.

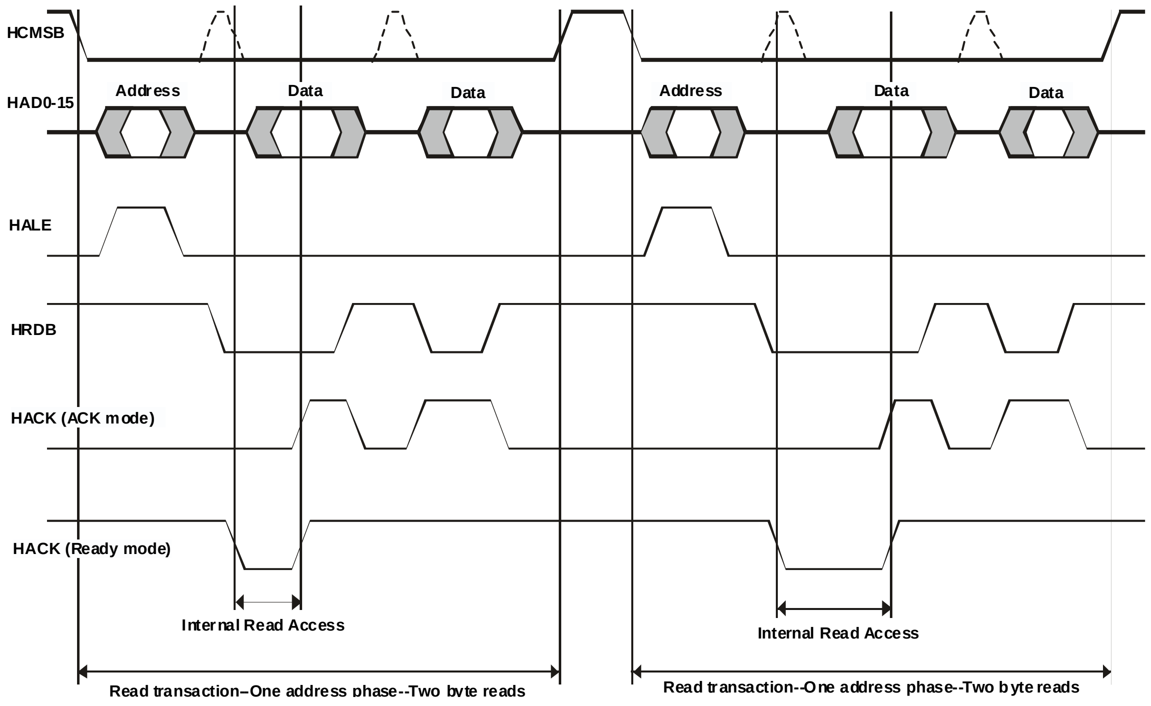


Figure 8-3. Pre-Fetch Read Triggered from HALE Falling Edge (ALEM)

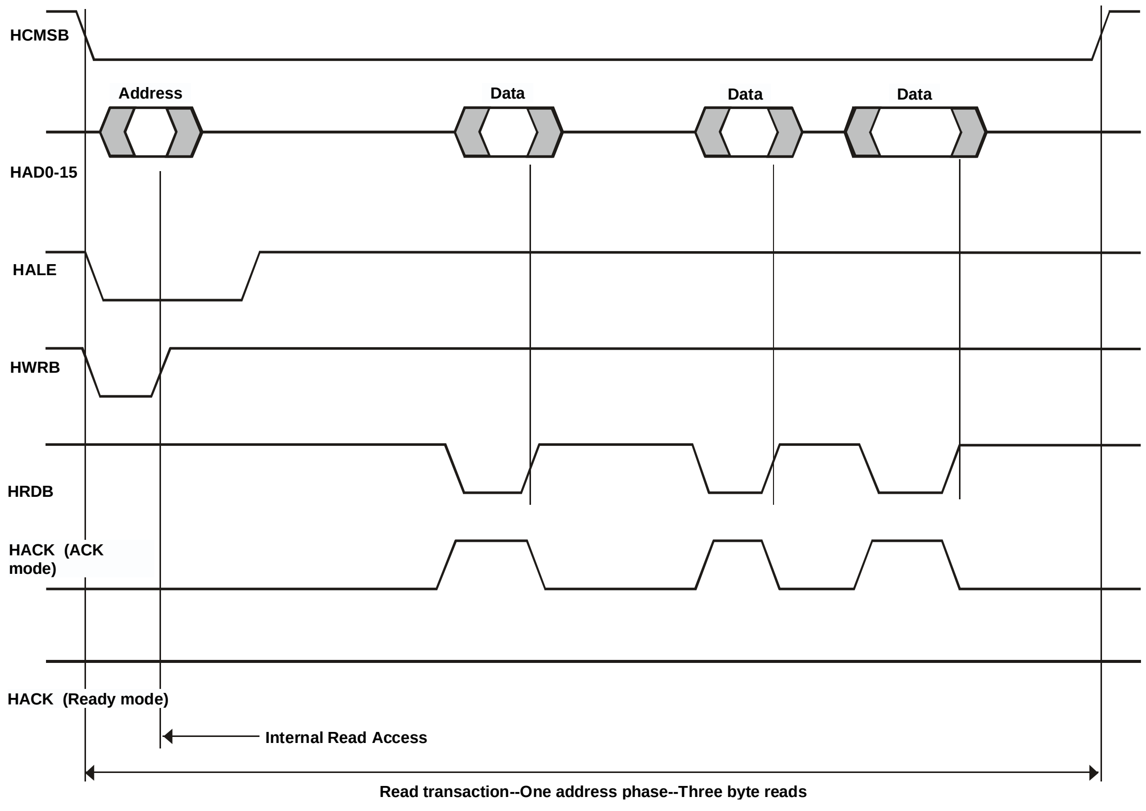


Figure 8-4. Pre-Fetch Read Triggered from $\overline{\texttt{HWRB}}$ Trailing Edge (ACCM)

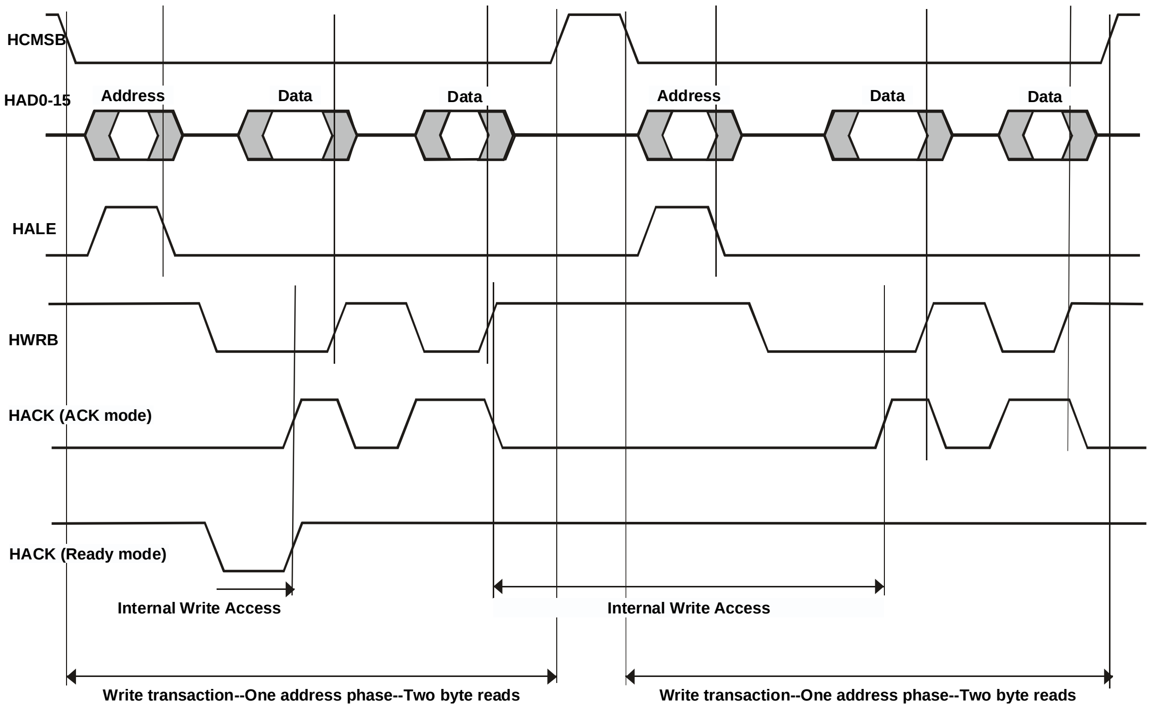


Figure 8-5. Write Triggered from Last $\overline{\texttt{HWR}}$ Trailing Edge (ALEM)

Table 8-6. HACK Mode Bits

Values driven at Reset		HPCR[7:6] Initial values		Acknowledge Mode
HACK_P	HACK	HPCR[7]	HPCR[6]	
0	0	0	1	Ready Mode
0	1	0	0	ACK Mode
1	0	0	0	ACK Mode
1	1	0	1	Ready Mode

The functional modes (assuming active-high signal) selected by HPCR[7:6] are:

- **00** (ACK mode): Acknowledge is active on strobes; HACK goes high from the leading edge of the strobe to indicate when the access can complete. After the host samples the HACK active, it can complete the access by removing the strobe. The host port then removes the HACK.
- **01** (READY mode): Ready active on strobes, goes low to insert waitstate during the access. If the host port can not complete the access, it drives the HACK/READY line inactive. In this case, the host must extend the access by keeping the strobe active. When the host samples the HACK active, it can then proceed and complete the access by removing the strobe.
- **10** Reserved
- **11** Reserved

In ACK or READY modes, the HACK is returned active for any address cycle. Waveform diagrams presented below are with HACK_P high; HACK is active-high.

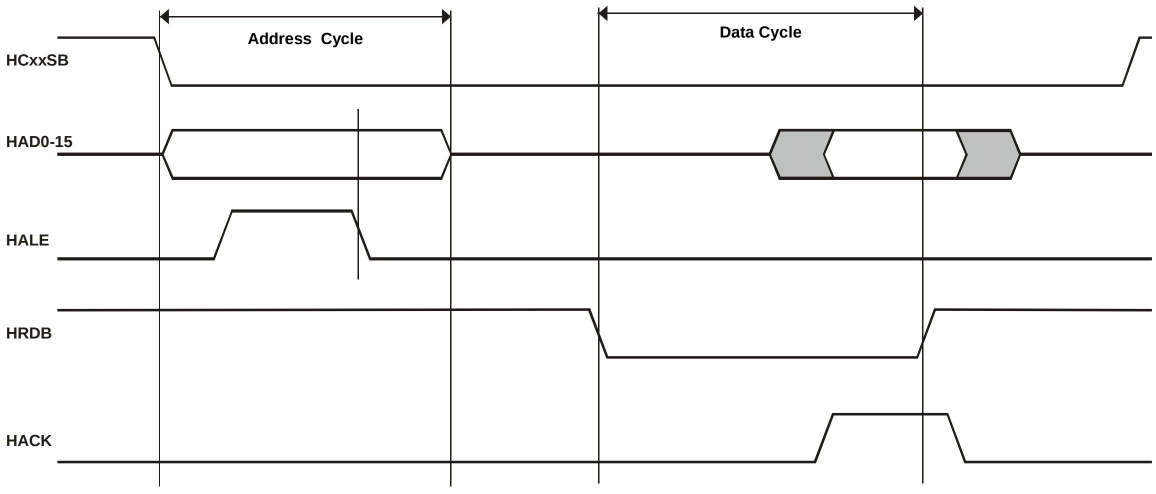


Figure 8-6. Waveforms for HACK Mode 00 (ACK Acknowledge)

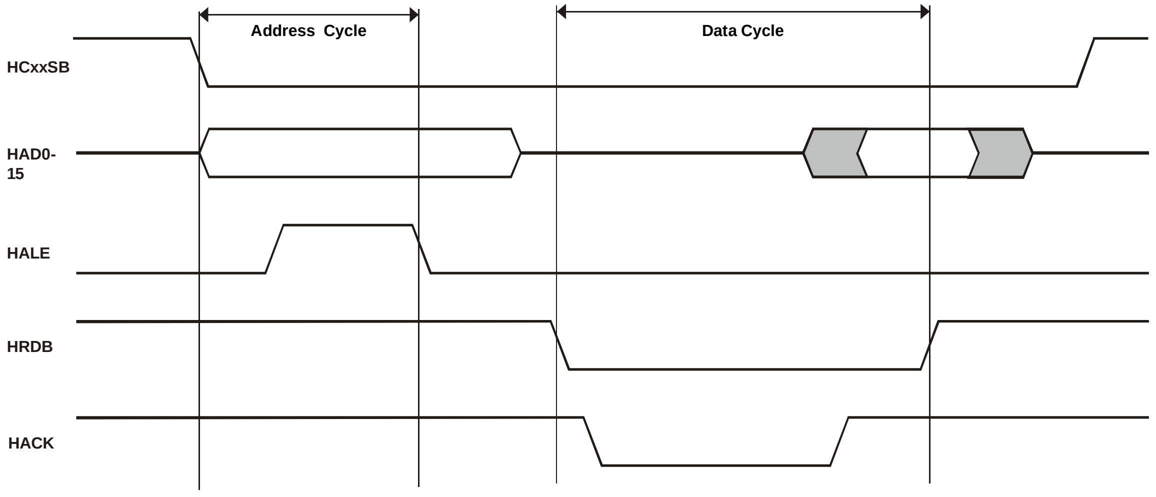


Figure 8-7. Waveforms for HACK Mode 01 (Ready)

Direct Access Mode Transactions

This section provides the following topics:

Direct Access Mode

Due to the difference between the external data bus width and the internal data type, several host accesses (a packet) may be required to complete the transaction. It is possible to have the host send an address for every data access or to send an address only for the first access of the transaction. The host port provides direct single data access and direct burst access capability. The burst size is 2, 3, or 4 words when the data bus width is 8 bits. The burst size is 1 or 2 words when the data bus width is 16 bits.

In direct access mode, the host processor must execute an address cycle to provide a packet address. This address cycle indicates the start of the read or write transaction. The address must always be in line with the byte endian-ness and ordering configuration bits: if ordering = 0, the address corresponds to the LSB byte or the LSB word, if ordering = 1, the address corresponds to the MSB byte or the MSB word. The two LSB bits determine whether the address is the first address of a packet.

Address cycles following the first address cycle of a packet are ignored, except when the two LSB bits of the address indicate a new packet start as defined in the previous table. Additional read or write strobes without intervening address latch strobes, result in the auto-increment or auto-decrement of the LSB bits of the address latched by the ADSP-2191. This mechanism allows transfer of the data packet as a burst. However, in direct access mode, a packet transfer transaction must always start with an address cycle where the address is the first word of the packet.

The host port is always ready to accept an address cycle without slowing down the host. In the case of an address cycle, the acknowledge signal is always returned (ACK or READY modes).

In the case of a data read cycle, the host port may have to wait for the data to be available before returning the `ACK` or may have to wait for the host port to complete a previous write. In the case of a write cycle, the host port may have to wait for a previous write to complete before returning the `ACK`.

Direct Access Read Modes

On a read transaction in direct access mode only, a host port can trigger an internal read at three different points. Based on the configuration bits, this can be done on the first data phase of a transaction (normal mode), on the address phase (pre-fetch mode), or at the last data phase of a transaction (pipeline mode). In pipelined read mode, the read internal access is triggered at the end of the packet, preparing the data for the next read transaction.

Table 8-7. Direct Access Read Modes

HPCR[5] Prefetch Read	HPCR[4] Pipeline Read	Read mode
0	0	Normal read, on first data read strobe
0	1	Pipelined read, on last data read strobe
1	0	Prefetch read on every address cycle
1	1	Reserved

- Normal read: `HPCR[5:4]= 00`
  Read request generated at the beginning of the data packet, from the first read strobe (leading edge).
- Pre-fetch read: `HPCR[5:4]= 10`
  Read request generated at the beginning of the data packet at the end of the address cycle (trailing edge of `HALE` or $\overline{\texttt{HWR}}$).
- Pipelined read: `HPCR[5:4]= 01`
  Read request generated at the end of the data packet read (from trailing edge of the last read strobe).

Direct Access Mode Timing Diagrams

Figure 8-8 on page 8-21 shows host port ALE mode read timing, Figure 8-9 on page 8-22 shows host port ACC mode read timing, Figure 8-10 on page 8-23 shows host port ALE mode write timing, and Figure 8-11 on page 8-23 shows host port ACC mode write timing.

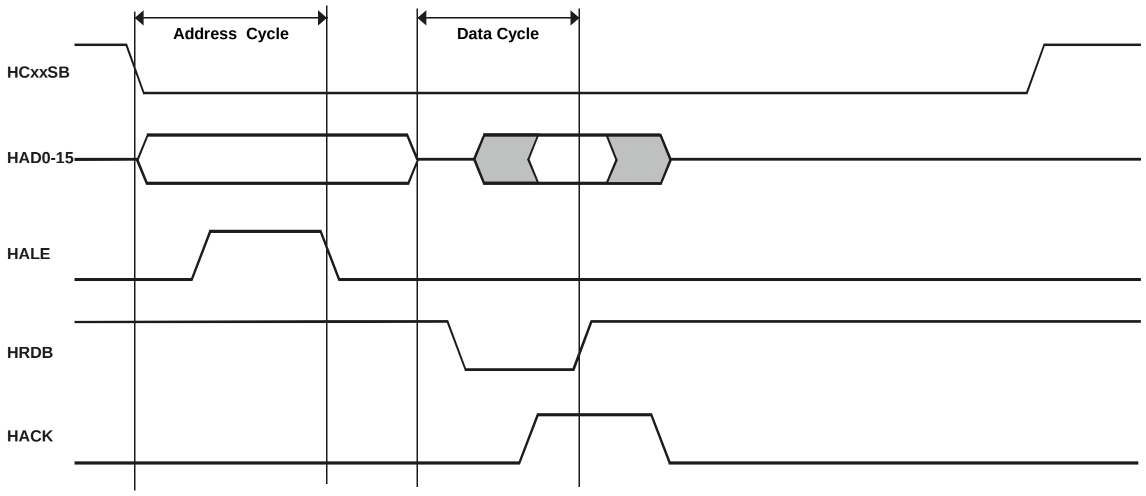


Figure 8-8. Direct Access Mode Read (ALEM)

Figure 8-12 on page 8-24 shows host port ALE mode burst read timing, Figure 8-13 on page 8-25 shows host port ACC mode burst read timing, Figure 8-14 on page 8-26 shows host port ALE mode burst write timing, and Figure 8-15 on page 8-27 shows host port ACC mode burst write timing.

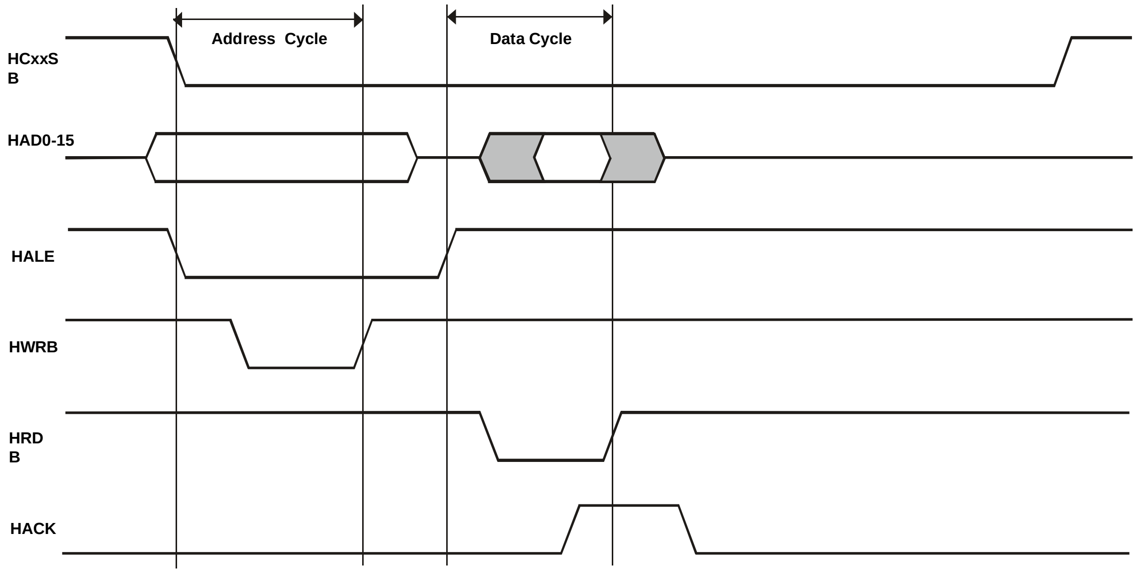


Figure 8-9. Direct Access Mode Read (ACCM)

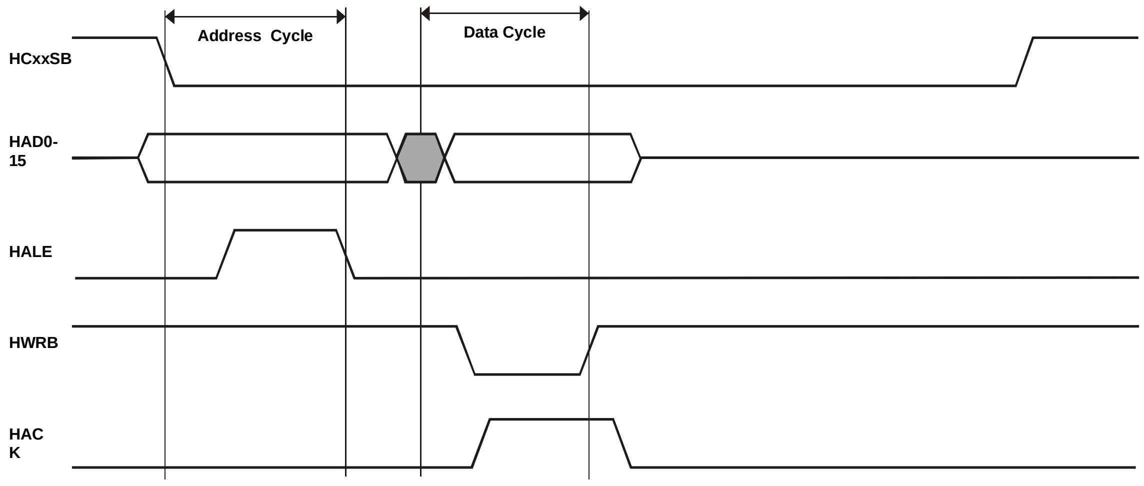


Figure 8-10. Direct Access Mode Write (ALEM)

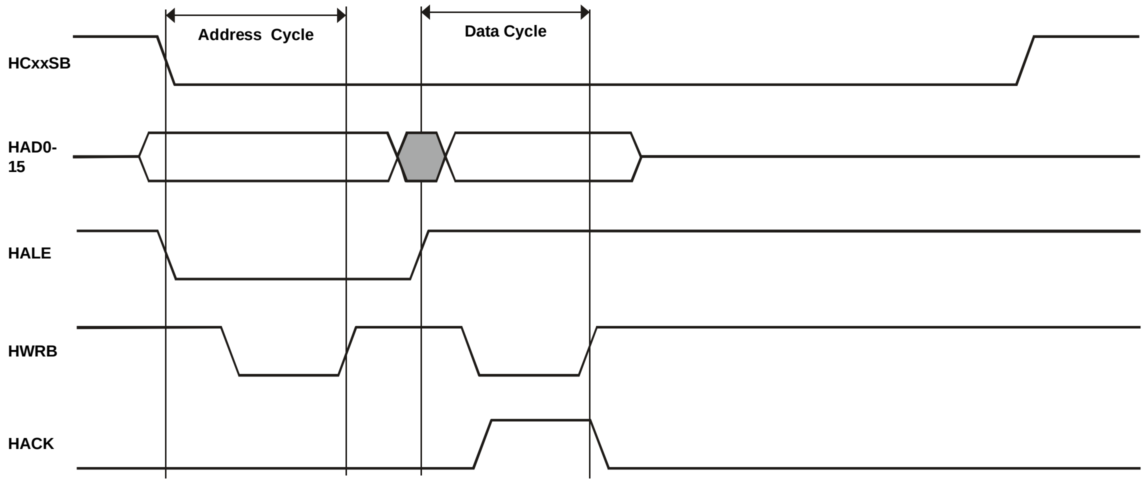


Figure 8-11. Direct Access Mode Write (ACCM)

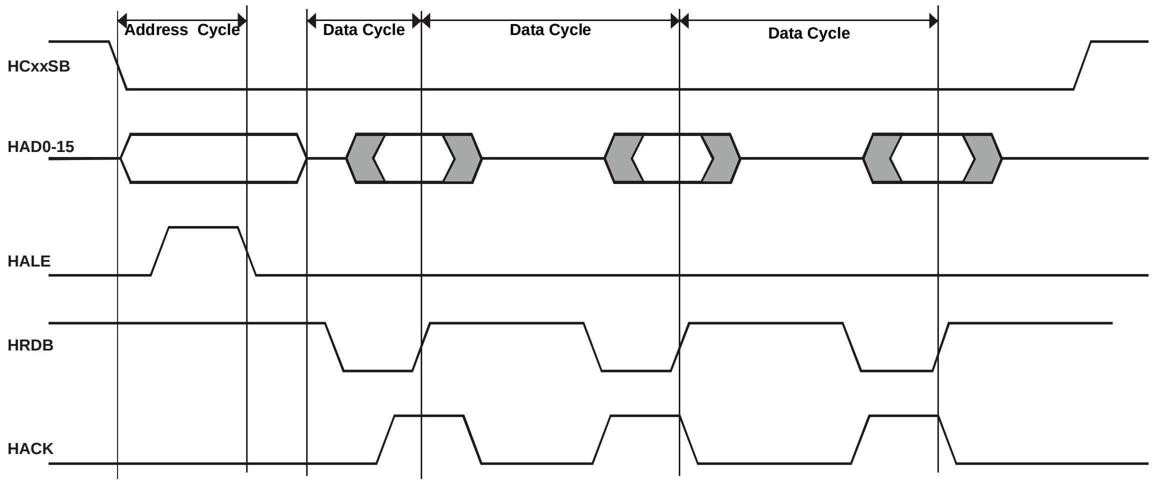


Figure 8-12. Direct Access Mode Burst Read (ALEM)
(3 Byte Read, 24-Bit Data Type)

Host Port DMA Mode Transactions

This section provides the following topics:

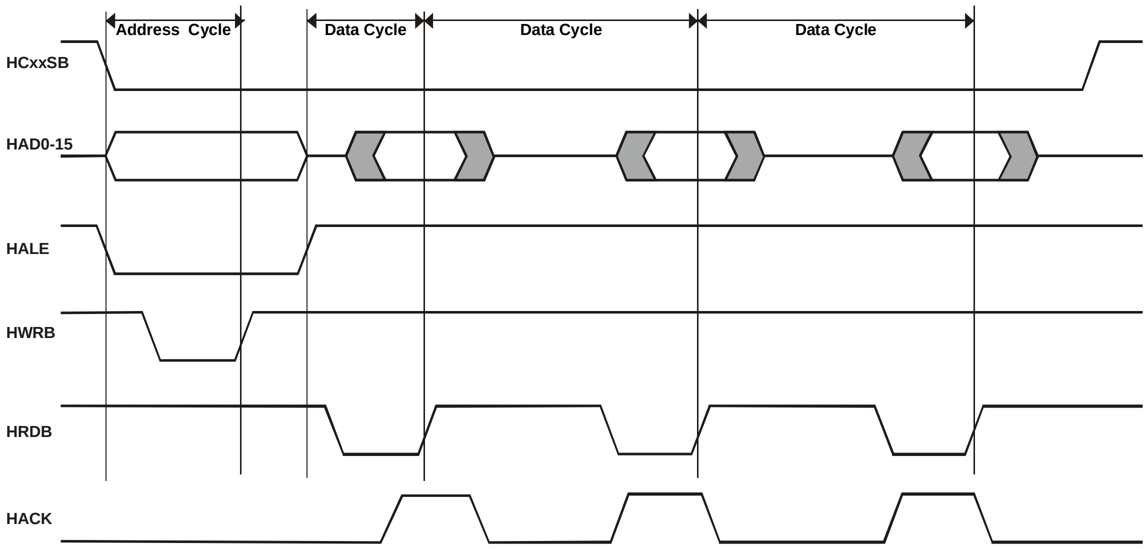


Figure 8-13. Direct Access Mode Burst Read (ACCM)
(3 Byte Read, 24-Bit Data Type)

Host Port DMA Mode

The host port includes a DMA controller that, when enabled, allows transfer of multiple blocks of data. A data block transfer is defined by parameters loaded into the host port DMA parameter registers. In this mode, the host action is only to send selects and data strobes to trigger the progress of the data transfer. On every data strobe, the host samples the data from the bus in the case of a read access (block transfer from ADPS-2191 to host) or drives the data onto the bus in the case of a write transfer (block transfer from the host to the ADSP-2191 memory).

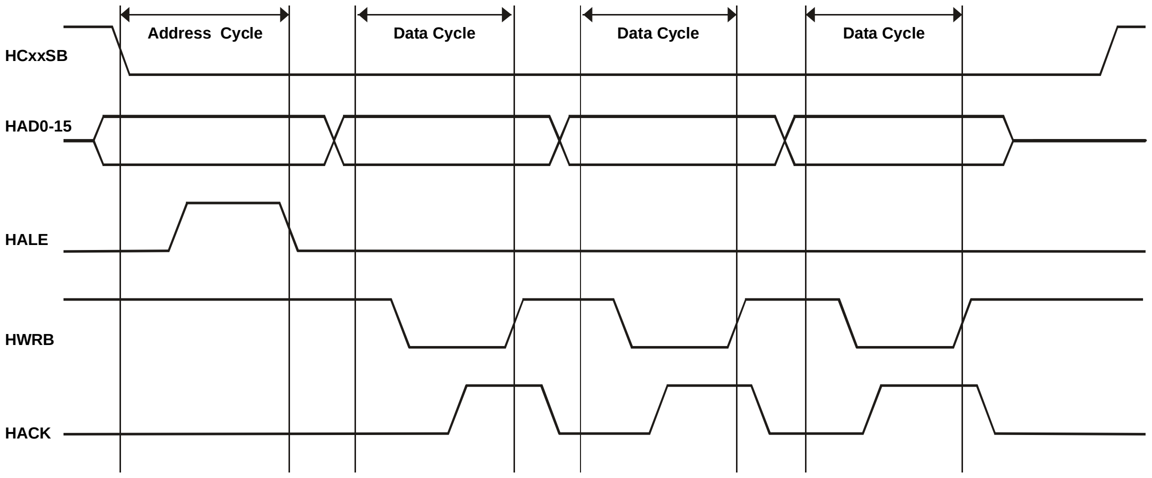


Figure 8-14. Direct Access Mode Burst Write (ALEM)
(3 Byte Write, 24 Bit Data Type)

When host port DMA is enabled, the host should not send any address and should not initiate a memory/boot transfer with an address cycle while the $\overline{\texttt{HCMS}}$ is active. If the host does an address cycle, the host port ignores the address.

The data strobes sent by the host must be in line with the DMA direction parameter set in the DMA configuration register. If the direction is “0”, the host must perform read cycles (read strobes); if the direction is “1”, the host must perform write cycles (write strobes). If the wrong strobe is used, the sequencing of the host port and DMA logic is not affected.

Data strobes clock the advancement of the packing/unpacking logic. The host port tracks of the start and end of a packet from the start of a block transfer.

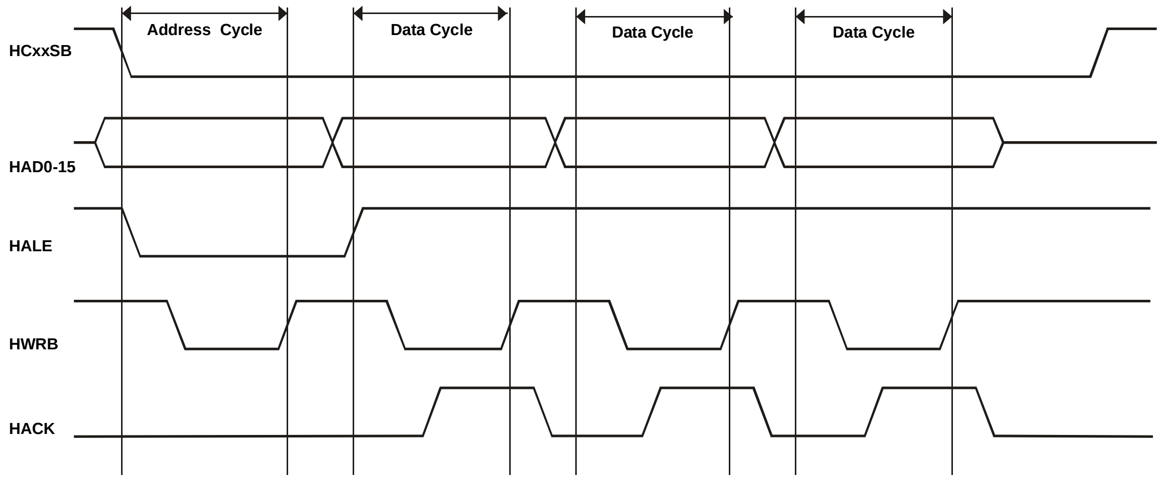


Figure 8-15. Direct Access Mode Burst Write (ACCM)
(3 Byte Write, 24-Bit Data Type)

In host port DMA mode, the reads are performed internal to the host port in pipelined read mode only. The host port generates the first read request as soon as the DMA is ready and configured in read mode. This data is stored into the read buffer and can be read upon receiving read strobes from the host. After the completion of a packet read, the host port sends an internal DMA request to reload the read buffer with the next data.

For a write transfer, the host port generates a write DMA request at the end of every packet, when the write buffer is full.

Host Port DMA Controller

The host port DMA controller provides the basic functionality for the host port DMA mode. The host port DMA controller has a set of DMA work unit definition registers. This set of registers describes a DMA work unit whose source or destination can be anywhere in DSP memory space, or in boot space. The DMA does not address I/O space. DMA parameters (five register values) are grouped into a descriptor block that can be stored in memory space page 0. The host port DMA controller downloads a descriptor block before starting the DMA transfer. Multiple descriptor blocks can reside in memory and can be linked together as a list of descriptors describing a complete complex task of transfers.

The ADSP-2191 DSP core may configure the host port DMA controller, or the off-chip host may use the host port to configure the host port DMA controller to perform a DMA access through the port. During a host port DMA access, the host port DMA controller loads the DMA address and other parameters from an internal memory resident DMA descriptor block. The host is required to strobe out read data or strobe in write data while the host port automatically increments the address.

Bus Arbitration and Usage Restrictions

When the DSP core and the off-chip host attempt to use the host port at the same time, there are some restrictions on the DMA controller operations. These include:

- The modification of the host port DMA parameters is limited in the same way as for the DSP core. If the external host attempts to latch a memory address ($\overline{\text{HCMS}}$ active) via the host port while the host port DMA is enabled (host port DMA mode), the address cycle is ignored and the address is discarded.
- If a $\overline{\text{HCMS}}$ qualified data strobe is asserted but does not correspond to the current DMA setting (example: assert write strobe while the DMA is enabled in read mode), the strobe is ignored.

- If the host makes a memory/boot access while DMA descriptors are changing, the acknowledge will take longer to be asserted. Use the DMA auto-buffer mode to avoid this.

- Host port DMA mode is only for memory and boot space ($\overline{\texttt{HCMS}}$). It is possible to host accessing I/O mapped-register ($\overline{\texttt{HCIOMS}}$) in direct access mode while DMA is enabled. However, this can be destructive with an I/O read when the DMA is in read mode, or with an I/O write outside of a packet boundary of the host DMA data stream.

  For the first case, the data read back from both internal memory location and I/O location will corrupt each other within HPI read buffers; this should be avoided. But a host can do write accesses via DMA auto mode while HPI does direct I/O read accesses, given the conditions that (1) DMA is enabled and (2) the valid address is presented in terms of access type (HPI external data type, and so on). This kind of usage was never tested though.

  For the second case, I/O write is not possible if DMA is enabled since the DMA bus will not get a direct access request from HPI. DMA must be disabled before any type of I/O write access. So the second case does not stand.

The DSP core should be held off from write accessing the host port DMA controller/host port I/O space while it is in use by the host. If both DSP and external host are likely to use the host port DMA controller/host port bus at the same time, employ a high-level synchronization protocol to avoid a race condition, as described in "Using Semaphores" on page 8-30.

Using Semaphores

The host port semaphore (HPSMPHx) registers ease development of token passing and other host/DSP communication protocols. These protocols let the DSP or host request that the other hold off access to I/O memory to avoid contention for the same locations (and information).

To use an HPSMPHx register, the host or DSP reads it. This read sets (=1) the register, indicating to the other (depending on how the protocol is written) that the shared resource (for example, I/O memory) is being used. When done with the resource, the host or DSP writes 1 to the register, clearing (=0) it.

Host Port DMA Mode Timing Diagrams

Figure 8-16 shows timing for a host port DMA read, and shows timing for a host port DMA write.

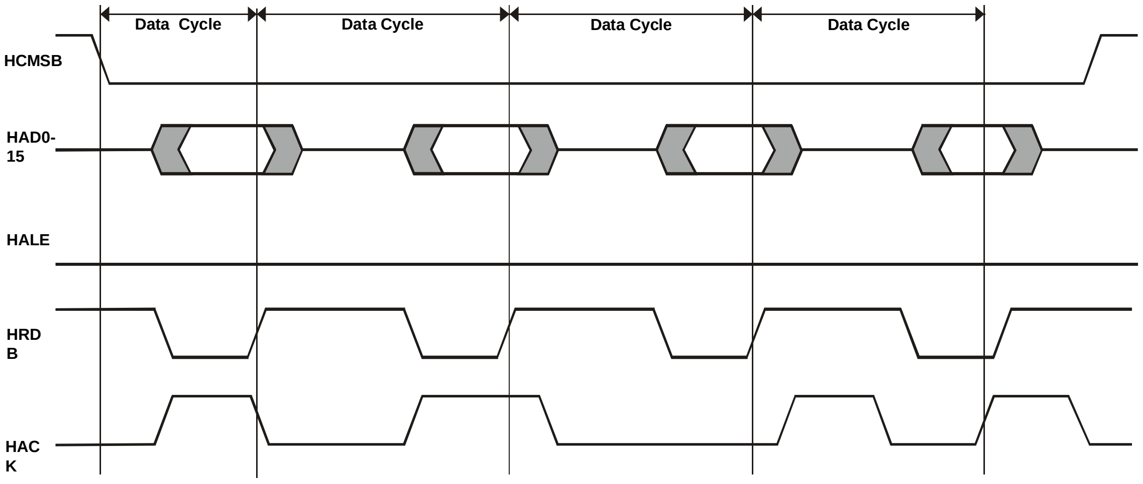


Figure 8-16. Host Port DMA Read

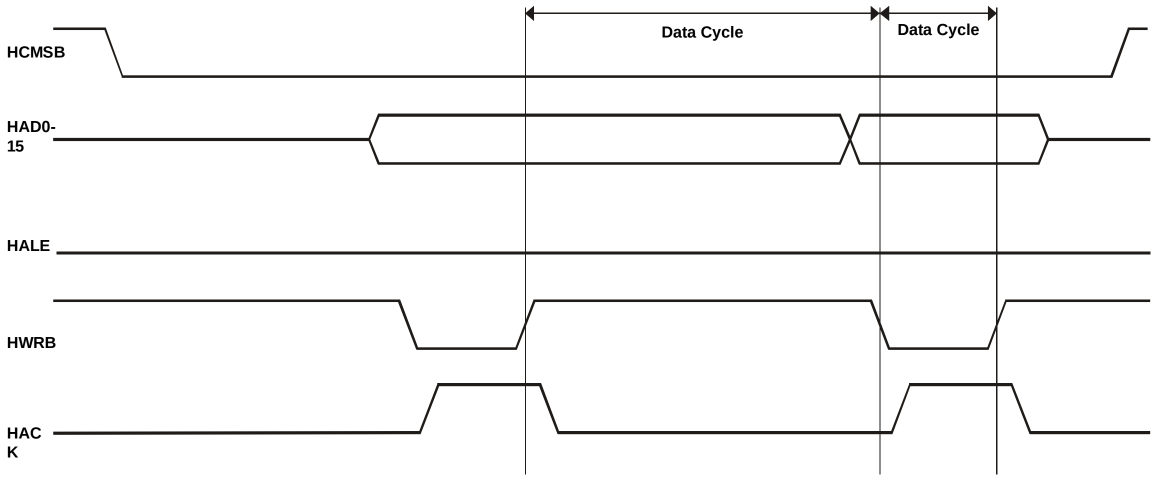


Figure 8-17. Host Port DMA Write

Interrupt Interface

If the interrupt on completion bit of the host port DMA descriptor is set and the DMA enable bit is set, the host port DMA controller generates an interrupt when the host port DMA word count register content transitions from one to zero. Correct initial programming of the word count registers is essential to assure that partial buffer contents are not allowed to corrupt subsequent DMA transfers.

If the interrupt on completion bit of the host port DMA descriptor is set and the DMA enable bit is cleared, the host port DMA controller generates an interrupt prior to shutting down.

If the interrupt on error bit of the host port DMA descriptor is set and the DMA operation completes with an error, the host port DMA controller generates an error interrupt prior to disabling the DMA engine and shutting down.

Host port initiated direct accesses do generate interrupts.

Setting Up the Host Port

This section describes a typical sequence that can be used by the host to set up the host port and transfer data through the interface. First, the ADSP-2191 is set to 16-bit mode using the configuration register. Then, 16- or 24-bit data is sent using the appropriate steps described later in this section.

Because changing configuration register values affects several parameters, the procedure below assumes that the `ACK`, byte order, endian-ness, and modes are returned to the state as when the ADSP-2191 is powered up. This procedure can be modified as appropriate.

To place the ADSP-2191 in 16-bit data mode (default at startup is 8-bit data mode):

1. Send two bytes to I/O register address `0x1C01` (=HPI `0x3802`). The first byte is `0x0041`; the second is `0x0000`.
2. 0x00x1 sets the host port bus width to 16-bits. 0x004x sets the acknowledge (`ACK`) mode to ready and clears byte endian-ness, data ordering, packet size, and pipelined reads to zero. The remaining bits can be read (reflecting the state of polarities at startup) but cannot be written.

To read or write to 16-bit space:

1. Send the I/O register 0x1C02 (=HPI `0x3804`).
2. Send the value `0x0000` to that register.
3. Send the memory address (word address as seen by the core*2).
4. Read or write the 16-bit value ABCD (where D0 is the right-most bit).

To read or write to 24-bit space:

1. Send the I/O register `0x1C02` (=HPI `0x3804`).
2. Send the value `0x0002` to that register.
3. Send the memory address (word address as seen by the core*4).
4. Read or write the 16-bit value EFxx (where D0 is the right-most bit).
5. Read or write the second 16-bit value ABCD.
6. The 24-bit value ABCDEF is the 24-bit value (where D0 is the right-most bit).

9 SERIAL PORTS (SPORTS)

This chapter provides the following sections:

Overview

This chapter describes the serial ports (SPORTs) available on the ADSP-2191.

In this text, the naming conventions for registers and pins use a lowercase `x` to represent a digit. For example, the name `DTx` indicates the `DT0`, `DT1`, and `DT2` pins (corresponding to SPORT0, SPORT1, or SPORT*2).*

The ADSP-2191 has three independent, synchronous serial ports (SPORT0, SPORT1, and SPORT2) that provide an I/O interface to a wide variety of peripheral serial devices. (SPORTs provide synchronous serial data transfer only; the ADSP-2191 provides asynchronous RS-232 data transfer via the UART.) Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. Each SPORT has one group of pins (data, clock, and frame sync) for transmission and a second set of pins for reception. Reception and transmission functions are programmed separately. SPORTs can be programmed for bit rate, frame sync, and bits per word by writing to registers in I/O space.

All three SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

SPORT2 shares I/O pins with the SPI interface (SPI0 and SPI1); the SPI interface and the SPORT2 serial port cannot be enabled at the same time.

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.

SPORTs can operate at up to 1/2 the full clock rate of `HCLK`, providing each with a maximum data rate of `CCLK`/2 Mbit/s in 1:1 (`CCLK`:`HCLK`) clock mode (where `CCLK` is the DSP core clock, and `HCLK` is the peripheral clock). Independent transmit and receive functions provide greater flexibility for serial communications. SPORT data can be transferred automatically to and from on-chip memory using DMA block transfers. Additionally, each SPORT offers a TDM (time division multiplexed) multichannel mode.

SPORT clocks and frame syncs can be internally generated by the DSP or received from an external source. The SPORTs can operate with little-endian or big-endian transmission formats, with word lengths selectable from three to 16 bits. They offer selectable transmit modes and optional m-law or A-law companding in hardware.

Each SPORT offers the following features and capabilities:

- Independent transmit and receive functions
- Serial data word transfers from three to sixteen bits in length, either MSB-first or LSB-first
- Data double-buffering (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT
- A-law and m-law hardware companding on transmitted and received words (see "Companding" on page 9-24 for more information)
- Internal generation of serial clock and frame sync signals—in a wide range of frequencies—or acceptance of clock and frame sync input from an external source
- Interrupt-driven, single-word transfers to and from on-chip memory under DSP core control
- Direct Memory Access transfer to and from memory under I/O processor 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).
- DMA transfers to and from on-chip memory—each SPORT can automatically receive and transmit an entire block of data
- Chaining of DMA operations for multiple data blocks

- Multichannel mode for TDM interfaces—each SPORT can receive and transmit data selectively from channels of a time-division-multiplexed serial bitstream multiplexed into up to 128 channels—this mode can serve as a network communication scheme for multiple processors
- Operation 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

Table 9-1 shows the pins for each SPORT.

Table 9-1. Serial Port (SPORT) Pins

Pin[1]	**Description**
DTx	Transmit Data
DRx	Receive Data
TCLKx	Transmit Clock
RCLKx	Receive Clock
TFSx	Transmit Frame Sync
RFSx	Receive Frame Sync

1 A lowercase x at the end of a pin name represents a possible value of 0, 1, or 2 (corresponding to SPORT0, SPORT1, or SPORT2).

A SPORT receives serial data on its `DR` input and transmits serial data on its `DT` output. It can receive and transmit simultaneously (full duplex operation). For both transmit and receive data, the data bits (`DR` or `DT`) are synchronous to the serial clocks (`RCLK` or `TCLK`); this is an output when the processor generates this clock or an input when the clock is externally-generated. Frame synchronization signals (`RFS` and `TFS`) indicate the start of a serial data word or stream of serial words.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur at the beginning of an individual word or at the beginning of a block of words.

Figure 9-1 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT's I/O space-mapped `SPx_TX` register via the peripheral bus. This data is optionally compressed by the hardware, then automatically transferred to the transmit shift register. The bits in the shift register are shifted out on the SPORT's `DT` pin, MSB first or LSB first, synchronous to the serial clock on the `TCLK` pin. The receive portion of the SPORT accepts data from the `DR` pin, synchronous to the serial clock. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's I/O space-mapped `SPx_RX` register, where it is available to the processor.

Figure 9-2 on page 9-7 shows the port connections for the SPORTs of the ADSP-2191.

SPORT Operation

This section provides an example of SPORT operation to illustrate the most common use of a SPORT. Since SPORT functionality is configurable, this example represents one of many possible configurations. See "Pin Descriptions" on page 14-2 for a table of all ADSP-2191 pins, including those used for the SPORTs.

Writing to a SPORT's `SPx_TX` register readies the SPORT for transmission. The `TFS` signal initiates the transmission of serial data. Once transmission has begun, each value written to the `SPx_TX` register is transferred to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB, as specified. Each bit is shifted out on the rising edge of `SCK`. After the first bit of a word has been trans-

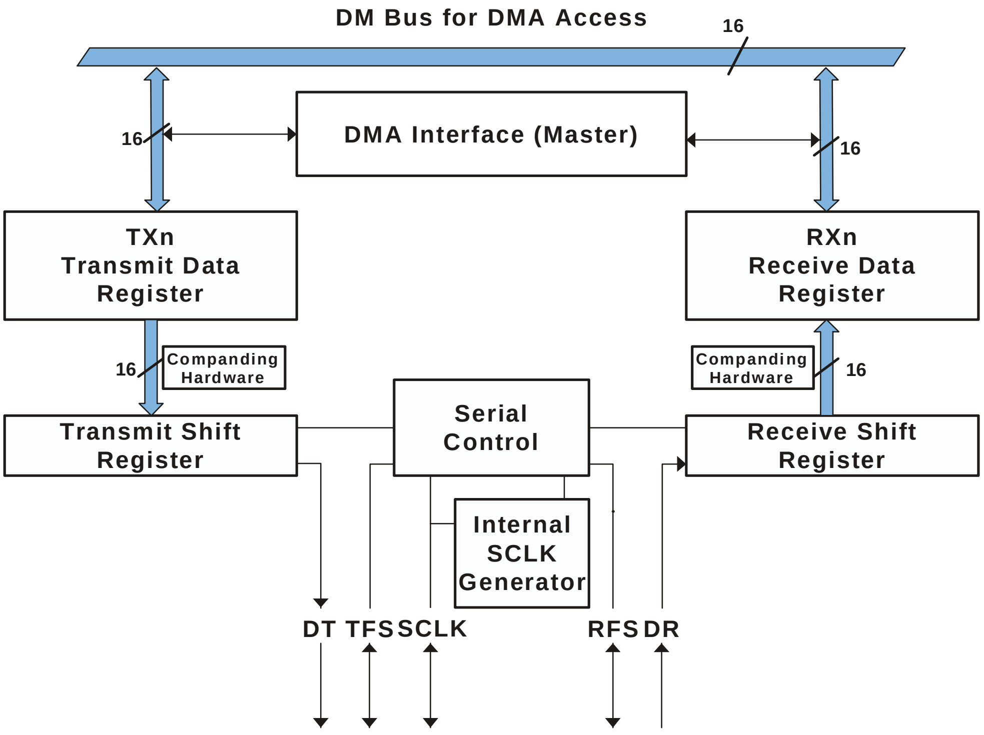


Figure 9-1. SPORT Block Diagram

ferred, the SPORT generates the transmit interrupt. The `SPx_TX` register is then available for the next data word, even though the transmission of the first word continues.

As a SPORT receives bits, the bits accumulate in an internal receive register. When a complete word has been received, it is written to the `SPx_RX` register, and the receive interrupt for that SPORT is generated. Interrupts are generated differently if DMA block transfers are performed; see "I/O Processor" on page 6-1 for general information about DMA and details on how to configure and use DMA with the SPORTs.

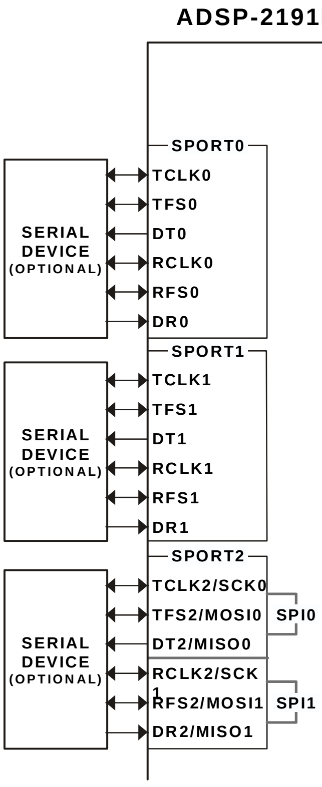


Figure 9-2. SPORT Connections

SPORT Disable

SPORTs are automatically disabled by a DSP hardware reset or software reset. A SPORT can also be disabled directly, by clearing the SPORT's transmit or receive enable bits (`TSPEN` in the `SPx_TCR` control register and `RSPEN` in the `SPx_RCR` control register). Each method affects the SPORT differently.

A DSP reset disables the SPORTs by clearing the SPx_TCR and SPx_RCR control registers (including the TSPEN and RSPEN enable bits) and the TDIVx, RDIVx, SPx_TFSDIVx, and SPx_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Disabling the TSPEN and RSPEN enable bits disables the SPORT(s) 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 (after the SPORT has been enabled), set the SPORT to receive an external clock.

SPORTs are ready to start transmitting or receiving data three SCK cycles after they are enabled (in the SPx_TCR or SPx_RCR control register). No serial clocks are lost from this point on.

Setting SPORT Modes

This section provides the following topics:

Overview

SPORT configuration is accomplished by setting bit and field values in configuration registers. Configure each SPORT prior to enabled it. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for the `xSCLKDIV` and `MxCS` registers, which can be modified while the SPORT is enabled).

To change values in all other SPORT configuration registers, disable the SPORT by clearing the `TSPEN` bit in `SPx_TCR` and/or the `RSPEN` bit in `SPx_RCR`.

Each SPORT has its own set of control registers and data buffers, as shown in Table 9-2. These control registers are described in detail in "ADSP-2191 DSP I/O Registers" on page B-1.

Table 9-2. SPORT Registers

Register Name	Function
SPx_TCR	SPORT Transmit Configuration Register
SPx_RCR	SPORT Receive Configuration Register
SPx_TX	Transmit Data Buffer
SPx_RX*	Receive Data Buffer
SPx_TSCKDIV	Transmit Clock Divide Modulus Register
SPx_RSCKDIV	Receive Clock Divide Modulus Register
SPx_TFSDIV	Transmit Frame Sync Divisor Register
SPx_RFSDIV	Receive Frame Sync Divisor Register
SPx_STATR*	SPORT Status Register
SPx_MTCS[0:7]	Multichannel Transmit Channel Select Registers
SPx_MRCS[0:7]	Multichannel Receive Channel Select Registers
SPx_MCMC1	Multichannel Mode Configuration Register 1
SPx_MCMC2	Multichannel Mode Configuration Register 2
SPxDR_PTR	DMA Current Pointer (receive)
SPxDR_CFG	DMA Configuration (receive)
SPxDR_SRP	DMA Start Page (receive)
SPxDR_SRA	DMA Start Address (receive)
SPxDR_CNT	DMA Count (receive)
SPxDR_CP	DMA Next Descriptor Pointer (receive)
An asterisk (*) indicates a read-only register. A lowercase x in a register name represents a possible value of 0, 1, or 2 (corresponding to SPORT0, SPORT1, or SPORT2).	

Table 9-2. SPORT Registers (Cont'd)

Register Name	Function
SPxDR_CPR	DMA Descriptor Ready (receive)
SPxDR_IRQ	DMA Interrupt Register (receive)
SPxDT_PTR	DMA Current Pointer (transmit)
SPxDT_CFG	DMA Configuration (transmit)
SPxDT_SRP	DMA Start Page (transmit)
SPxDT_SRA	DMA Start Address (transmit)
SPxDT_CNT	DMA Count (transmit)
SPxDT_CP	DMA Next Descriptor Pointer (transmit)
SPxDT_CPR	DMA Descriptor Ready (transmit)
SPxDT_IRQ	DMA Interrupt Register (transmit)
An asterisk (*) indicates a read-only register. A lowercase x in a register name represents a possible value of 0, 1, or 2 (corresponding to SPORT0, SPORT1, or SPORT2).	

SPORT control registers are programmed by writing to the appropriate address in memory. Symbolic names of the registers and individual control bits can be used in DSP programs—the `#define` definitions for these symbols are contained in the `def2191.h` file, which is provided in the `INCLUDE` directory of the ADSP-2191 DSP development software. The `def2191.h` file is shown in "Register and Bit #define File (def2191.h)" on page B-115. All control and status bits in the SPORT registers are active high unless otherwise noted.

Because the SPORT control registers are I/O-mapped, programs read or write them using the `IO( )` register read/write instructions. The SPORT control registers can be written or read by external devices (for example, a host processor) to set up a SPORT DMA operation.

Except for `TCLKDIV`/`SCLKDIV` registers and multichannel configuration registers, most configuration registers can be changed only while the SPORT is disabled (`TSPEN`/`RSPEN`=0). Changes take effect after the SPORT is re-enabled.

Transmit Configuration (SPx_TCR) Register and Receive Configuration (SPx_RCR) Register

The main control registers for each SPORT are the transmit configuration register (`SPx_TCR`) and the receive configuration register (`SPx_RCR`), which are defined in Figure B-10 on page B-39 and Figure B-11 on page B-40.

A SPORT is enabled for transmit when bit 0 (`TSPEN`) of the transmit configuration register is set to 1; it is enabled to receive when bit 0 (`RSPEN`) of the receive configuration register is set to 1. Both of these bits are cleared at reset (during a hard reset or a soft reset), disabling all SPORT channels.

When the SPORT is enabled to transmit (`TSPEN` set) or receive (`RSPEN` set), corresponding SPORT configuration register writes are disabled (except for `SPx_RSCKDIV`, `SPx_TSCKDIV`, and multichannel mode channel enable registers). Writes are always enabled to the `SPx_TX` buffer. `SPx_RX` is a read-only register.

After a write to a SPORT register, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

When changing operating modes, clear a SPORT control register (written with all zeros) before the new mode is written to the register.

The `TXS` status bit in the SPORT status register indicates whether the `SPx_TX` buffer is full (`1`) or empty (`0`).

The transmit underflow status bit (TUVF) in the SPORT status register is set whenever the TFS signal occurs (from an external or internal source) while the SPx_TX buffer is empty. The internally-generated TFS may be suppressed whenever SPx_TX is empty by clearing the DITFS control bit in the SPORT Configuration Register (DITFS=0).

When DITFS=0 (the default), the internal transmit frame sync signal (TFS) is dependent upon new data being present in the SPx_TX buffer; the TFS signal is generated for new data only. Setting DITFS to 1 selects data-independent frame syncs and causes the TFS signal to be generated regardless whether new data is present, transmitting the contents of the SPx_TX buffer regardless. SPORT DMA typically keeps the SPx_TX buffer full, and when the DMA operation is complete, the last word in SPx_TX is continuously transmitted.

The SPx_TCR and SPx_RCR transmit and receive configuration registers control the SPORTs' operating modes for the I/O processor. Figure B-11 on page B-40 lists the bits in SPx_RCR, and Figure B-10 on page B-39 lists the bits in SPx_TCR.

The following bits control SPORT modes. See "Setting Peripheral DMA Modes" on page 6-17 for information about configuring DMA with SPORTs.

Bits for the SPI Transmit Configuration (SPx_TCR) register:

- Transmit Enable. SPx_TCR Bit 0 (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set, =1) or disabled (if cleared, =0).

  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 TD write code in the TX interrupt service routine. For this reason, the code should initialize the interrupt service routine (ISR) and be ready to service TX interrupts before setting TSPEN.

Clearing TSPEN causes the SPORT to stop driving data 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.

- Internal Transmit Clock Select. SPx_TCR Bit 1 (ICLK). This bit selects the internal transmit clock (if set, =1) or the external transmit clock on the TCLK or RCLK pin (if cleared, =0).
- Data Formatting Type Select. SPx_TCR Bits 3-2 (DTYPE). The DTYPE, SENDN, and SLEN bits configure the formats of the data words transmitted over the SPORTs. The two DTYPE bits specify one of four data formats (00=right-justify and zero-fill unused MSBs, 01=right-justify and sign-extend into unused MSBs, 10=compand using m-law, 11=compand using A-law) used for single- and multichannel operation.
- Endian Format Select. SPx_TCR Bit 4 (SENDN). The DTYPE, SENDN, and SLEN bits configure the formats of the data words transmitted over the SPORTs. The SENDN bit selects the endian format (0=serial words are transmitted MSB bit first, 1=serial words are transmitted LSB bit first).
- Serial Word Length Select. SPx_TCR Bits 8-5 (SLEN). The DTYPE, SENDN, and SLEN bits configure the formats of the data words transmitted over the SPORTs (shifted out via the TXDATA pin). Calculate serial word length (number of bits in each word transmitted over the SPORTs) by adding 1 to the value of SLEN:

  ```
  Serial Word = SLEN + 1;
  ```

  SLEN can be set to a value of 2 to 15; 0 and 1 are illegal values bit. Two common settings for SLEN are 15 (transmit a full 16-bit word) and 7 (transmit an 8-bit byte). The ADSP-2191 is a 16-bit DSP, so program instruction or DMA engine loads of the TX data register always move 16 bits into the register; the SLEN bits informs the SPORT how many of those 16 bits to shift out of the register over the serial link.

The frame sync signal is controlled by the frame sync divider registers, not by SLEN. To produce a frame sync pulse on each byte or transmitted word, the proper frame sync divider must be programmed into the proper frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- Internal Transmit Frame Sync Select. SPx_TCR Bit 9 (ITFS). This bit selects whether the SPORT uses an internal TFS (if set, =1) or an external TFS (if cleared, =0).
- Transmit Frame Sync Required Select. SPx_TCR Bit 10 (TFSR). This bit selects whether the SPORT requires (if set, =1) or does not require (if cleared, =0) a transfer frame sync for every data word.

  The TFSR bit is normally set (=1). A frame sync pulse marks the beginning of each word or data packet. Most systems need frame sync to function properly.
- Data Independent Transmit Frame Sync Select. SPx_TCR Bit 11 (DITFS). This bit selects whether the SPORT uses a data-independent TFS (sync at selected interval, if set, =1) or a data-dependent TFS (sync when data in SPx_TX, if cleared, =0).

  The frame sync pulse marks the beginning of the data word. If DITFS is set (=1), the frame sync pulse is issued on time, regardless whether the TX register has been loaded; if DITFS is cleared (=0), the frame sync pulse is generated only when the TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set (=1), and the DSP should keep loading the TDATA register on time. If the receiver will tolerate occasional late frame sync pulses, clear DITFS (=0) to prevent the SPORT from transmitting old data twice or transmitting garbled data if the DSP is late in loading the TX register.

- Low Transmit Frame Sync Select. SPx_TCR Bit 12 (LTFS). This bit selects an active low TFS (if set, =1) or active high TFS (if cleared, =0).
- Late Transmit Frame Sync. SPx_TCR Bit 13 (LATFS). This bit configures late frame syncs (if set, =1) or early frame syncs (if cleared, =0).
- Clock Rising Edge Select. SPx_TCR Bit 14 (CKRE). This bit selects whether the SPORT uses the rising edge (if cleared, =0) or falling edge (if set, =1) of the TCLK clock signal for sampling data and the frame sync.
- Internal Clock Disable. SPx_TCR Bit 15. This bit, when set (=1), disables the TCLK clock. By default, this bit is cleared (=0), enabling TCLK.

Bits for the Receive Configuration (SPx_RCR) register:

- Receive Enable. SPx_RCRBit 0 (RSPEN). This bit selects whether the SPORT is enabled to receive (if set, =1) or is disabled (if cleared, =0).

  Setting the RSPEN bit turns on the SPORT, causing it to drive the DRx pin (and the RX bit clock and receive frame sync pins if so programmed). Program all SPORT control registers before RSPEN is set. Typical SPORT initialization code first writes SPX_RCR with everything except TSPEN; the last step in the code rewrites SPX_RCR with all of the necessary bits including RSPEN.

Setting RSPEN enables the SPORT RX interrupt. For this reason, the code should initialize the interrupt service routine and be ready to service RX interrupts before setting RSPEN.

Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT circuitry. In low-power applications, extend battery life by clearing RSPEN whenever the SPORT is not in use.

- Internal Receive Clock Select. SPx_RCR Bit 1 (ICLK). This bit selects the internal receive clock (if set, =1) or external receive clock (if cleared, =0).

- Data Formatting Type Select. SPx_RCR Bits 3-2 (DTYPE). The DTYPE, SENDN, and SLEN bits configure the formats of the data words received over the SPORTs. The two DTYPE bits specify one of four data formats (00=right-justify and zero-fill unused MSBs, 01=right-justify and sign-extend into unused MSBs, 10=compand using m-law, 11=compand using A-law) to be used for single- and multichannel operation.

- Endian Format Select. SPx_RCR Bit 4 (SENDN). The DTYPE, SENDN, and SLEN bits configure the formats of the data words received over the SPORTs. The SENDN bit selects the endian format (0=serial words are received MSB bit first, 1=serial words are received LSB bit first).

- Serial Word Length Select. SPx_RCR Bit 8-5 (SLEN). The DTYPE, SENDN, and SLEN bits configure the formats of the data words received over the SPORTs. Calculate the serial word length (the number of bits in each word received over the SPORTs) by adding "1" to the value of the SLEN bit. The SLEN bit can be set to a value of 2 to 15; note that 0 and 1 are illegal values for this bit.

- Internal Receive Frame Sync Select. SPx_RCR Bit 9 (IRFS). This bit selects whether the SPORT uses an internal RFS (if set, =1) or an external RFS (if cleared, =0).
- Receive Frame Sync Required Select. SPx_RCR Bit 10 (RFSR). This bit selects whether the SPORT requires (if set, =1) or does not require (if cleared, =0) a receive frame sync for every data word.
- Low Receive Frame Sync Select. SPx_RCR Bit 12 (LRFS). This bit selects an active low RFS (if set, =1) or active high RFS (if cleared, =0).
- Late Receive Frame Sync. SPx_RCR Bit 13 (LARFS). This bit configures late frame syncs (if set, =1) or early frame syncs (if cleared, =0).
- Clock Rising Edge Select. SPx_RCR Bit 14 (CKRE). This bit selects whether the SPORT uses the rising edge (if set, =1) or falling edge (if cleared, =0) of the RCLK clock signal for sampling data and the frame sync.
- Internal Clock Disable. SPx_RCR Bit 15 (ICLKD). This bit, when set (=1), disables the RCLK clock. By default this bit is cleared (=0), enabling RCLK.

Register Writes and Effect Latency

When the SPORT is disabled (TSPEN and RSPEN are cleared), SPORT register writes are internally completed at the end of the next CLKIN cycle after which they occurred, and the register reads back the newly written value on the next cycle after that.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPx_RSCKDIV, SPx_TSCKDIV, and multichannel mode channel registers). SPx_TX register writes are always enabled; SPx_RX is a read-only register.

After a write to a SPORT register, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Transmit (SPx_TX) Data Buffer and Receive Data Buffer (SPx_RX)

SPx_TX is the transmit data buffer for the SPORT. It is a 16-bit buffer which must be loaded with the data to be transmitted. The data is loaded by the DMA controller or by the program running on the DSP core. SPx_RX is the receive data buffer for the SPORT. It is a 16-bit buffer that is automatically loaded from the receive shifter when a complete word has been received. Word lengths of less than 16 bits are right-justified in the receive and transmit buffers.

The SPx_TX buffers act like a two-location FIFO because they have a data register plus an output shift register as shown in Figure 9-1 on page 9-4. Two 16-bit words may be stored in the TX buffers at any one time. When the SPx_TX buffer is loaded and a previous word has been transmitted, the buffer contents are automatically loaded into the output shifter. An interrupt is generated when the output shifter has been loaded, signifying that the SPx_TX buffer is ready to accept the next word (the SPx_TX buffer is "not full"). This interrupt does not occur if SPORT DMA is enabled.

The transmit underflow status bit (TUVF) is set in the SPORT status register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode, TUVF is set whenever the serial shift register is not loaded, when that transmission should begin on an enabled channel. The TUVF status bit is "sticky" and is cleared only by disabling the SPORT.

The SPx_RX buffers act like a two-location FIFO because they have a data register plus an input shift register. Two 16-bit words can be stored in the SPx_RX buffer. The third word overwrites the second word when the first word has not been read out (by the DSP core or the DMA controller). Should this happen, the receive overflow status bit (ROVF) is set in the

SPORT status register. The overflow status is generated on the last bit of the second word. The ROVF status bit is "sticky" and is cleared only by disabling the SPORT.

An interrupt is generated when the SPx_RX buffer has been loaded with a received word (the SPx_RX buffer is "not empty"). This interrupt is masked out if SPORT DMA is enabled.

If the program causes the core processor to attempt a read from an empty SPx_RX buffer, any old data is read. If the program causes the core processor to attempt a write to a full SPx_TX buffer, the new data overwrites the SPx_TX register. If it is not known whether the core processor can access the SPx_RX or SPx_TX buffer without causing such an error, read the buffer's full or empty status first (in the SPORT status register) to determine whether the access can be made.

The RXS and TXS status bits in the SPORT status register are updated upon reads and writes from the core processor, even when the SPORT is disabled. The SPx_RX register is read-only. The SPx_TX register can be read regardless whether the SPORT is enabled.

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for an internal source or an external source) is HCLK/2. The frequency of an internally-generated clock is a function of the processor clock frequency (as seen at the CLKOUT pin) and the value of the 16-bit serial clock divide modulus registers, SPx_TSCKDIV and SPx_RSCKDIV.

$$\text{SPx\_TCLK/SPx\_RCLK frequency} = \frac{\text{HCLK frequency}}{2 \times (\text{SPx\_TSCKDIV/SPxRSCKDIV} + 1)}$$

If the value of SPx_TRSKDIV/SPx_RSCKDIV is changed while the internal serial clock is enabled, the change in TCLK/RCLK frequency takes effect at the start of the rising edge of TCLK/RCLK following the next leading edge of TFS/RFS.

The SPx_TFSDIV and SPx_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to internally generated or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

```
# of serial clocks between frame sync assertions = xFSDIV + 1
```

Use the following equation to determine the correct value of xFSDIV, given the serial clock frequency and desired frame sync frequency:

$$\text{SPx\_TFSCLK/SPx\_RFSCLK frequency} = \frac{\text{SCLK frequency}}{(\text{SPx\_RFSDIV / SPx\_TFSDIV} + 1)}$$

The frame sync would thus be continuously active if xFSDIV=0. However, the value of xFSDIV should not be less than the serial word length minus one (the value of the SLEN field in the transmit or receive control register); a smaller value of xFSDIV could cause an external device to abort the current operation or have other unpredictable results. If the SPORT is not being used, the xFSDIV 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. This can also limit the maximum serial clock speed. See the *ADSP-2191 DSP Microcomputer Data Sheet* for timing specifications.

Be careful when operating with externally generated clocks near the frequency of HCLK/2. A delay between the clock signal's arrival at the TCLK pin and the output of the data may limit the receiver's speed of operation. See the *ADSP-2191 DSP Microcomputer Data Sheet* for exact timing specifications. At full speed serial clock rate, the safest practice is to use an externally generated clock and externally generated frame sync (ICLK=0 and IRFS=0).

Frame Sync and Clock Example

The following code fragment is a brief example of setting up the clocks and frame sync.

```
/* Set SPORT0 frame sync divisor */
AR = 0x00FF;
IO(SP0_RFSDIV) = AR;
IO(SP0_TFSDIV) = AR;

/* Set SPORT0 Internal Clock Divider */
AR = 0x0002;
IO(SP0_RSCLKDIV) = AR;
```

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the DTYPE, SENDN, and SLEN bits of the SPx_TCR and SPx_RCR transmit and receive configuration registers.

Word Length

Each SPORT channel (transmit and receive) independently handles words with lengths of three to 16 bits. The data is right-justified in the SPORT data registers if it is fewer than 16 bits long, residing in the LSB positions. The value of the serial word length (`SLEN`) field in the `SPx_TCR` and `SPx_RCR` registers of each SPORT determines the word length according to this formula:

Serial Word Length = SLEN + 1

⊘ Do not set `SLEN` to zero or one; values from 2 to 15 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 four or longer (so `SLEN >= 3`).

Endian Format

Endian format determines whether the serial word is transmitted most significant bit (MSB) first or least significant bit (LSB) first. Endian format is selected by the `SENDN` bit in the `SPx_TCR` and `SPx_RCR` transmit and receive configuration registers. When `SENDN=0`, serial words are transmitted (or received) MSB-first. When `SENDN=1`, serial words are transmitted (or received) LSB-first.

Data Type

The `DTYPE` field of the `SPx_TCR` and `SPx_RCR` transmit and receive configuration registers specifies one of the four data formats for both single and multichannel operation, as shown in the following table:

Table 9-3. DTYPE and Data Formatting

DTYPE	Data Formatting
00	Right-justify, zero-fill unused MSBs
01	Right-justify, sign-extend into unused MSBs

Table 9-3. DTYPE and Data Formatting (Cont'd)

DTYPE	Data Formatting
10	Compand using m-law
11	Compand using A-law

These formats are applied to serial data words loaded into the SPx_RX and SPx_TX buffers. SPx_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 ADSP-2191 SPORTs support the two most widely used companding algorithms: A-law and m-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 SPx_RX register is the right-justified, expanded value of the eight LSBs received and sign-extended. A write to SPx_TX compresses the 16-bit value to eight LSBs (sign-extended to the width of the transmit word) and written to the internal transmit register. If the magnitude of the 16-bit value is greater than the 13-bit A-law or 14-bit m-law maximum, the value is automatically compressed to the maximum positive or negative value.

Clock Signal Options

Each SPORT has a transmit clock signal (TCLK) and a receive clock signal (RCLK). The clock signals are configured by the ICLK and CKRE bits of the SPx_TCR and SPx_RCR transmit and receive configuration registers. Serial clock frequency is configured in the SPx_TSCKDIV and SPx_RSCKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both input and output.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ICLK bit of the SPx_TCR and SPx_RCR configuration registers specifies the clock source.

When ICLK=1, the clock signal is generated internally by the DSP and the TCLK or RCLK pin is an output; the clock frequency is determined by the value of the serial clock divisor in the SPx_TSCKDIV or SPx_RSCKDIV registers.

When ICLK=0, the clock signal is accepted as an input on the TCLK or RCLK pins and the serial clock divisors in the SPx_TSCKDIV/SPx_RSCKDIV registers are ignored; the externally generated serial clock need not be synchronous with the DSP system clock.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame synchronization) and RFS (receive frame synchronization). A variety of framing options are available; these options are configured in the SPORT control registers. The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed vs. Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (transmit frame sync required) and RFSR (receive frame sync required) control bits determine whether frame sync signals are required. These bits are located in the SPx_TCR and SPx_RCR transmit and receive configuration 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 SPx_TX buffer before the previous word is shifted out and transmitted. For more information, see "Data-Independent Transmit Frame Sync" on page 9-30.

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.

When DMA is enabled in this mode, with frame syncs not required, DMA requests may be held off by chaining or may not be serviced frequently enough to guarantee continuous unframed data flow.

Figure 9-2 on page 9-7 illustrates framed serial transfers, which have the following characteristics:

- TFSR and RFSR bits in the SPx_TCR and SPx_RCR transmit and receive configuration 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 SPx_TCR and SPx_RCR configuration registers.

See "Timing Examples" on page 9-47 for more timing examples.

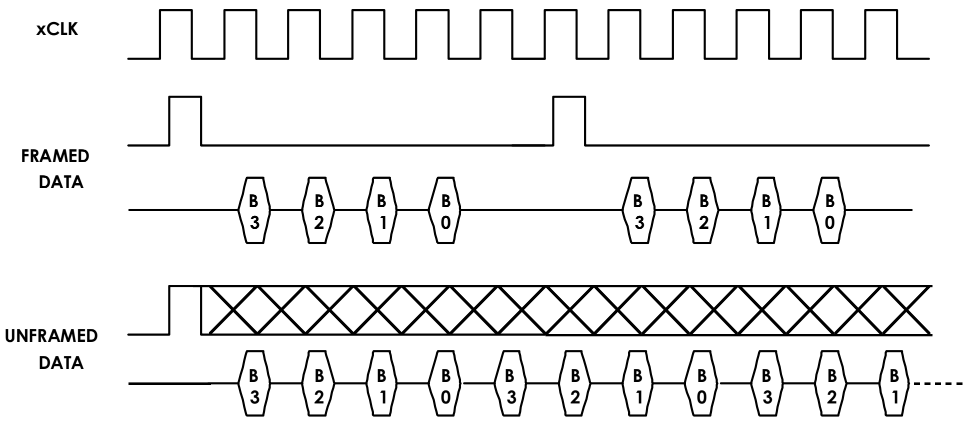


Figure 9-3. Framed vs. Unframed Data

Internal vs. External Frame Syncs

Transmit and receive frame syncs can be independently generated internally or input from an external source as determined by the ITFS and IRFS bits of the SPx_TCR and SPx_RCR transmit and receive configuration registers.

When ITFS=1 or IRFS=1, the corresponding frame sync signal is generated internally by the SPORT, and the TFS pin or RFS pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the SPx_TFSDIV or SPx_RFSDIV registers.

When ITFS=0 or IRFS=0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pins, and the frame sync divisors in the SPx_TFSDIV/SPx_RFSDIV registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low vs. Active High Frame Syncs

Frame sync signals may be active high or active low (inverted). The `LTFS` and `LRFS` bits of the transmit (`SPx_TCR`) and receive (`SPx_RCR`) configuration registers determine the frame syncs' logic level, as follows:

- 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 the rising or falling edges of the SPORT clock signals. The `CKRE` bit of the `SPx_TCR` and `SPx_RCR` transmit and receive configuration registers selects the sampling edge of the serial data. Setting `CKRE=0` in the `SPx_TCR` transmit configuration register selects the rising edge of `TCLKx`. `CKRE=1` selects the falling edge.

Data and frame sync signals change state on the clock edge not selected. For example, for data to be sampled on the rising edge of a clock, it must be transmitted on the falling edge of the clock.

For receive data and frame syncs, setting `CKRE=1` in the `SPx_RCR` receive configuration register selects the rising edge of `RCLK` as the sampling point for the transmission. `CKRE=0` selects the falling edge.

For transmit data and frame syncs, setting CKRE=1 in the SPx_TCR transmit configuration register selects the falling edge of the TCLK for the transmission (so the rising edge of TCLK can be used as the sampling edge by the receiver). CKRE=0 selects the rising edge for the transmission.

The transmit and receive functions of two SPORTs connected together, for example, should always select the same value for CKRE so any internally generated signals are driven on one edge and any received signals are sampled on the opposite edge.

Early vs. Late Frame Syncs (Normal and 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 transmit (SPx_TCR) and receive (SPx_RCR) configuration 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 is the case when frame delay is 1.

If data transmission is continuous in early framing mode (the last bit of each word is immediately followed by the first bit of the next word), 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 four of longer (so 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 zero.) Receive data bits are sampled by serial clock edges, but the frame sync signal is checked only 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 only are checked during the first bit.

Figure 9-2 on page 9-7 illustrates the two modes of frame signal timing:

- LATFS or LARFS bits of the SPx_TCR and SPx_RCR transmit and receive configuration 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 first bit only.
- Data transmitted MSB-first (SENDN=0) or LSB-first (SENDN=1).
- Frame sync and clock generated internally or externally.

See "Timing Examples" on page 9-47 for more timing examples.

Data-Independent Transmit Frame Sync

Normally, the internally generated transmit frame sync signal (TFS) is output only when the SPx_TX buffer has data ready to transmit. The DITFS mode (data-independent transmit frame sync) bit allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPx_TCR transmit configuration register configures this option.

When DITFS=0, the internally-generated TFS is output only when a new data word has been loaded into the SPx_TX buffer. Once data is loaded into SPx_TX, the next TFS is generated. This mode allows data to be transmitted only when it is available.

When DITFS=1, the internally generated TFS is output at its programmed interval regardless whether new data is available in the SPx_TX buffer. Whatever data is present in SPx_TX is re-transmitted with each assertion of

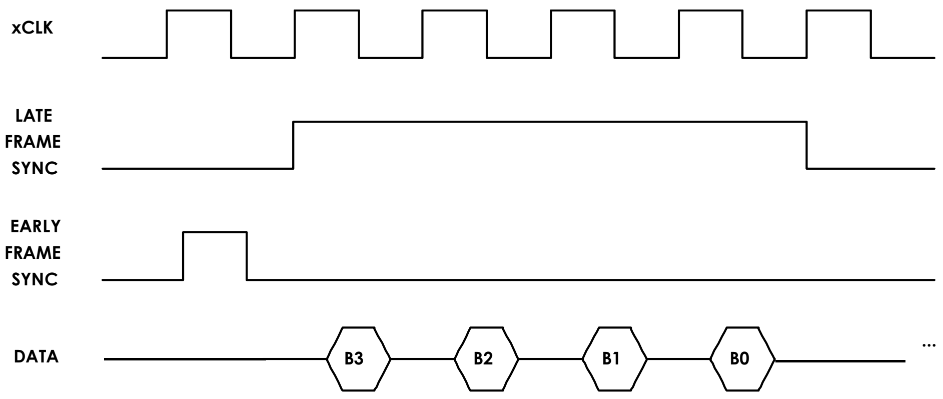


Figure 9-4. Normal vs. Alternate Framing

TFS. The TUVF transmit underflow status bit (in the SPxSTATR status register) is set when this occurs and old data is retransmitted. The TUVF status bit is also set when the SPx_TX buffer does not have new data if an externally generated TFS occurs. In this mode, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPx_TX data register is required to start the transfer.

Multichannel Operation

This section provides the following topics:

Overview

SPORTs offer a multichannel mode of operation, which allows a SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bit stream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmission or reception; each SPORT can receive and transmit data selectively from any of the 128 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 DT pin is always driven (not three-stated) if the SPORT is enabled (TSPEN=1 in the SPx_TCR transmit configuration register), unless it is in multichannel mode and an inactive time slot occurs.

In multichannel mode, the TCLK pin is an input and must be connected to its corresponding RCLK pin. RCLK can be provided externally or generated internally by the SPORT.

🛇 The MCM channel select registers must be programmed before enabling SPx_TX/SPx_RX operation. This is especially important in DMA data unpacked mode, since SPORT FIFO operation begins immediately after SPx_TX/SPx_RX is enabled and depends on the values of the MCM channel select registers. Enable MCM_EN prior to enabling SPx_TX and/or SPx_RX operation.

Figure 9-1 on page 9-6 shows example timing for a multichannel transfer, which has the following characteristics:

- Uses TDM method where serial data is sent or received on different channels sharing the same serial bus.
- Can independently select transmit and receive channels.
- RFS signals start of frame.
- TFS is used as "transmit data valid" for external logic, true only during transmit channels.
- Example: Receive on channels 0 and 2. Transmit on channels 1 and 2.

See "Timing Examples" on page 9-47 for more timing examples.

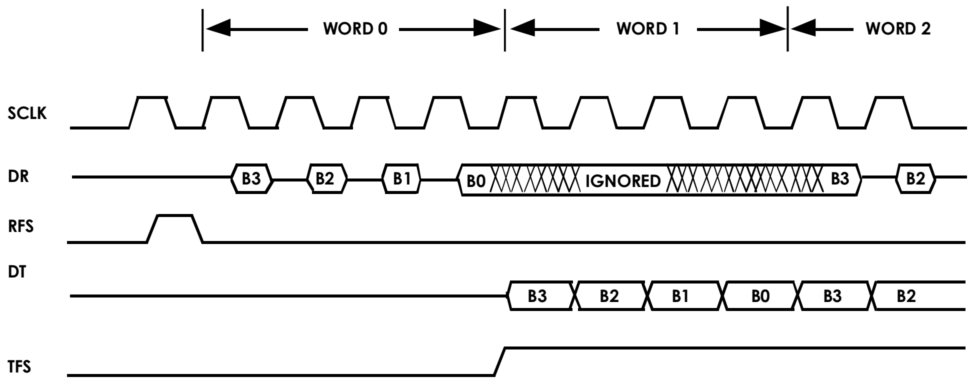


Figure 9-5. Multichannel Operation

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block (or frame) of multichannel data words.

When multichannel mode is enabled on a SPORT, the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal synchronizes the channels and restarts each multichannel sequence. RFS assertion occurs at the beginning of the channel 0 data word.

Since RFS is used by both the SPx_TX and SPx_RX channels of the SPORT in MCM configuration, both SPx_RX configuration registers should always be programmed the same way as the SPx_TX configuration register, even if SPx_RX operation is not enabled.

In multichannel mode, 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).

TFS serves as a transmit data valid signal, which is active during transmission of an enabled word. The SPORT's DT pin is three-stated when the time slot is not active, and the TFS signal serves as an output enabled signal for the DT pin. The SPORT drives TFS in multichannel mode regardless whether ITFS is cleared.

Once the initial FS is received and a frame transfer has started, the SPORT ignores all other FS signals until the complete frame has been transferred.

In multichannel mode, the RFS signal is used for the block (frame) start reference, after which the transfers are performed continuously with no FS required. Thus, internally generated frame syncs are data-independent.

Multichannel Frame Delay

The 4-bit MFD field in the SPxDR_CFG register 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 zero 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

Window size defines the range of the channels that can be enabled/disabled in the current configuration. It can be any value (from 8 to 128) in increments of 8); the default value (0) corresponds to 8 channels. Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 bits), resulting in a smaller DMA buffer size (in this example, 32 bits instead of 128 bits) to save DMA bandwidth. Window size cannot be changed while the SPORT is enabled.

Window Offset

The window offset specifies where (in the 127 channel range) to place the start of the window. 0 specifies no offset and permits the use of all 128 channels. For example, a program can define a window size of 5 and an offset of 93; this 5-channel window would reside in the range from 93 to 97. Window offset cannot be changed while the SPORT is enabled.

If the combination of the window size and the window offset places the window outside of the range of the channel enable registers, none of the channels in the frame are enabled, since this combination is invalid.

Other Multichannel Fields in SPx_TCR and SPx_RCR

A multichannel frame contains more than one channel, as specified by the window size and window offset; the multichannel frame is a combined sequence of the window offset and the channels contained in the window. The total number of channels in the frame is calculated by adding the window size to the window offset.

The channel select offset mode is bit 4 in the MCM configuration register 2. When this mode is selected, the first bit of the SPx_MTCSx register or SPx_MRCSx register is linked to the first bit directly following the offset of the window. If the channel select offset mode is not enabled, the first bit of the SPx_MTCSx or SPx_MRCSx register is placed at offset 0.

The 7-bit CHNL field in the SPx_STATR status register indicates the channel currently selected during multichannel operation. This field is a read-only status indicator. CHNL(6:0) increments by one as each channel is serviced, and in channel select offset mode the value of CHNL is reset to 0 after the offset has been completed. For example, for a window of 8 and an offset of 21, the counter displays a value between 0 and 28 in the regular mode, but in channel select offset mode the counter resets to 0 after counting up to 21 and the frame completes when the CHNL reaches 7 (indicating the eighth channel).

The FSDR bit in the MCM configuration register 2 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 (FSDR=0) the data is transmitted on the same edge that the TFS is generated. For example, a positive edge TFS transmits data on the positive edge of the SCK. This is either the same edge of the following one, depending on when LATFS is set.

When frame synch/data relationship is used (FSDR=1), the frame synch 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 Registers

A channel is a multi-bit word (from 3 to 16 bits in length) that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select the words to be received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, and disabled channel words are ignored. Up to 128 channels are available. The SPx_MRCSx and SPx_MTCSx multichannel selection registers enable and disable individual channels; the Multichannel Receive Channel Select (SPx_MRCSx) registers specify the active receive channels, and the Multichannel Transmit Channel Select (SPx_MTCSx) registers specify the active transmit channels.

Each register has 16 bits, corresponding to the 16 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). For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit to 1 in the SPx_MTCSx register causes the SPORT to transmit the word in that channel's position of the data stream. Clearing the bit to 0 in the SPx_MTCSx register three-states the SPORT's data transmit (DT) pin during the time slot of that channel.

Setting a particular bit to 1 in the SPx_MRCSx register causes the SPORT to receive the word in that channel's position of the data stream; the received word is loaded into the SPx_RX buffer. Clearing the bit to 0 in the SPx_MRCSx register causes the SPORT to ignore the data.

Companding is selected on an all-or-none channel basis. A-law or m-law companding is selected with the DTYPE bit 1 in the SPx_TCR and SPx_RCR transmit and receive configuration registers, and applies to all active channels. (See "Companding" on page 9-24 for information about companding.)

Multichannel Enable

Setting the MCM bit in the multichannel mode configuration control register 1 enables multichannel mode. When MCM=1, multichannel operation is enabled; when MCM=0, all multichannel operations are disabled.

Setting the MCM bit enables multichannel operation for the receive and transmit sides of the SPORT. If a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

Multichannel DMA Data Packing

Multichannel DMA data packing/unpacking are specified with the DMA data packed/unpacked enable bits for the SPx_RX and SPx_TX multichannel configuration registers.

If the bits are set (indicating that data is packed), the SPORT expects that 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 of the frames). 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 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/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 has no restrictions on changing the number of enabled channels while the SPORT is enabled.

Multichannel TX FIFO Prefetch

The TX and RX channels have 8-word DMA FIFOs. Normally, the DMA engine is capable of pre-fetching the TX words for up to seven channels ahead of the one currently being transmitted.

In TX unpacked mode, matters are more complex since the disabled channels are ignored and the DMA data for these channels is skipped (not placed into the TX FIFO). This leaves room for the prefetch logic to fill the FIFO with words corresponding to the channels farther ahead of the current one.

For example, suppose there is a window of 32 with no offset, with only channels 5 and 25 enabled, TX unpacked. While the SPORT is preparing to transmit channel 5, pre-fetch logic is capable of loading the data for channel 25 of the current frame, plus for channels 5 and 25 of the next three frames. The problem with this is that one of the goals of the unpacked mode is to allow on-the-fly reconfiguration of the TX channel selects, so that while channel 5 is being transmitted, the program can, for example, enable channels 27-32. In the described scenario, this would have been dangerous since pre-fetch logic could have been far past channels 27-32 of the current frame, and would fail to load the data for these newly enabled registers into the FIFO.

To restrict this undesirable behavior, a 2-bit control field is introduced (`MCFF`, the transmit FIFO pre-fetch max distance). `MCFF` can be programmed to restrict pre-fetch logic from fetching the data farther away than the specified value ZZ (where ZZ can be 2, 4, 8, or 16 channels away from the one currently being transmitted).

Multichannel Mode Example

The following code fragment is an example of setting up multichannel mode for SPORT use.

```
/* Set MCM Transmit and Receive Channel Selection Reg */
AR = 0x001F;      /* Enable Channels 0-4 for Tx */
IO(SP0_MTCS0) = AR;
AR = 0x0000;       /* ... Disable remaining 123-Channels */
IO(SP0_MTCS1) = AR;
IO(SP0_MTCS2) = AR;
IO(SP0_MTCS3) = AR;
IO(SP0_MTCS4) = AR;
IO(SP0_MTCS5) = AR;
IO(SP0_MTCS6) = AR;
IO(SP0_MTCS7) = AR;

AR = 0x001F;     /* Enable Channels 0-4 for Rx */
IO(SP0_MRCS0) = AR;
AR = 0x0000;    /* ... Disable remaining 123-Channels */
IO(SP0_MRCS1) = AR;
IO(SP0_MRCS2) = AR;
IO(SP0_MRCS3) = AR;
IO(SP0_MRCS4) = AR;
IO(SP0_MRCS5) = AR;
IO(SP0_MRCS6) = AR;
IO(SP0_MRCS7) = AR;

               /* Set SPORT0 MCM Configuration Reg 1 */
               /* MCM enabled, 1 Frame Delay */
AR = 0x0003;
IO(SP0_MCMC1) = AR;

               /* Set SPORT0 MCM Configuration Reg 2 */
               /* Tx and Rx Packing */
```

```
AR = 0x000C;
IO(SP0_MCMC2) = AR;
```

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: single-word transfers or DMA block transfers. Both methods are interrupt-driven, using the same internally generated interrupts.

When SPORT DMA is not enabled in the `SPx_TCR` or `SPx_RCR` transmit or receive configuration registers, the SPORT generates an interrupt every time it has received a data word or has started to transmit a data word. 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. Service routines can then operate on the block of data rather than on single words, significantly reducing overhead. The ADSP-2191 DMA engines cycle steal from the core, resulting in one cycle of overhead imposed on the core for each DMA word transferred.

See "I/O Processor" on page 6-1 for more information about configuring and using DMA with the SPORTs.

SPORT DMA Autobuffer Mode Example

The following code fragments show an example of DMA autobuffer mode for SPORT use.

The DMA autobuffer mode is set up in this code fragment.

```
             /* SPORT0 DMA AUTOBUFFER XMIT */
AR = 0x0010;    /* Set Autobuffer, Direction, and Clear_Buffer */
IO(SP0_CONFIG_DMA_TX) = AR;

AR = 0;                 /* SPORT0 TX DMA Internal Memory Page */
IO(SP0_START_PG_TX) = AR;

AR = tx_buf;         /* SPORT0 TX DMA Internal Memory Address */
IO(SP0_START_ADDR_TX) = AR;

AR = LENGTH(tx_buf);   /* SPORT0 TX DMA Internal Memory Count */
IO(SP0_COUNT_TX) = AR;

/* SPORT0 DMA AUTOBUFFER RCV */
AR = 0x0010;    /* Set Autobuffer, Direction, and Clear_Buffer */
IO(SP0_CONFIG_DMA_RX) = AR;

AR = 0;                 /* SPORT0 RX DMA Internal Memory Page */
IO(SP0_START_PG_RX) = AR;

AR = rx_buf;          /* SPORT0 RX DMA Internal Memory Address */
IO(SP0_START_ADDR_RX) = AR;

AR = LENGTH(rx_buf);   /* SPORT0 RX DMA Internal Memory Count */
IO(SP0_COUNT_RX) = AR;

   /* ENABLE SPORT0 DMA and DIRECTION */
   /* IN DMA CONFIGURATION REGISTER */
AR = 0x1015;             /* Enable TX Interrupts */
IO(SP0_CONFIG_DMA_TX) = AR;

AR = 0x1017;             /* Enable RX Interrupts */
IO(SP0_CONFIG_DMA_RX) = AR;
```

The SPORT is enabled in the following code fragment.

```
AX0 = IO(SP0_RX_CONFIG);            /* Enable SPORT0 RX */
AR = SETBIT 0 OF AX0;
IO(SP0_RX_CONFIG) = AR;

AX0 = IO(SP0_TX_CONFIG);            /* Enable SPORT0 TX */
AR = SETBIT 0 OF AX0;
IO(SP0_TX_CONFIG) = AR;
```

SPORT Descriptor-Based DMA Example

The following code fragment is an example of setting up descriptor-based DMA mode for SPORT use.

```
/*    SPORT0 DMA DESCRIPTOR BLOCK TX */
AR = 0x0080;               /* Set Direction, and Clear_Buffer */
IO(SP0_CONFIG_DMA_TX) = AR;

AR = xmit_ddb;   /* SPORT0 xmit DMA Next Descriptor Pntr Reg */
DM(xmit_ddb + 4) = AR;

AR = LENGTH(tx_buf);  /* SPORT0 xmit DMA Internal Memory Count */
DM(xmit_ddb + 3) = AR;

AR = tx_buf;       /* SPORT0 xmit DMA Internal Memory Address */
DM(xmit_ddb + 2) = AR;

AR = 0;              /* SPORT0 xmit DMA Internal Memory Page */
DM(xmit_ddb + 1) = AR;

AR = 0x8005;          /* Enable DMA, interrupt on completion, */
                      /* software control */
DM(xmit_ddb) = AR;
```

```
/* SPORT0 DMA DESCRIPTOR BLOCK RX */
AR= 0x0082;               /* Set Direction, and Clear_Buffer */
IO(SP0_CONFIG_DMA_RX) = AR;

AR = rcv_ddb;      /* SPORT0 rcv DMA Next Descriptor Pntr Reg */
DM(rcv_ddb + 4) = AR;

AR = LENGTH(rx_buf);  /* SPORT0 rcv DMA Internal Memory Count */
DM(rcv_ddb + 3) = AR;

AR = rx_buf;       /* SPORT0 rcv DMA Internal Memory Address */
DM(rcv_ddb + 2) = AR;

AR = 0;                /* SPORT0 rcv DMA Internal Memory Page */
DM(rcv_ddb + 1) = AR;

AR = 0x8007;          /* Enable DMA, interrupt on completion, */
                      /* software control */
DM(rcv_ddb) = AR;

/* DMA CONFIG */
AR = xmit_ddb;         /* Load TX DMA NEXT Descriptor Pointer */
IO(SP0_NEXT_DESCR_TX) = AR;

AR = rcv_ddb;          /* Load RX DMA NEXT Descriptor Pointer */
IO(SP0_NEXT_DESCR_RX) = AR;

/* Signify DMA Descriptor Ready */
AR = 0x0001;
IO(SP0_DESCR_RDY_TX) = AR;           /* DMA Descriptor Ready */
IO(SP0_DESCR_RDY_RX) = AR;

AR = 0x0001;
IO(SP0_CONFIG_DMA_TX) = AR;
```

```
AR = 0x0003;
IO(SP0_CONFIG_DMA_RX) = AR;
```

Support for Standard Protocols

The ADSP-2191 supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard:

- `SPx_TFSDIVx = SPx_RFSDIVx = 0x03FF` (1024 clock cycles per frame, 122ns wide, 125ms period frame sync)
- `TFSR/RFSR` set (FS required)
- `LTFS/LRFS` set (active-low FS)
- `TSCLKDIV = RSCLKDIV =` 8 (for 8.192 MHz (+/- 2%) bit clock)
- `MCM` set (multi-channel 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

SPORTs can recover the data rate clock (`SCK`) 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 the 2-MHz or 8-MHz clock from the incoming 4-MHz or 16-MHz clock, with the proper phase relationship. A 2-bit mode signal chooses the applicable clock mode, including a non-divide/bypass mode for normal operation.

SPORT Pin/Line Terminations

The DSP 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 vs. Unframed" on page 9-26, "Early vs. Late Frame Syncs (Normal and Alternate Timing)" on page 9-29, and "Frame Syncs in Multichannel Mode" on page 9-35). This section contains additional examples to illustrate more 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-2191 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 9-6 on page 9-49 through Figure 9-11 on page 9-50 show framing for receiving data.

In Figure 9-6 on page 9-49 and Figure 9-7 on page 9-49, the normal framing mode is shown for non-continuous data (any number of SCK cycles between words) and continuous data (no SCK cycles between words). Figure 9-8 on page 9-49 and Figure 9-9 on page 9-50 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 that the output meets the input timing requirement; therefore, with two SPORT channels, one SPORT channel can provide RFS for the other SPORT channel.

Figure 9-10 on page 9-48 and Figure 9-11 on page 9-50 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 SCK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multi-word bursts (continuous reception).

Figure 9-12 on page 9-51 through Figure 9-17 on page 9-53 show framing for transmitting data and are very similar to Figure 9-6 on page 9-49 through Figure 9-11 on page 9-50.

In Figure 9-12 on page 9-51 and Figure 9-13 on page 9-51, the normal framing mode is shown for non-continuous data (any number of SCK cycles between words) and continuous data (no SCK cycles between words). Figure 9-14 on page 9-52 and Figure 9-15 on page 9-52 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the TFS output meets the TFS input timing requirement.

Figure 9-16 on page 9-53 and Figure 9-17 on page 9-53 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 SCK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

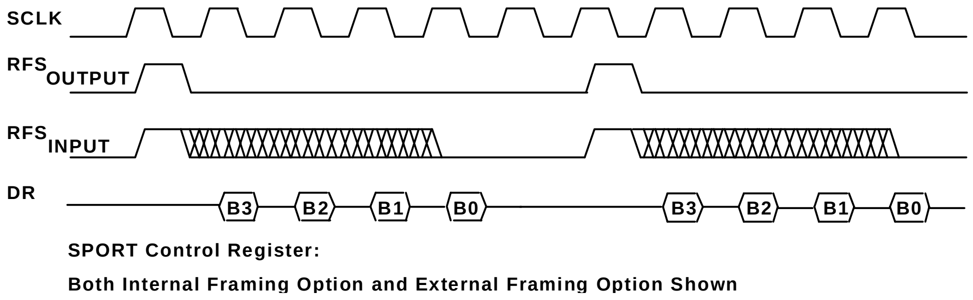

SCLK
RFS OUTPUT
RFS INPUT
DR
B3
B2
B1
B0
SPORT Control Register:
Both Internal Framing Option and External Framing Option Shown

Figure 9-6. SPORT Receive, Normal Framing

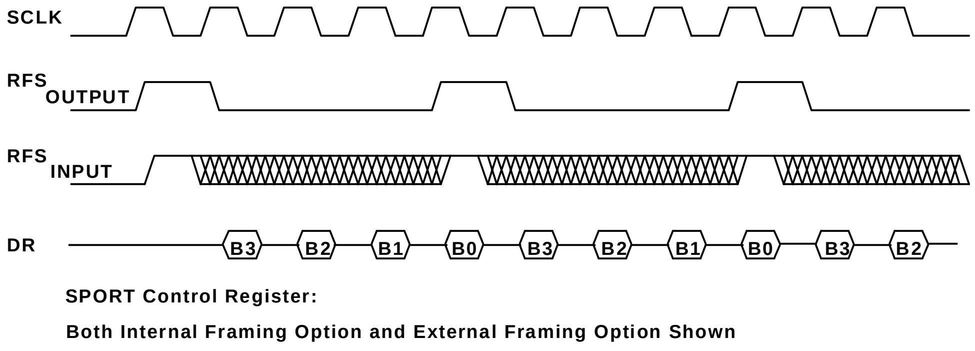

SCLK
RFS OUTPUT
RFS INPUT
DR
B3
B2
B1
B0
SPORT Control Register:
Both Internal Framing Option and External Framing Option Shown

Figure 9-7. SPORT Continuous Receive, Normal Framing

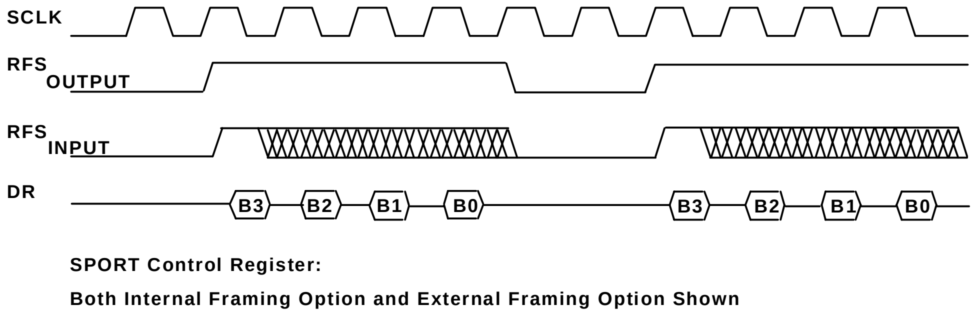

SCLK
RFS OUTPUT
RFS INPUT
DR
B3
B2
B1
B0
SPORT Control Register:
Both Internal Framing Option and External Framing Option Shown

Figure 9-8. SPORT Receive, Alternate Framing

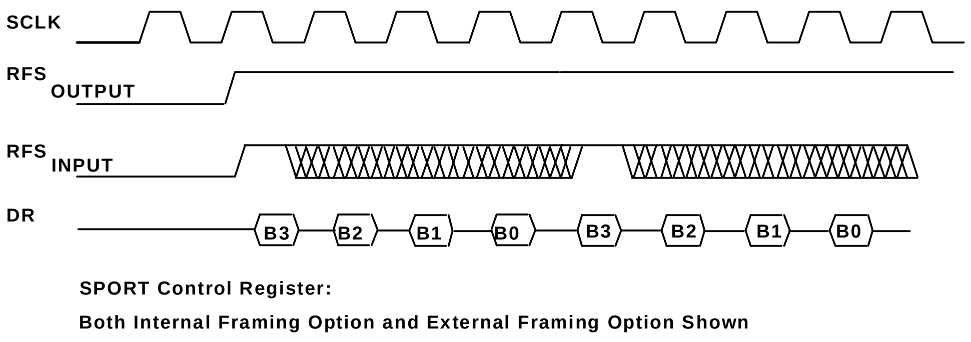


Figure 9-9. SPORT Continuous Receive, Alternate Framing

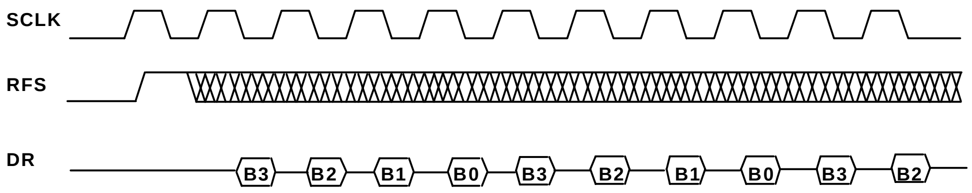


Figure 9-10. SPORT Receive, Unframed Mode, Normal Framing

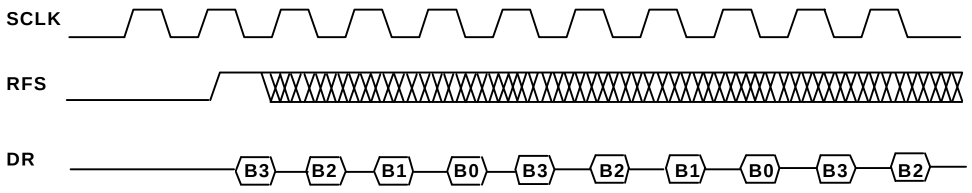


Figure 9-11. SPORT Receive, Unframed Mode, Alternate Framing

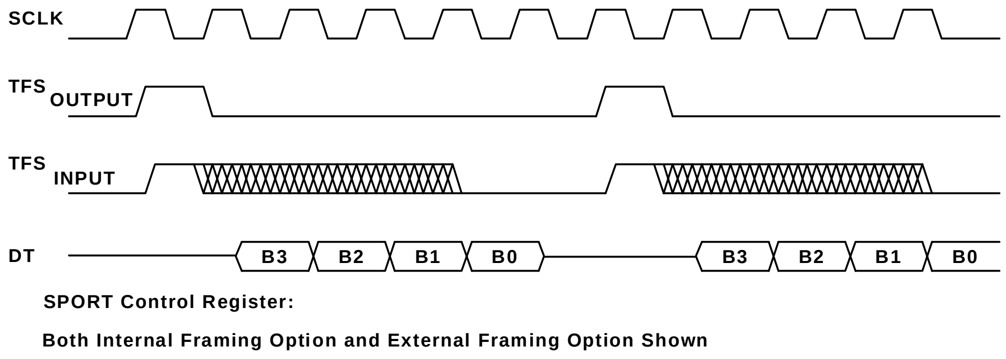


Figure 9-12. SPORT Transmit, Normal Framing

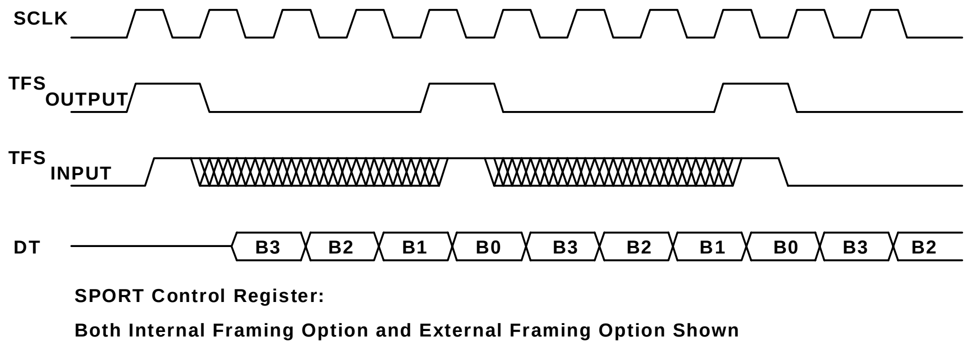
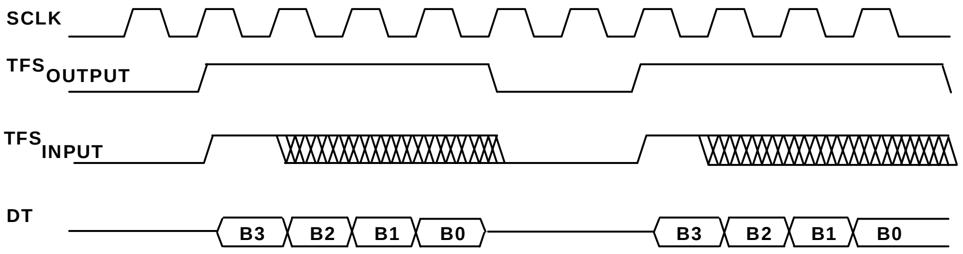


Figure 9-13. SPORT Continuous Transmit, Normal Framing

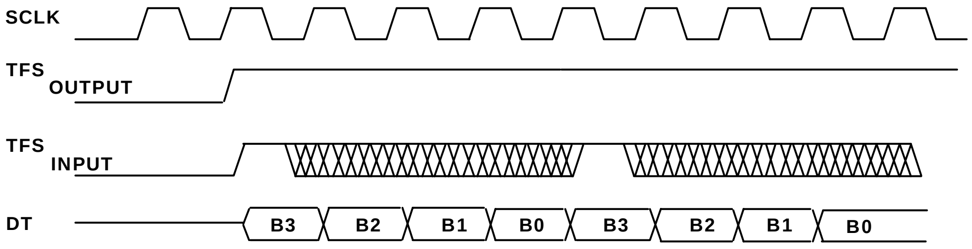


SPORT Control Register:

Both Internal Framing Option and External Framing Option Shown

Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

Figure 9-14. SPORT Transmit, Alternate Framing



SPORT Control Register:

Both Internal Framing Option and External Framing Option Shown

Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

Figure 9-15. SPORT Continuous Transmit, Alternate Framing

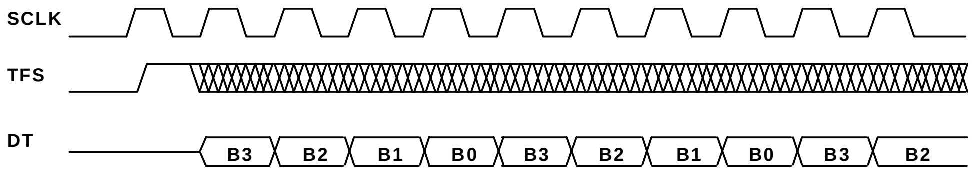


Figure 9-16. SPORT Transmit, Unframed Mode, Normal Framing

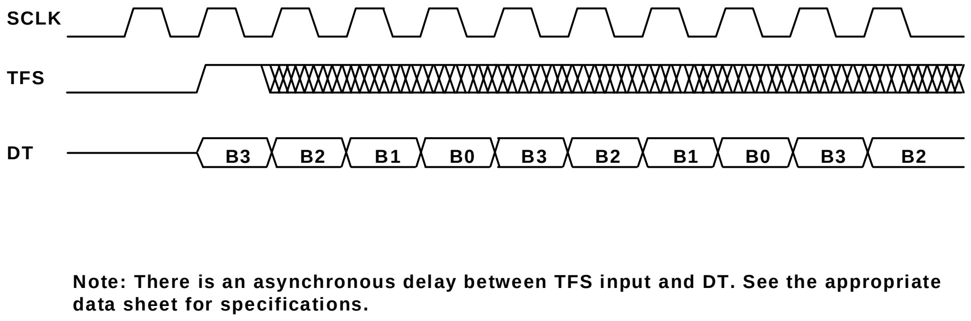

SCLK
TFS
DT
B3
B2
B1
B0
B3
B2
B1
B0
B3
B2
Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

Figure 9-17. SPORT Transmit, Unframed Mode, Alternate Framing

10 SERIAL PERIPHERAL INTERFACE (SPI) PORTS

This chapter provides the following sections:

Overview

The DSP has two independent Serial Peripheral Interface (SPI) ports (SPI0 and SPI1) that provide an I/O interface to a wide variety of SPI-compatible peripheral devices. Each SPI port employs a set of control registers and data buffers.

The SPI interface shares I/O pins with the SPORT2 serial port; SPORT2 and the SPI interface cannot be enabled at the same time.

In this text, the naming conventions for registers and pins use a lowercase `x` to represent a digit. For example, `MISOx` indicates `MISO0` and `MISO1` pins (corresponding to SPI port 0 or SPI port 1*)*.

With a range of configurable options, the SPI ports provide a glueless hardware interface with other SPI-compatible devices. SPI is a 4-wire interface consisting of two data pins, a device-select pin, and a clock pin. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multi-master environments. The ADSP-2191 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 multi-master scenario and to avoid data contention.

Typical SPI-compatible peripheral devices that can be used to interface to the ADSP-2191 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 ADSP-2191 SPI supports the following features:

- Full-duplex operation
- Master-slave mode multimaster environment
- Open drain outputs
- Programmable baud rates, clock polarities and phases
- Slave booting from another master SPI device

The ADSP-2191 Serial Peripheral Interface is an industry standard synchronous serial link that helps the DSP communicate with multiple SPI-compatible devices. The SPI peripheral is a synchronous, 4-wire interface consisting of two data pins (`MOSI` and `MISO`); one device select pin

($\overline{\text{SPISS}}$); and a gated clock pin (SCK). The two data pins permit full-duplex operation to other SPI-compatible devices. The SPI also includes programmable baud rates, clock phase, and clock polarity.

An SPI must be enabled via the System Configuration (SYSCR) register.

The SPI can operate in a multi-master environment by interfacing with several other devices, acting as a master device or a slave device. In a multi-master environment, the SPI interface uses open drain data pad driver outputs to avoid data bus contention.

Figure 10-1 is a block diagram of the ADSP-2191 SPI interface. The interface is essentially a shift register that serially transmits and receives data bits, one bit a time at the SCK rate, to/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, or shifted out serially via the shift register as new data is received or shifted in serially at the other end of the same shift register. The SCK synchronizes the shifting and sampling of the data on the two serial data pins (MOSI and MISO).

See "Pin Descriptions" on page 14-2 for a table of all ADSP-2191pins, including those used for SPI.

During SPI data transfers, one SPI device acts as the SPI link master, controlling the data flow by generating the SPI serial clock and asserting the SPI device select signal. The other SPI device acts as the slave, accepting new data from the master into its shift register, while transmitting requested data out of the shift register through its SPI transmit data pin. Multiple ADSP-2191 DSPs can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (called broadcast mode). However, only one slave may drive its output to write data back to the

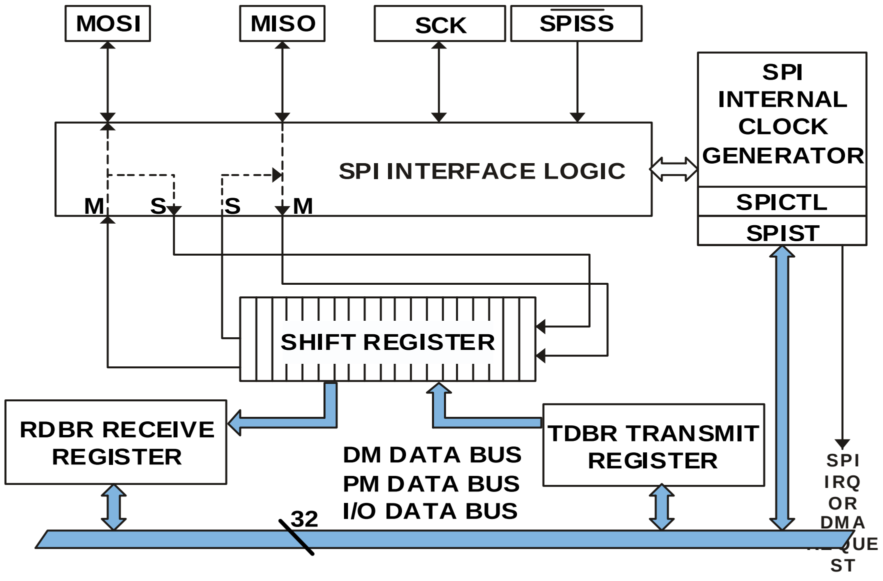


Figure 10-1. ADSP-2191 SPI Block Diagram

master at any given time. This must be enforced in broadcast mode, where several slaves can be selected to receive data from the master, but one slave only can be enabled to send data back to the master.

In a multi-master or multi-device ADSP-2191 environment where multiple ADSP-2191s are connected via their SPI ports, all `MOSI` pins are connected together, all `MISO` pins are connected together, and all `SCK` pins are connected together.

For a multi-slave environment, the ADSP-2191 can use 14 programmable flags (`PF2` - `PF15`) to be used as dedicated SPI slave-select signals for the SPI slave devices.

At reset, the SPI is disabled and configured as a slave.

Interface Signals

This section describes the SPI's interface signals.

Serial Peripheral Interface Clock Signal (SCK)

Serial Peripheral Interface clock signal (SCK) 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. SCK cycles once for each bit transmitted. It is an output signal when the device is configured as a master, and an input signal when the device is configured as a slave.

SCK is a gated clock that is active during data transfers, only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

SCK is used to shift out and shift in data driven on the MISO and MOSI lines. Data is always shifted out on one edge of the clock and sampled on the opposite clock edge. The clock polarity and clock phase relative to data are programmable into the SPI Control (SPICTLx) registers and define the transfer format.

Serial Peripheral Interface Slave Select Input Signal ($\overline{\text{SPISS}}$)

Serial Peripheral Interface Slave Select input signal ($\overline{\text{SPISS}}$) is an active low signal used to enable a ADSP-2191 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 for multi-master environment. In multi-master mode, an error occurred if the $\overline{\text{SPISS}}$ input signal of a master is asserted (driven low); this means that another device is also trying to be the master device.

Master Out Slave In (MOSI)

Master Out Slave In pin (MOSI) is one of the bidirectional I/O data pins. When the ADSP-2191 is configured as a master, MOSI becomes a data transmit (output) pin, transmitting output data. When the ADSP-2191 is configured as a slave, MOSI becomes a data receive (input) pin, receiving input data. In a ADSP-2191 SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

Master In Slave Out (MISO)

Master In Slave Out pin (MISO) is one of the bidirectional I/O data pins. When the ADSP-2191 is configured as a master, MISO becomes a data receive (input) pin, receiving input data. When the ADSP-2191 is configured as a slave, MISO becomes a data transmit (output) pin, transmitting output data. In an ADSP-2191 SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

One slave only is allowed to transmit data at any given time.

An SPI configuration example, shown in Figure 10-2, illustrates how an ADSP-2191 can be used as a slave SPI device. The 8-bit host microcontroller is the SPI master. The ADSP-2191 can be booted via its SPI interface, allowing the download of user application code and data prior to runtime.

Interrupt Behavior

The behavior of the SPI interrupt signal depends on the transfer initiation and interrupt mode (TIMOD). In DMA mode, the interrupt can be generated upon completion of a DMA multi-word transfer or upon an SPI error condition (MODF, TXE when TRAN=0, or RBSY when TRAN=1). When not using DMA mode, an interrupt is generated when the SPI is ready to accept new

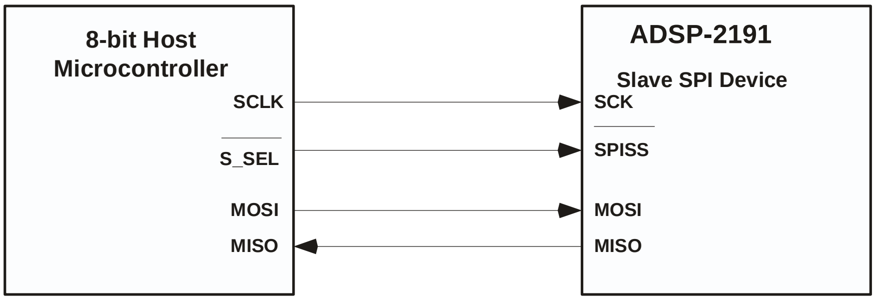


Figure 10-2. ADSP-2191 as Slave SPI Device

data for a transfer; the TXE and RBSY error conditions do not generate interrupts in these modes. An interrupt is also generated in a master when the mode-fault error occurs.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control (SPICTLx) Registers" on page 10-10.

SPI Registers

The SPI peripheral in the ADSP-2191 includes a several user-accessible registers; some of which are also accessible through the DMA bus. Four registers (SPIBAUDx, SPICTLx, SPIFLGx, and SPISTx) contain control and status information. Two registers (RDBRx and TDBRx) are used for buffering receive and transmit data. Eight registers relate DMA functionality. The shift register (SFDR), which is internal to the SPI module, is not directly accessible.

Refer to "Error Signals and Flags" on page 10-29 for information about how bits in these registers signal errors and other conditions, and "Register Mapping" on page 10-19 for a table showing the mapping of all SPI registers.

This section provides the following topics:

- “SPI Baud Rate (SPIBAUDx) Registers” on page 10-9
- “SPI Control (SPICTLx) Registers” on page 10-10
- “SPI Flag (SPIFLGx) Register” on page 10-12
- “SPI Status (SPISTx) Registers” on page 10-16
- “Transmit Data Buffer (TDBRx) Registers” on page 10-18
- “Receive Data Buffer (RDBRx) Registers” on page 10-19
- “Data Shift (SFDR) Register” on page 10-19
- “Register Mapping” on page 10-19

SPI Baud Rate (SPIBAUDx) Registers

The SPI Baud Rate (SPIBAUDx) registers 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 the following formula:

$$\text{CKx frequency} = \frac{\text{Perpheral clock frequenc}}{2 \times \text{SPIBAUDx}}$$

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the peripheral clock rate (HCLK).

Table 10-1 provides bit descriptions for the SPIBAUDx register.

Table 10-1. SPIBAUDx Register Bits

Bit(s)	Function	Default
15:0	Baud Rate: Peripheral clock (HCLK) divided by 2*(Baud)	0

Table 10-2 lists several possible baud rate values for the SPIBAUDx register.

Table 10-2. SPI Master Baud Rate Example

SPIBAUD Decimal Value	SPI Clock Divide Factor	Baud Rate for HCLK @ 100MHz
0	N/A	N/A
1	N/A	N/A
2	4	25MHz
3	6	16.7MHz
4	8	12.5MHz
65,535 (0xFFFF)	131,070	763Hz

SPI Control (SPICTLx) Registers

The SPI Control (SPICTLx) registers configure and enable the SPI system. These registers enable the SPI interface, select the device as a master or slave, and determine data transfer format and word size.

Table 10-3 provides the bit descriptions for the SPICTLx register.

Table 10-3. SPICTLx Register Bits

Bit(s)	Name	Function	Default
1:0	TIMOD	Defines transfer initiation mode and interrupt generation. 00 - Initiate transfer by read of receive buffer. Interrupt active when receive buffer is full 01 - Initiate transfer by write to transmit buffer. Interrupt active when transmit buffer is empty 10 - Enable DMA transfer mode. Interrupt configured by DMA 11 - Reserved	00
2	SZ	Send Zero or last word when TDBRx empty. 0 = send last word 1 = send zeroes	0

Table 10-3. SPICTLx Register Bits (Cont'd)

Bit(s)	Name	Function	Default
3	GM	When RDBRx full, get data or discard incoming data. 0 = discard incoming data 1 = get more data (overwrites the previous data)	0
4	PSSE	Enables Slave-Select ($\overline{\text{SPISS}}$) input for master. When not used, $\overline{\text{SPISS}}$ can be disabled, freeing up a chip pin as general purpose I/O. 0 = disable 1 = enable	0
5	EMISO	Enable MISO pin as an output. This is needed when master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. All slaves (except for the one from whom the master wishes to receive) should have this bit set. 0 = MISO disabled 1 = MISO enabled	0
7:6		Reserved	00
8	SIZE	Word length. 0 = 8 bits 1 = 16 bits	0
9	LSBF	Data format. 0 = MSB sent/received first 1 = LSB sent/received first	0
10	CPHA	Clock phase (selects the transfer format). 0 = SPIOSELx is set automatically by hardware. SCK starts toggling at the middle of first data bit 1 = SPIOSELx has to be set by software. SCK starts toggling at the beginning of first data bit	1
11	CPOL	Clock polarity. 0 = active-high SCK (SCK low is the idle state) 1 = active-low SCK (SCK high is the idle state)	0
12	MSTR	Configures SPI module as master or slave. 0 = device is a slave device 1 = device is a master device	0

Table 10-3. SPICTLx Register Bits (Cont'd)

Bit(s)	Name	Function	Default
13	WOM	Open drain data output enable (for MOSI and MISO). 0 = Normal 1 = Open Drain	0
14	SPE	SPI module enable 0 = SPI Module is disabled 1 = SPI Module is enabled	0
15		Reserved	0

SPI Flag (SPIFLGx) Register

The SPI Flag (SPIFLGx) registers are read/write registers used to enable individual SPI slave-select lines when the SPI is enabled as a master. Each SPIFLG register has seven bits (FLS) to select the outputs to be driven as slave-select lines and seven bits (FLG) to activate the selected slave-selects.

If the SPI is enabled and configured as a master, up to 14 of the chip's general-purpose programmable flag pins may be used as slave-select outputs. For each FLS bit set in the SPIFLGx register, the corresponding PFx pin is configured as a slave-select output. For example, if bit FLS1 is set in SPIFLG0, the PF2 pin will be driven as a slave-select (SPI0SEL1).

Refer to the following tables for the mapping of SPIFLGx register bits to PFx pins. For FLS bits that are not set, the corresponding PFx pins are configured and controlled by the chip's general-purpose PFx registers (DIR and others). When the chip is configured for 16-bit external data transfers (as defined by the external bus width bit, E_BWS, in the EMICTRL register), the SPIFLG0 and SPIFLG1 registers each lose the capability to control four slave devices due to the EMI's use of the PF15-8 pins. Because the EMI uses these PFx pins, the MUXed SPI use of those pins no longer applies; therefore, SPIxSEL7-4 are no longer available as slave select outputs.

In order for the SPIxSELx pins to be configured as SPI slave-select outputs, SPIx must be enabled as a master (that is, the SPE and MSTR bits in the SPICTLx register must be set). Otherwise, none of the bits in the SPIFLGx register have any effect. When the EMI is configured to be 16 bits, SPI-FLGx bits 4-7 and bits 12-15 have no effect due to the EMI's use of the corresponding PFx pins."

Table 10-4 provides bit mappings for the SPIFLG0 register.

Table 10-4. SPIFLG0 Register Bits

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	SPI0SEL1 Enable	PF2	0
2	FLS2	SPI0SEL2 Enable	PF4	0
3	FLS3	SPI0SEL3 Enable	PF6	0
4	FLS4	SPI0SEL4 Enable	PF8	0
5	FLS5	SPI0SEL5 Enable	PF10	0
6	FLS6	SPI0SEL6 Enable	PF12	0
7	FLS7	SPI0SEL7 Enable	PF14	0
8		Reserved		1
9	FLG1	SPI0SEL1 Value	PF2	1
10	FLG2	SPI0SEL2 Value	PF4	1
11	FLG3	SPI0SEL3 Value	PF6	1
12	FLG4	SPI0SEL4 Value	PF8	1
13	FLG5	SPI0SEL5 Value	PF10	1
14	FLG6	SPI0SEL6 Value	PF12	1
15	FLG7	SPI0SEL7 Value	PF14	1

Table 10-5 provides bit mappings for the SPIFLG1 register.

Table 10-5. SPIFLG1 Register Bits

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	SPI1SEL1 Enable	PF3	0
2	FLS2	SPI1SEL2 Enable	PF5	0
3	FLS3	SPI1SEL3 Enable	PF7	0
4	FLS4	SPI1SEL4 Enable	PF9	0
5	FLS5	SPI1SEL5 Enable	PF11	0
6	FLS6	SPI1SEL6 Enable	PF13	0
7	FLS7	SPI1SEL7 Enable	PF15	0
8		Reserved		1
9	FLG1	SPI1SEL1 Value	PF3	1
10	FLG2	SPI1SEL2 Value	PF5	1
11	FLG3	SPI1SEL3 Value	PF7	1
12	FLG4	SPI1SEL4 Value	PF9	1
13	FLG5	SPI1SEL5 Value	PF11	1
14	FLG6	SPI1SEL6 Value	PF13	1
15	FLG7	SPI1SEL7 Value	PF15	1

When the PFx pins are configured as slave-select outputs, the value driven onto these outputs depends on the value of the CPHA bit in the SPICTLx register. If CPHA=1, the value is set by software control of the FLG bits. If CPHA=0, the value is determined by the SPI hardware, and the FLG bits are ignored.

When CPHA=1, the SPI protocol permits the slave-select line to remain asserted (low) or be de-asserted between transferred words. This requires that you write to the SPIFLGx register, setting or clearing the appropriate FLG bits as needed. For example, to drive PF3 as a slave-select, FLS1 in

SPIFLG1 must be set. Clearing FLG1 in SPIFLG1 drives PF3 low; setting FLG1 drives PF3 high. If needed, PF3 can be cycled high and low between transfers by setting FLG1 and then clearing FLG1; otherwise, PF3 remains active (low) between transfers.

When CPHA=0, the SPI protocol requires that the slave-select be de-asserted 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 FLS1 bit in the SPIFLG1 register. Writing to the FLG1 bit is not required, because the SPI hardware automatically drives the PF3 pin.

Slave-Select Inputs

The behavior of the SPISSx inputs depend on the configuration of the SPI. If the SPI is a slave, SPISS acts as the slave-select input. When enabled as a master, SPISS can serve as an error-detection input for the SPI in a multi-master environment. The PSSE bit in the SPICTLx register enables this feature. When PSSE=1, the SPISS input is the master mode error input; otherwise, SPISS is ignored. The state of these input pins can be observed in the Programmable Flag Data register (FLAGC or FLAGS).

Using the SPIFLG Register's FLS Bits for Multiple-Slave SPI Systems

The FLS bits in the SPIFLG register are used in a multiple-slave SPI environment. For example, if eight SPI devices are in a system with an ADSP-2191 master, the master ADSP-2191 can support the SPI mode transactions across all seven other devices. This configuration requires that only one ADSP-2191 is a master within this multi-slave environment. For example, assume that SPI0 is the master. The seven flag pins (PF2, PF4, PF6, PF8, PF10, PF12, and PF14) on the ADSP-2191 master can be connected to each of the slave SPI device's $\overline{\text{SPISS}}$ pin. In this configuration, the FLS bits in the SPIFLG register can be used three ways.

In cases 1 and 2, the ADSP-2191 is the master, and the seven microcontrollers/peripherals with SPI interfaces are used as slaves. In this setup, the ADSP-2191 can:

1. Transmit to all seven SPI devices at the same time (broadcast mode). All the FLS 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 ADSP-2191 DSPs:

3. If all the slaves are also ADSP-2191s, the requestor can receive data from only one ADSP-2191 (enable this by setting the EMISO bit in the other six slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in other microcontrollers. Therefore, it would be possible to use the EMISO feature with any other SPI device that has this functionality.

Figure 10-3 shows one ADSP-2191 as a master with three ADSP-2191s (or other SPI-compatible devices) as slaves.

SPI Status (SPISTx) Registers

Use the SPI Status (SPISTx) register to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPISTx registers can be read at any time.

Some of the bits in SPISTx registers are read-only (RO), and others can be cleared by a write-one-to-clear (W1C) operation. Bits that just provide information about the SPI are read-only; these bits are set and cleared by the hardware. W1C bits are set when an error condition occurs; these bits are set by hardware, and must be cleared by software. (To clear a W1C bit, write a 1 to the desired bit position of the SPISTx register. For example, if

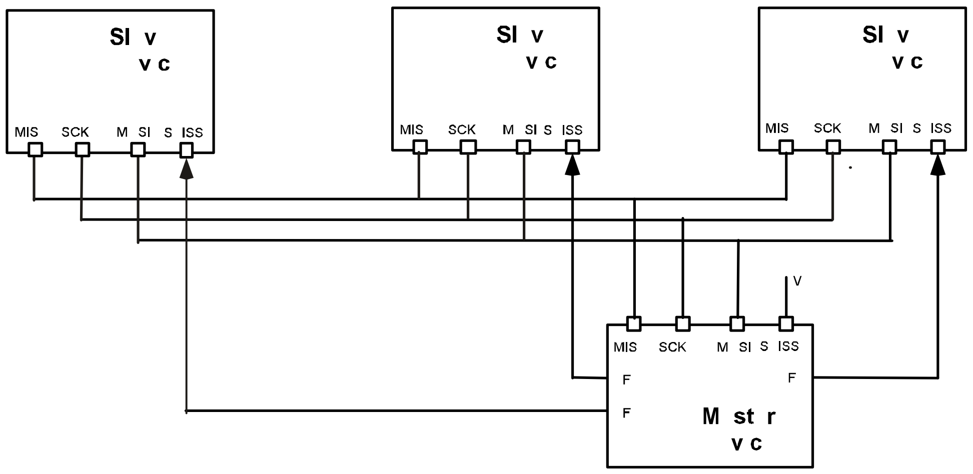


Figure 10-3. Single-Master, Multiple-Slave Configuration (All ADSP-2191s)

the `TXE` bit is set, write a 1 to bit 2 of `SPISTx` to clear the `TXE` error condition. This allows you to read the status register without changing its value.)

Write-one-to-clear (W1C) bits only can be cleared by writing one to them. Writing zero does not clear (or affect) a W1C bit.

Table 10-6 provides bit descriptions for the `SPISTx` register.

Table 10-6. SPI Status (SPISTx) Register Bits

Bit	Name	Function	Type	Default
0	SPIF	This bit is set when an SPI single-word transfer is complete.	RO	1
1	MODF	Mode fault error. This bit is set in a master device when some other device tries to become the master.	W1C	0
2	TXE	Transmission error. This bit is set when a transmission occurred with no new data in the TDBRx register.	W1C	0

Table 10-6. SPI Status (SPISTx) Register Bits (Cont'd)

Bit	Name	Function	Type	Default
3	TXS	TDBRx data buffer status. 0 = empty 1 = full	RO	0
4	RBSY	Receive error. This bit is set when data is received and the receive buffer is full.	W1C	0
5	RXS	RX data buffer status. 0 = empty 1 = full	RO	0
6	TXCOL	Transmit collision error. When this bit is set, corrupt data may have been transmitted.	W1C	0

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.

Transmit Data Buffer (TDBRx) Registers

The Transmit Data Buffer (`TDBRx`) registers are 16-bit read-write (RW) registers. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in `TDBR` is loaded into the Shift Data (`SFDR`) register. A normal core read of `TDBRx` may occur at any time and does not interfere with, or initiate, SPI transfers.

When the DMA is enabled for transmit operation, data is loaded into this register before being transmitted and then loaded into the shift register just prior to the beginning of a data transfer.

A normal core write to `TDBRx` should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of TDBRx will be transmitted repeatedly. A normal core write to TDBRx is permitted in this mode, and this data will be transmitted. If the send zeroes control bit (SZ) is set, TDBRx may be reset to 0 in certain circumstances.

If multiple writes to TDBRx occur while a transfer is in progress, only the last data written will be transmitted; no intermediate values written to TDBRx will be transmitted. Multiple writes to TDBRx, though possible, are not recommended.

Receive Data Buffer (RDBRx) Registers

The Receive Data Buffer (RDBRx) registers are 16-bit read-only (RO) registers. At the end of a data transfer, the data in the shift register is loaded into RDBRx. During a DMA receive operation, the data in RDBRx is automatically read by the DMA. A shadow register (RDBRSx) for the receive data buffer (RDBRx) is provided for use in debugging software. RDBRSx is at a different address from RDBRx, but its contents are identical to that of RDBRx. When a software read of RDBRx occurs, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD=00). No such hardware action occurs when the shadow register is read. RDBRSx is a read-only (RO) register.

Data Shift (SFDR) Register

The Data Shift (SFDR) register is the 16-bit data shift register; it is not accessible by the software or the DMA. The SFDR is buffered so a write to TDBRx does not overwrite the shift register during an active transfer.

Register Mapping

Table 10-7 illustrates the mapping of all SPI registers. Refer to the notes following the table for more information about this data.

Table 10-7. SPI Register Mapping

Register Name	Function
SPICTLx	SPI port control
SPIFLG	SPI port flag
SPISTx	SPI port status
TDBRx	SPI port transmit data buffer
RDBRx	SPI port receive data buffer
SPIBAUDx	SPI port baud control
RDBRSx	SPI port data
SPIxD_PTR	SPI port DMA current pointer
SPIxD_CFG	SPI port DMA configuration
SPIxD_SRP	SPI port DMA start page
SPIxD_SRA	SPI port DMA start address
SPIxD_CNT	SPI port DMA count
SPIxD_CP	SPI port DMA next descriptor pointer
SPIxD_CPR	SPI port DMA descriptor ready
SPIxD_IRQ	SPI port interrupt status

Some items to note about Table 10-7 include:

- SPICTLx: The SPE and MSTR bits can also be modified by hardware (when MODF is set).
- SPISTx: The SPIF bit can be set by clearing SPE in SPICTLx.
- TDBRx: Register contents can also be modified by hardware (by DMA and/or when SZ=1).
- RDBRx: When this register is read, hardware events are triggered.

- RDBRSx: Although this register has the same contents as RDBRx, no action is taken when it is read.
- SPIxD_SRP, SPIxD_SRA, and SPIxD_CNT can be written to only via software when the DAUTO DMA configuration bit is set.
- SPIxD_CFG: Three of the control bits (TRAN, DCOME, and DERE) only can be written to via software when the DAUTO DMA bit is set.
- SPIxD_CFG: The MODF, TXE, and RBSY bits are sticky; these bits remain set even when the corresponding SPISTx bits are cleared.

SPI Transfer Formats

The ADSP-2191 SPI supports four different combinations of serial clock phase and polarity. User application code can select any of these combinations using the CPOL and CPHA bits in the SPI Control (SPICTLx) register.

Figure 10-4 on page 10-13 and Figure 10-5 on page 10-14 demonstrate the two basic transfer formats as defined by the CPHA bit; one diagram is for CPHA=0, and the other is for CPHA=1. Two waveforms are shown for SCK: one for CPOL=0 and the other for CPOL=1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the $\overline{\text{SPISS}}$ signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE=0) with MSB first (LSBF=0). Any combination of the SIZE and LSBF bits of the SPICTLx register is allowed. For example, a 16-bit transfer with the LSB first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When `CPHA=0`, the slave select line, $\overline{\texttt{SPISS}}$, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When `CPHA=1`, $\overline{\texttt{SPISS}}$ may remain active (low) between successive transfers or be inactive (high). This must be controlled by the software.

Figure 10-4 shows the SPI transfer protocol for `CPHA=0`. Note that `SCK` starts toggling in the middle of the data transfer, `SIZE=0`, and `LSBF=0`.

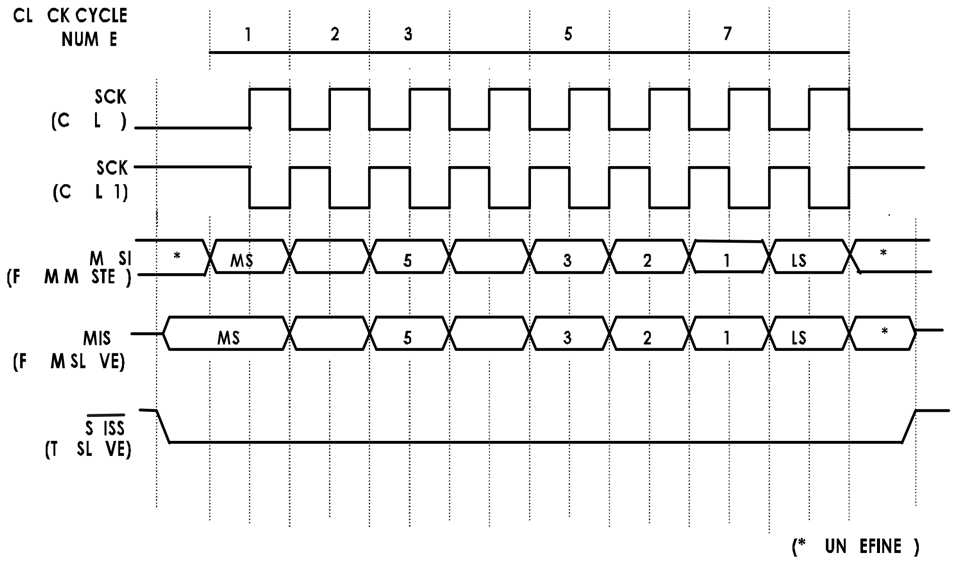


Figure 10-4. SPI transfer protocol for `CPHA=0`

Figure 10-5 shows the SPI transfer protocol for `CPHA=1`. Note that `SCK` starts toggling at the beginning of the data transfer, `SIZE=0`, and `LSBF=0`.

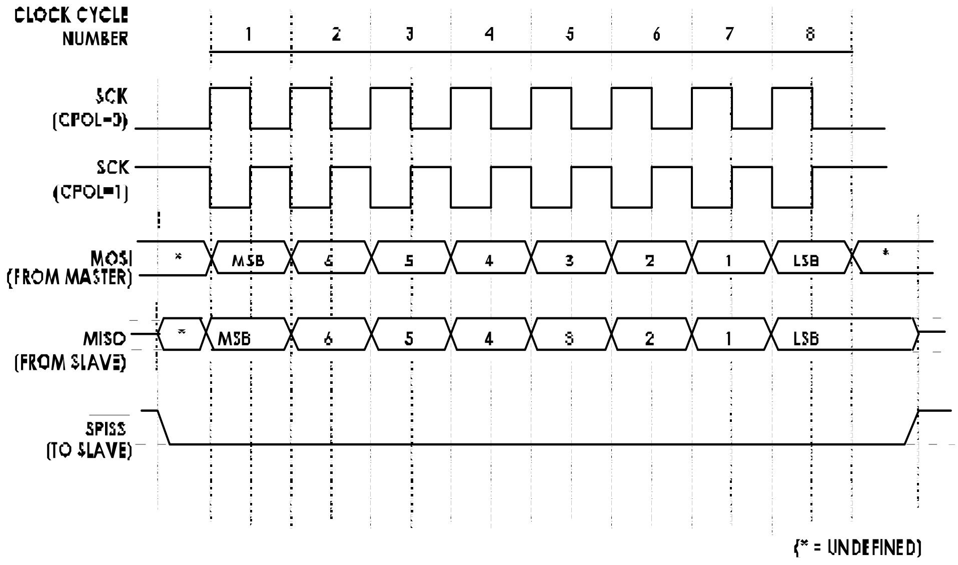


Figure 10-5. SPI Transfer Protocol for CPHA=1

SPI General Operation

This section provides the following topics:

- "Overview" on page 10-24
- "Clock Signals" on page 10-25
- "Master Mode Operation" on page 10-25
- "Transfer Initiation from Master (Transfer Modes)" on page 10-26

Overview

The SPI in ADSP-2191 DSP can be used in a single-master or multi-master environment. The MOSI, MISO, and the SCK signals are 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 slave must be in transmit mode driving the MISO line. If the transmit or receive is not needed, it can be ignored. This section describes:

- Clock signals
- SPI operation as a master and as a slave
- Error generation

Precautions must be taken when changing the SPI module configuration, in order to avoid data corruption. The configuration must not be changed during a data transfer. Change clock polarity only when no slaves are selected (except when an SPI communication link consists of a single master and a single slave, CPHA=1, and the slave's slave-select input 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 have been configured).

In a multi-master or multi-slave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open-drain drivers, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both pins (MOSI and MISO) when this option is selected.

The WOM bit controls this feature. When WOM is set and the ADSP-2191 SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic-high. The MOSI pin is not three-stated when the driven data is a logic-low. Similarly, when WOM is set and the ADSP-2191 SPI is configured as a slave, MISO is three-stated when the data driven out on MISO is a logic-high.

Clock Signals

The SCK signal is a gated clock that is active only during data transfers, and only 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 peripheral clock rate. For master devices, the clock rate is determined by the 16-bit value of the Baud Rate (SPIBAUDx) register; for slave devices, the value in SPIBAUDx is ignored. When the SPI device is a master, SCK is an output signal; when the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave-select input is driven inactive (high).

SCK shifts out and shifts in the data driven onto the MISO and MOSI lines. The data is always shifted out on one edge of the clock (the active edge) and sampled on the opposite edge of the clock (the sampling edge). Clock polarity and clock phase relative to data are programmable into the SPI Control (SPICTLx) register and define the transfer format.

Master Mode Operation

When SPI is configured as a master (and DMA mode is not selected), the interface operates as follows:

1. The core writes to the SPIFLG register, setting one or more SPI flag select bits (FLS). This ensures that the desired slaves are properly de-selected while the master is configured.

2. The core writes to the SPICTLx and SPIBAUDx 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 SPI flag bits (FLG) of SPIFLG.

4. The TIMOD bits in the SPICTLx register determine the SPI transfer initiate mode. The transfer on the SPI link begins upon a data write by the core to the transmit data buffer register (TDBRx) or a data read of the receive data buffer (RDBRx).

5. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before starting to shift, the shift register is loaded with the contents of the TDBRx register. At the end of the transfer, the contents of the shift register are loaded into RDBRx.

6. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initialize mode.

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 the SPICTLx register. If SZ=1 and the transmit buffer is empty, the device repeatedly transmits 0's on the MOSI pin; one word is transmitted for each new transfer initiate command. If SZ=0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM=1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the RDBRx buffer. If GM=0 and the receive buffer is full, the incoming data is discarded, and the RDBRx register 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 the SPICTL register. Based on those two bits and the status of the interface, a new transfer is started upon either a read of RDBR or a write to TDBR. This is summarized in Table 10-8.

Table 10-8. Transfer Initiation

TIMOD	Function	Transfer initiated upon	Action, Interrupt
00	Transmit and Receive	Initiate new single-word transfer upon read of RDBR and previous transfer completed.	Interrupt active when receive buffer is full. Read of RDBR clears interrupt.
01	Transmit and Receive	Initiate new single-word transfer upon write to TDBR and previous transfer completed.	Interrupt active when transmit buffer is empty. Writing to TDBR clears interrupt.
10	Transmit or Receive with DMA	Initiate new multi-word transfer upon write to DMA enable bit. Individual word transfers begin with either a DMA write to TDBR or a DMA read of RDBR (depending on TRAN bit), and last transfer complete.	Interrupt active upon DMA error or multi-word transfer complete. Write-1 to DMA Interrupt register clears interrupt.
11	Reserved	N/A	N/A

Slave Mode Operation

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the $\overline{\text{SPISS}}$ select signal to the active state (low) or by the first active edge of the clock (SCK), depending on the state of CPHA.

The following steps illustrate SPI operation in the slave mode:

1. The core writes to the SPICTLx register 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 the TDBRx register.

3. Once the $\overline{\text{SPISS}}$ falling edge is detected, the slave starts sending and receiving data on active SCK edges.

4. Reception/transmission continues until $\overline{\text{SPISS}}$ 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 $\overline{\text{SPISS}}$ and/or active SCK clock edge.

If the transmit buffer remains empty, or the receive buffer remains full, the devices operates according to the states of the SZ and GM bits in the SPICTLx register. If SZ=1 and the transmit buffer is empty, the device repeatedly transmits 0's on the MISO pin. If SZ=0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM=1 and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the RDBRx buffer. If GM=0 and the receive buffer is full, the incoming data is discarded, and the RDBRx register is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, Table 10-9 lists the actions necessary to prepare the device for a new transfer.

Table 10-9. Transfer Preparation

TIMOD	Function	Action, Interrupt
00	Transmit and Receive	Interrupt active when receive buffer is full. Read of RDBR clears interrupt.
01	Transmit and Receive	Interrupt active when transmit buffer is empty. Writing to TDBR clears interrupt.
10	Transmit or Receive with DMA	Interrupt configured in SPI DMA Configuration Register. Interrupt active upon DMA error or multi-word transfer complete. Write-1 to DMA Interrupt register clears interrupt.
11	Reserved	N/A

Error Signals and Flags

The status of a device is indicated by the SPISTx register. See "SPI Status (SPISTx) Registers" on page 10-16 for information about the SPISTx register.

Mode-Fault Error (MODF)

The MODF bit is set in the SPI Port Status (SPISTx) register when the $\overline{\text{SPISS}}$ input pin of a device enabled as a master is driven low by another device in the system. This occurs in multi-master systems when another device is also trying to be the master. To enable this feature, set the PSSE bit in SPICTLx. Contention between two drivers can potentially damage the driving pins. As soon as this error is detected, the following actions are taken:

1. The MSTR control bit in SPICTLx is cleared, configuring the SPI interface as a slave.
2. The SPE control bit in SPICTLx is cleared, disabling the SPI system.
3. The MODF status bit in SPISTx is set.
4. An SPI interrupt is generated.

These four conditions persist until the MODF bit is cleared by a write-1 (W1C) software operation. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents you from setting 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, check the state of the $\overline{\text{SPISS}}$ input pin to ensure that the pin is high; otherwise, once SPE and MSTR are set, another mode-fault error condition occurs immediately. The state of the input pin is observable in the Programmable Flag Data (FLAGC or FLAGS) register.

Because SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave-select output pins will revert to being controlled by the Programmable Flag registers. This can lead to contention on the slave-select lines if these lines are still being driven by the ADSP-2191. To ensure that the slave-select output drivers are disabled once a MODF error occurs, the program must configure the Programmable Flag registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the PFx pins used as slave-selects; programs can do this by writing to the DIR register prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave-selects are automatically reconfigured as PFx pins, the slave-select output drivers will be disabled.

Transmission Error (TXE) Bit

The TXE bit is set in the SPI Status (SPISTx) register when all of the conditions of transmission are met but there is no new data in TDBRx (TDBRx is empty). In this case, the contents of the transmission depend on the state of the SZ bit in the SPI Control (SPICTLx) register. The TXE bit is cleared by a write-1 (W1C) software operation.

Reception Error (RBSY) Bit

The RBSY flag is set in the SPISTx register when a new transfer has completed before the previous data is read from the SPI Receive Data (RDBRx) register. This bit indicates that a new word was received while the receive buffer was full. The RBSY flag is cleared by a write-1 (W1C) software operation. The state of the GM bit in the SPICTLx register determines whether the RDBRx register is updated with the newly-received data.

Transmit Collision Error (TXCOL) Bit

The TXCOL flag is set in the SPI Status (SPISTx) register when a write to the TDBRx register coincides with the load of the shift register. The write to TDBRx can be via the software or the DMA. This bit indicates that corrupt data may have been loaded into the shift register and transmitted; in this case, the data in TDBRx may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is cleared by a write-1 (W1C) software operation.

This bit is never set when the SPI is configured as a slave with CPHA=0; the collision may occur, but it cannot be detected.

Beginning and Ending an SPI Transfer

An defined start and end of an SPI transfer depends on whether the device is configured as a master or a slave, the CPHA mode selected, and the transfer initiation mode (TIMOD) selected. For a master SPI with CPHA=0, a transfer starts when the TDBRx register is written or the RDBRx register is read, depending on TIMOD. At the start of the transfer, the enabled slave-select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA=0, the transfer starts as soon as the $\overline{\text{SPISS}}$ input goes low.

For CPHA=1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read; the TXS bit defines when the transmit buffer can be filled. The end of a single-word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer (RDBRx). RXS is set shortly after the last sampling edge of SCK. The latency is typically a few HCLK cycles and is independent of CPHA, TIMOD, and the baud rate. If configured to generate an interrupt when RDBRx is full (TIMOD=00), the interrupt goes active one HCLK 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 set at the same time as RXS; for a master device, SPIF is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

Thus, 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 (SPIBAUD<4). SPIF is set before RXS is set, and consequently before new data has been latched into RDBRx, because of the latency. Therefore, for SPIBAUD=2 or SPIBAUD=3, wait for RXS to be set (after SPIF is set) before reading RDBRx. For larger SPIBAUD settings, RXS is guaranteed to be set before SPIF is set.

DMA

The SPI port also can use Direct Memory Accessing (DMA). For more information on DMA, see "I/O Processor" on page 6-1. For specific information on SPI DMA, see the following sections:

- "SPI Port DMA Settings" on page 6-23
- "Using SPI Port DMA" on page 6-33
- "SPI DMA in Master Mode" on page 6-33
- "SPI DMA in Slave Mode" on page 6-35
- "SPI DMA Errors" on page 6-37

SPI Example

The following example illustrates how to set up the Serial Peripheral Interface (SPI) on the ADSP-2191 DSP. The ADSP-2191 has two identical Serial Peripheral Interfaces (SPI0 and SPI1), which can be independently configured as master or slave devices.

In the following example, SPI0 is configured as a master. The sample code demonstrates the configuration and initialization of SPI0 to transfer a 16-location buffer via SPI0 at a 9600 baud rate. This example can be demonstrated in hardware with the ADSP-2191 by wiring SPI0's `MISO` and `MOSI` pins together in a simple loopback configuration.

Because it uses loopback mode, this example does not use a slave device; however, a typical SPI system may have multiple slave devices for every master. To distinguish between slaves, there are seven slave-select lines for each master SPI on the ADSP-2191. To initialize the appropriate slave-select lines, the core writes to the `SPIFLGx` register, setting one or more of the SPI flag select bits (`FLS`). In a master-slave system, writing to the `SPIFLGx` register first ensures that the desired slave is properly de-selected while the master is being configured.

```
IOPG = SPI0_Controller_Page;
                    /* Set Up Device Select Over SPI0 Interface */
AR = 0xFF02;        /* Enable Slave on Programmable Flag Pin 2 */
IO(SPIFLG0) = AR;   /* Write to SPI0 Flag Register */
```

Next, the core writes to the `SPIBAUD` and `SPICTLx` registers, enabling SPI0 as a master and configuring the system with the appropriate word length (16-bit), transfer format (MSB first), baud rate (9600), and any other necessary configuration values:

```
/* Write to SPI0 Baud rate register */
AR = 0x1047;
IO(SPIBAUD0) = AR;     /* SCLK0 ~= 9600 */
```

```
        /* Set up SPI0 Configuration Register */
AR = 0x5D08;
IO(SPICTL0) = AR;           /* Enable SPI0 as MASTER */
```

In a typical system, the core activates the desired slave by clearing the flag bit (`FLG`) of the `SPIFLGx` register, depending on `CPHA` (bit 10 in the `SPICTLx` register). In this example, `CPHA=1`.

The SPI0 transfer initiate mode (`TIMOD` bits in the `SPICTLx` register) is configured to initiate a transfer upon a read of the receive data buffer (`RDBRx`). In this mode, an interrupt is generated whenever `RDBRx` is full. With each core read of `RDBRx`, SPI0 continues to send and receive words.

If the transmit buffer remains empty or the receive buffer remains full, SPI0 operates according to the states of the `SZ` and `GM` bits in the `SPICTLx` register. `GM=1` in this example, so the device continues to receive new data from the `MISO` pin, overwriting the older data in the `RDBRx` buffer.

Once SPI0 has been configured, interrupts are enabled, and a dummy read of `RDBRx` is performed (due to the `TIMOD` configuration) to initiate the first transfer:

```
ENA INT;                 /* Globally enable interrupts */
IOPG = SPI0_Controller_Page;
AR = IO(RDBR0);          /* read from RDBR0 to start x-fer */
```

The following sample source code illustrates SPI0 setup for the ADSP-2191. These included code modules were built using the VisualDSP 2.0++ development tools for the ADSP-219x processor family and the ADSP-2191 EZ-KIT Lite evaluation kit.

This part of the example sets up the SPI master.

```
#include <def2191.h>

/*   GLOBAL & EXTERNAL DECLARATIONS */
```

```
.GLOBAL Start;

    /* DM data */

.section/data data1;
.var TX_Buf_MASTER[16] = 0xA000, 0xAA10, 0xAA20, 0xAA30,
                         0xAA40, 0xAA50, 0xAA60, 0xAA70,
                         0xAA80, 0xAA90, 0xAAA0, 0xAAB0,
                         0xAAC0, 0xAAD0, 0xAAE0, 0xAAF0;
.var RX_Buf_MASTER[16];   /* Buffer For SPI0 Receive Data */

/*     Program memory code */

.SECTION /pm program;
Start:
_main:
   call Program_SPI0_Interrupt;    /* Initialize Interrupt */
   call Program_SPI0_Interrupt;    /* Priorities */
   call SPI0_Register_Initialization;    /* Initialize SPI0 */
   call Initiate_Transfers;              /* Start Transfer */

wait_forever:
   jump wait_forever;

/*    INTERRUPT PRIORITY CONFIGURATION */

.section/code program;    /* PROGRAM STARTS HERE */
Program_SPI0_Interrupt:
   IOPG = 0;
   ar=io(SYSCR);     /* Map Interrupt Vector Table to Page 0*/
   ar = setbit 4 of ar;
   ar = setbit 0 of ar;        /* select SPI's (Not SPORT2) */
   io(SYSCR)=ar;
```

```
    DIS int;                  /* Disable all interrupts */
    IRPTL = 0x0;              /* Clear all interrupts */
    ICNTL = 0x0;              /* Interrupt nesting disable */
    IMASK = 0;                /* Mask all interrupts */

        /* Set up Interrupt Priorities */
    IOPG = Interrupt_Controller_Page;
    ar = 0xBB1B;               /* Assign SPI0 priority of 1 */
    io(IPR1) = ar;
    ar = 0xBBBB;  /* Assign the remainder with lowest priority */
    io(IPR0) = ar;
    io(IPR2) = ar;
    io(IPR3) = ar;

    AY0=IMASK;
    AY1=0x0020;                /* Unmask SPI Interrupt */
    AR = AY0 or AY1;
    IMASK=AR;

    RTS;

/*     SPI0 REGISTER INTIALIZATION - MASTER */

.SECTION /pm program;
SPI0_Register_Initialization:

    IOPG = SPI0_Controller_Page;
    /* Set Up Device Select Over SPI0 Interface */
    AR = 0xFF02;     /* Enable Slave On Programmable Flag Pin 2 */
    IO(SPIFLG0) = AR;     /* Write to SPI0 Flag Register */

    /* Write to SPI0 Baud rate register */
    AR = 0x1047;
    IO(SPIBAUD0) = AR;                /* SCLK0 ~= 9600 */
```

```
   /* Set up SPI0 Configuration Register */
   AR = 0x5D08;
   IO(SPICTL0) = AR;              /* Enable SPI0 as MASTER */

   RTS;

/* INITIATE AND START TRANSFER */

.SECTION /pm program;
Initiate_Transfers:
   /* Initialize DAG registers */
   I0 = TX_Buf_MASTER;         /* Pointer to TX SPI0 Buffer */
   I1 = RX_Buf_MASTER;         /* Pointer to RX SPI0 Buffer */
   M1 = 1;
   L0 = 0;                     /* Linear Addressing */
   L1 = 0;

   ENA INT;                    /* Globally enable interrupts */
   IOPG = SPI0_Controller_Page;
   AR = IO(RDBR0);         /* read from RDBR0 to start x-fer */

   CNTR = length(TX_Buf_MASTER);   /* Set up loop to transmit*/
                                   /* the entire TX buffer */

/* Loop until entire TX buffer has been received */
   DO looping until ce;
   AR = DM(I0, M1);
   IO(TDBR0) = AR;          /* Write to SPI0 Transmit Buffer */
looping:idle;               /* Wait for SPI0 Receive Interrupt */

   DIS INT;                 /* Globally disable Interrupts */
   RTS;
```

This part of the example sets up the SPI interrupt service routine.

```
#include <def2191.h>

/*      EXTERNAL DECLARATIONS */

.EXTERN Start;

     /* DM data */

.SECTION /dm data1;
.VAR counter_int5 = 0;

     /* PM Reset interrupt vector code */

.section/pm IVreset;
   jump Start;
   nop; nop; nop;

    /* SPI0 ISR */

.section / code IVint5;
   ENA SR;
   AY1 = IOPG;
   IOPG = SPI0_Controller_Page;

   AR = IO(RDBR0);               /* Interrupt counter */
                                 /* for debug purposes */
   DM(I1+= M1) = AR;

   AR = DM(counter_int5);        /* Read from SPI0 (Master) */
                                 /* Receive Buffer Register */
   AR = AR + 1;
   DM(counter_int5) = AR;
```

```
IOPG = AY1;
RTI;
```

11 UART PORT

This chapter provides the following sections:

Overview

The UART peripheral is a full-duplex, Universal Asynchronous Receiver / Transmitter compatible with the industry-standard 16450. The UART converts data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bit, and parity generation options. The UART also contains modem control and interrupt handling hardware although only the data signals TxD and RxD are routed to pins on the ADSP-2191. Interrupts may be generated from 12 unique events.

As the UART is ADSP-2191 DMA capable with support for separate TX and RX DMA master channels, the UART may be used in a programmed I/O mode or in a DMA mode of operation. I/O mode requires software management of the data flow using interrupts or polling. DMA mode requires minimal software intervention as the DMA engine itself moves the data.

Access the UART and DMA Channel registers through I/O memory space (registers). For a description of I/O memory space, see "ADSP-2191 DSP I/O Registers" on page B-1.

The UART memory map is a 16450 legacy with byte-wide registers (remapped as half words with MSByte zero-filled) and the packing of multiple registers into the same address location.

The UART's two interrupt outputs are referred to as the RX and TX interrupts. The TX interrupt is available in DMA mode only. In I/O mode, all interrupts use the RX interrupt channel.

In DMA mode, the Break and Modem status interrupts are not available.

In I/O mode, the RX interrupt is generated for the following cases:

- `RBR` full
- Receive overrun error
- Receive parity error
- Receive framing error
- Break interrupt (`RXD` held low)
- Modem status interrupt
- `THR` empty

For information on the DMA process, see "I/O Processor" on page 6-1.

Serial Communications

An asynchronous serial communication protocol is followed with these options:

- 5–8 data bits
- 1, 1½, or 2 stop bits
- None, even, or odd parity
- Baud rate = HCLK / (16 * DIVISOR), where DIVISOR = 1 to 65536

All data words require a start bit and at least one stop bit. This creates a range of 7 to bits for each word. The format of a received and transmitted character frame is controlled by the UART Line Control (LCR) register as described in Figure B-26 on page B-78. Data is transmitted and received least significant bit first.

Transmit operation is initiated by writing to the Transmit Hold (THR) register. After a synchronization delay, the data is moved to the Transmit Shift (TSR) register, where it is shifted out at a baud (bit) rate equal to HCLK / (16 * DIVISOR) with start, stop, and parity bits appended as required. All data words begin with a low-going start bit. The transfer of the THR to the TSR sets the transmit register's empty status flag.

Figure 11-1 on page 11-4 shows the physical bit stream as it could be measured on the TxD pin.

Receive operation uses the same data format as the transmit configuration except that the number of stop bits is always one. Upon detection of the start bit, the received word is shifted in the Receive Shift (RSR) register, assuming again a bit rate of HCLK / (16 * DIVISOR). After receiving the appropriate number of bits (including stop bits), the data and status are updated, and the RSR is transferred to the Receive Buffer (RBR) register.

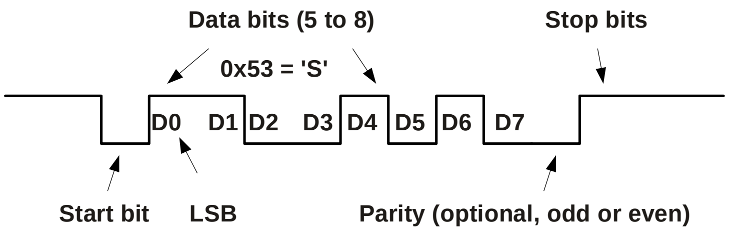


Figure 11-1. Bit Stream as Measured on TxD Pin

The receive buffer register full status flag (Data Ready `DR`) is updated after the appropriate synchronization delay and transfer of the received word to the buffer.

A sampling clock equal to 16 times the baud rate is used to sample 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 error (drift of sampling point from the center of each bit) needs to be taken into account. The sampling point is re-synchronized with each start bit so that the error only accumulates over the length of a single word. A receive filter removes spurious pulses of less than twice the sampling clock period.

The bit rate is characterized by the Peripheral Clock (`HCLK`) and 16-bit `DIVISOR`. The `DIVISOR` is split into the low significant byte (`DLL`) and the high significant byte (`DLH`). If `DLL` and `DLH` are equal to zero, a `DIVISOR` value of 65536 is assumed. As on the 16450, the divisor latch registers (`DLL` and `DLH`) are mapped to the same addresses as `RBR`, `THR`, and `IER`. The `DLAB` bit in the `LCR` register must be set before these registers can be accessed. Refer to "UART Registers" on page B-72 for details.

I/O Mode

In I/O mode, data is moved to and from the UART by the DSP's core. To transmit a character, load it into the Transmit Hold (THR) register. Received data can be read from the Receive Buffer (RBR) register.

Since the DSP architecture does not provide send and receive FIFOs, the core must write and read one character at time. To prevent data loss and misalignments of the serial data stream, the Line Status (LSR) register provides two status flags for handshaking.

The THR empty (THRE) flag is set when the THR register is ready for new data and cleared when the core loads new data in. Writing to THR when it is not empty overwrites the register by the new value and the previous character is never transmitted. The data ready (DR) flag signals when new data is available in the RBR register. This flag is cleared automatically when the core reads from RBR. Reading the RBR when it is not full means that the previously received word will be read again.

With interrupts disabled, the program might poll these status flags to determine when data is ready to move. Because polling is processor intensive, it is not typically used in real-time signal processing environments.

Alternatively, UART writes and reads may be accomplished by interrupt services routines (ISRs). Both flag pins (THRE and DR) can generate an interrupt request, when enabled in the Interrupt Enable (IER) register. Interrupts may also be masked using the IMASK register and the global interrupt enable bit in the INCTL register. In I/O mode, both interrupt sources share the same interrupt request mentioned as "UART Receive Interrupt" listed in Table B-4 on page B-26. The interrupt service routine may evaluate the STATUS bit field within the Interrupt Identification (IIR) register to determine the exact interrupt source.

Be aware that this single-character interrupt mode constrains the software to respond within a guaranteed time to prevent overrun errors from occurring in the receive channel.

DMA Mode

In the DMA mode, separate and independent RX and TX DMA channels move data between the peripheral and memory. The processor is relieved of the task of moving data and simply sets up the appropriate transfers through either the ADSP-2191 descriptor mechanism or the autobuffer mode. In DMA mode RX and TX have separate interrupt channels. The DMA interrupt mechanism works independent from the `IER` and `IIR` registers.

No additional buffering is provided in the UART's DMA channel, so the same latency requirements exist as in I/O mode. Latency, however, is determined by the bus activity and arbitration mechanism and not the processor loading and interrupt priorities.

Descriptors

DMA functionality is most often controlled by the ADSP-2191 DMA descriptor method. Each descriptor block contains all the information on a particular data movement operation as well as the pointer to the next descriptor block of information. When a descriptor block is complete, work proceeds to the next descriptor block. The data structure for a single descriptor block is illustrated in Table 11-1 on page 11-6 where HEAD refers to the address of the current descriptor:

Table 11-1. Descriptor Block Data Structure

Address	Location Name	Description
HEAD+0	DMA Configuration	Descriptor ownership and control info
HEAD+1	DMA Start Page	Page info for transfer
HEAD+2	DMA Start Addr	Address of transfer
HEAD+3	DMA Word Cnt	Number of words in transfer
HEAD+4	Next Descriptor	Pointer to address of next descriptor

Since descriptors are contained in Page 0 of internal memory, each DMA master unit must fetch this descriptor information from internal memory and update the peripheral's internal DMA registers. The dynamic allocation of descriptors is controlled by the "ownership" bit of each descriptor block (bit 15 of the configuration). Before a full descriptor block download begins, this bit is checked to determine whether the descriptor block is configured and ready for use. If it is not ready, the DMA engine goes into a wait mode until the bit is set. This "snoop" mode waits until a core write occurs to the descriptor ready register. There must be a concluding write to this register after a descriptor block is added to Page 0 to trigger the DMA hardware to reread the configuration descriptor. Once the "ownership" bit is read as a one, the remaining four words of the block are loaded. The descriptor information is then used to carry out the required data transactions.

The following illustrates the typical steps the software takes in setting up a single descriptor.

1. Memory writes HEAD+1,+2,+3,+4 to memory Page 0.
2. Memory writes HEAD+0 to memory Page 0. Bit 15 (ownership) must be set to 1.
3. I/O writes descriptor ready register of respective DMA channel (if necessary).
4. I/O writes `NXTPTR` register of DMA channel.
5. I/O writes the Configuration register, setting `DEN` high (if not already enabled)

The interrupt service routine must write a `0x01` to the `UARDT_IRQ` or `UARDR_IRQ` register to clear the DMA interrupt request. The last instance may reset the entire `UARDT_CFG` or `UARDR_CFG` register to disable the DMA.

After a work unit is complete, the DMA writes the completion status information back to the Page 0 and the descriptor's HEAD address. It also resets the ownership bit to 0, returning ownership to the processor. The software is responsible for cleaning up or reusing descriptors as desired. The management of the descriptor cleanup is typically initiated by a completion interrupt.

Autobuffer Mode

Autobuffer DMA mode can be used when simple linear or circular buffer types of transfers are required. This mode is less flexible than descriptor DMA mode but requires less software overhead and DMA bandwidth.

Most DMA registers are normally read-only, but become writable when autobuffer DMA mode is enabled. (The register section indicates these registers with an *). After configuring the DMA, the software enables the DMA with the `DMAEn` control bit. DMA operation proceeds as described. When the count reaches its endpoint, the start address and count are reset to their original value, effectively creating a circular buffer. If enabled, interrupts are generated at the halfway point (rounded down) and the completion point of the transfer (for example: an 11 word transfer would generate interrupts after the 6th and 11th transfers).

The following example illustrates typical autobuffer DMA mode use:

- I/O writes config register setting only the `DAUTO` bit high.
- I/O writes page, start, and count values.
- I/O writes config again, setting direction and sets `DEN` high.

Transfers can be aborted at any point by setting `DEN` low.

Mixing Modes

The I/O mode and the DMA modes use different synchronization mechanisms. This means that serial communication should be completed before the program can switch from I/O mode to DMA mode or vice versa. Before setting up a DMA transmission, ensure that the Transmit Hold (THR) register and the Transmit Send (TSR) register are empty by polling the THRE bit and the TEMT bit in the Line Status (LSR) register. In the opposite case, wait until the two bits of the DMA buffer status in the UARDT_CFG register are cleared.

Code Examples

The following sections provide code examples.

- "Initializing the UART" on page 11-10
- "Polling the TX Channel" on page 11-10
- "Interrupt Controlled Transmission" on page 11-11
- "Using Descriptor DMA on the UART TX Channel" on page 11-12
- "Setting Up Autobuffer DMA on the UART TX Channel" on page 11-14
- "Auto-Baud Rate Detection Using Timer 0" on page 11-15

Initializing the UART

The following code example initializes the UART assuming a HCLK clock of 16 MHz. The serial format is 8 bits, 9600 bps, no parity, and one stop bit.

```
#include <def2191.h>
#define DLAB 0x80          /* divisor latch access bit in LCR */
#define WLS8CHAR 0x03      /* Word length of 8 Bits */

                   /* UART registers are mapped to IO page 5 */
IOPG = UART_Controller_Page;

           /* To access divisor latch registers set DLAB bit */
AR = DLAB; IO(LCR) = AR;

                  /* 16MHz / 16 / 9600 =104d =68h */
AR = 0x68; IO(DLL) = AR;
AR = 0x00; IO(DLH) = AR;

               /* 8 bit, no parity, one stop bit, clear DLAB */
AR = WLS8CHAR; IO(LCR) = AR;
```

Polling the TX Channel

The following example code demonstrates how polling can be implemented on the TX channel. A real-world implementation might require additional time-out functionality.

```
#include <def2191.h>
#define THRE 0x20 /* THR empty */

                /* AR holds the next value to transmit */
putc:
                /* UART registers are mapped to IO page 5 */
```

```
   IOPG = UART_Controller_Page;

                /* wait until Transmit Hold register is ready */
poll_thre:
   AX0 = IO(LSR);
   AF = AX0 AND THRE;
   IF EQ JUMP poll_thre;

                /* finally send next value */
   IO(THR) = AR;
   RTS;
```

Interrupt Controlled Transmission

To prepare interrupt controlled transmission, use the following lines. In I/O mode, the TX channel signals interrupt requests to the RX interrupt. By default, the RX interrupt uses IRQ 11 as assumed in this example.

```
#include <def2191.h>
#define ETBEI 0x02            /* Enable TX Interrupt */

                  /* UART registers are mapped to IO page 5 */
IOPG = UART_Controller_Page;

                  /* load first value manually */
AR = DM(first_value);
CALL putc;

              /* enable TX source on RX interrupt using IRQ 11 */
AR = ETBEI; IO(IER) = AR;

                /* clear pending RX requests and enable IRQ 11*/
IRPTL = 0x0000;
NOP;
```

```
IMASK = 0x0800;
                /* enable Interrupt globally */
ENA INT;
```

Using Descriptor DMA on the UART TX Channel

The following example illustrates a typical setup of a descriptor DMA on the UART TX channel. Two strings will be sent out. Once the first one has been completed, the descriptor block of the second is activated without core activity. An interrupt is generated after the second (and last) has been shifted out. Note that strings are usually null-terminated by convention (C-style). The trailing null character is not used in DMA mode.

```
#include <def2191.h>

.section /dm dmdata;

.var sText1 [] = 'Test String 1',13,10,0;
.var sText2 [] = 'Test String 2',13,10,0;

               /* descriptor blocks must be placed in page 0 */
.var tcbTX1 [5] =
   0x8001,            /* ownership and enable */
   PAGE(sText1),      /* start page */
   sText1,            /* start address */
   LENGTH(sText1)-1,  /* number of bytes */
   tcbTX2;            /* next descriptor block */

.var tcbTX2 [5] =
   0x8005,            /* interrupt on completion */
   PAGE(sText2),      /* start page */
   sText2,            /* start address */
   LENGTH(sText2)-1,  /* number of bytes */
   0;                 /* don 't care if ISR disables DMA */
```

```
.section /pm program;

IOPG = UART_Controller_Page;

    /* enable TX interrupt on IRQ 12 as assigned by default */
IRPTL = 0x0000;
NOP;
IMASK = 0x1000;
ENA INT;

                 /* first disable DMA and Autobuffer mode */
AR = 0x0000; DM(UARDT_CFG) = AR;

                 /* load address of first descriptor block */
AR = tcbTX1; IO(UARDT_CP) = AR;

                 /* signal Descriptor Block ready */
AR = 0x0001; IO(UARDT_CPR) = AR;

                  /* start DMA by setting Enable Bit */
AR = 0x0001; IO(UARDT_CFG) = AR;
```

This example assumes that the DMA is disabled by the interrupt service routine invoked when the last character of `sText2` has been loaded by the DMA engine. Alternatively, the next descriptor pointer of the last descriptor in the chain (`tcbTX2` in this example) may point to any page 0 address that contains a zero value. Then, the DMA is stopped automatically, because the additional TCB load clears the DMA enable bit.

Note that the ownership bit in the descriptor is set. Doing so, the program might poll the ownership bit within `tcbTX2` in memory to determine the completion of the DMA transfer without having interrupts enabled.

Setting Up Autobuffer DMA on the UART TX Channel

The following example illustrates a typical setup of an autobuffer DMA on the UART TX channel. The TX interrupt is requested twice. Without any additional software flag, the interrupt service routine cannot determine whether it was invoked by a halfway request or a completion request.

```
#include <def2191.h>

.section /dm dmdata;
.var sText [] = 'Hello World',13,10,0;

.section /pm program;
IOPG = UART_Controller_Page;

      /* set to autobuffer mode -make DMA registers writeable */
AR = 0x0010; IO(UARDT_CFG) = AR;

      /* set start page and address */
AR = PAGE(sText); IO(UARDT_SRP) = AR;
AR = sText; IO(UARDT_SRA) = AR;

      /* set number of characters without terminating zero */
AR = LENGTH(sText)-1;   IO(UARDT_CNT) = AR;

      /* enable DMA, Autobuffer Mode, Interrupt on completion */
AR = 0x0015; IO(UARDT_CFG) = AR;

      /* finally enable IRQ 12 -TX is assigned to by default */
IRPTL = 0x0000;
NOP;
IMASK = 0x1000;
ENA INT;
```

Auto-Baud Rate Detection Using Timer 0

In WIDTH_CNT mode, the three general-purpose timers can disconnect their input from the TMRx pin and connect it to the UART's RXD pin. This enables glueless auto-baud support.

In the following example, the DSP waits after reset until it detects a start condition on the RXD pin. By convention, it assumes that the first received data is an "@" character (ASCII 0x40) in an 8-bit format and the parity bit is disabled. In this way, the DSP can measure the period from falling edge to falling edge and divide this period by the number of contained bits.

Measuring periods is safer than measuring pulses.

Use the following formula to determine the proper value for the divisor latch registers:

$$\text{Divisor Latch} = \frac{\text{Period Count}}{16 \times \text{Number of Bits}}$$

Period count is composed of two 16-bit registers (T_PRDH0 and T_PRDL0). Using the "@" character, the number of bits is eight. The divide operation can be performed by a right shift. The two 8-bit registers (DLH and DLL) form the divisor latch.

```
#include <def2191.h>

.section / code program;
_autobaud:

    /* *** Config Timer 0  *** */

    IOPG = Timer_Page;

    /* Low-Pulse / Period Capture Mode on Auxiliary / Ena Int */
    AR = 0x003A;    IO(T_CFGR0) = AR;
    /* clear IRQ latch */
    AR = 0x0001;    IO(T_GSR0) = AR;

    /* enable timer 0 */
    AR = 0x0100;    IO(T_GSR0) = AR;

    AY0 = 0x1;

    /* poll Timer 0 interrupt latch TIMIL0 bit */
xwait:
    AX0 = IO(T_GSR0);
    AX0 and AY0;
    IF EQ JUMP xwait;

    /* disable timer 0 */
    AR = 0x0200;    IO(T_GSR0) = AR;

    /* expecting 0x40 @ character */
    /* frame looks like 1111100000001011 */

    AX1 = IO(T_PRDH0);
    AX0 = IO(T_PRDL0);

    /* need to divide by 8 bits and additionally by 16 */
    /* equals right shift of 7 bit positions */

    SR = LSHIFT AX0 BY -7 (LO);
    SR = SR OR LSHIFT AX1 BY -7 (HI);

    /* sr0 contains the 16-bit DL value now */
    IOPG = UART_Controller_Page;

    /* set DLAB bit */
    AY0 = 0x83;
```

```
IO(LCR) = AY0;

/* break down into bytes */
AY0 = 0xFF;
AR = SR0 AND AY0;
SR = LSHIFT SR0 BY -8 (LO);

IO(DLL) = AR;
IO(DLH) = SR0;

/* clear DLAB bit */
AR = 0x03;
IO(LCR) = AR;
```

12 TIMER

This chapter includes these sections:

Overview

The ADSP-2191 features three identical 32-bit timers; each timer can be individually configured in any of three modes:

- "Pulsewidth Modulation (PWMOUT) Mode" on page 12-6 (PWMOUT)
- "Pulsewidth Count and Capture (WDTH_CAP) Mode" on page 12-11 (WDTH_CAP)
- "External Event Watchdog (EXT_CLK) Mode" on page 12-14 (EXT_CLK)

Each timer has one dedicated bi-directional pin (TMRx), which functions as an output pin in PWMOUT mode and as an input pin in WDTH_CAP mode and EXT_CLK mode. To provide these functions, each timer has seven, 16-bit registers. For range and precision, six of these registers can be paired (high/low) to allow for 32-bit values; see Figure 12-1 on page 12-2.

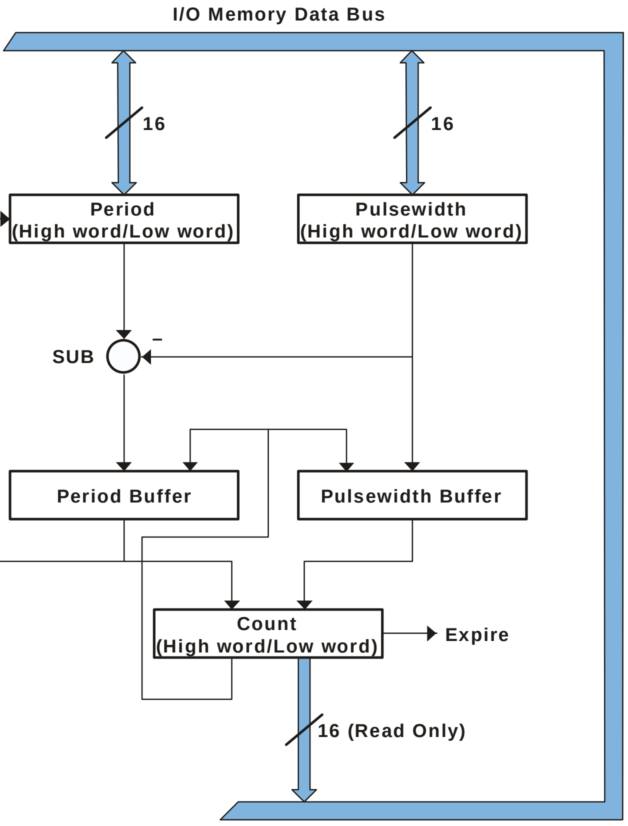

I/O Memory Data Bus
16
16
Period (High word/Low word)
Pulsewidth (High word/Low word)
SUB
–
Period Buffer
Pulsewidth Buffer
Count (High word/Low word)
Expire
16 (Read Only)

Figure 12-1. Timer Block Diagram

The registers for each timer are:

- Timer x Configuration (`T_CFGRx`)
- Timer x High Word Count (`T_CNTHx`)
- Timer x Low Word Count (`T_CNTLx`)
- Timer x High Word Period (`T_PRDHx`)
- Timer x Low Word Period (`T_PRDLx`)

- Timer x High Word Pulsewidth (T_WHRx)
- Timer x Low Word Pulsewidth (T_WLRx)

Because the paired “counter” registers operate as a single value, the timer counters are 32-bits wide. When clocked internally, the clock source is the ADSP-2191’s peripheral clock (HCLK). Assuming the peripheral clock is running at 80 MHz, the maximum period for the timer count is ((232-1) * 12.5 ns) = 53.69 seconds.

The Timer Global Status and Control (T_GSRx) registers indicate the status of all three timers, requiring a single read to check the status of all three timers. Each T_GSRx register contains timer enable bits (T_GSR0 enables TIMER0, and so on) to enable the corresponding timer. Within T_GSRx, each timer has a pair of “sticky” status bits that require a “write-one-to-set” (TIMENx) or “write-one-to-clear” (TIMDISx)—see Table 12-1 on page 12-4—to enable or disable the timer. Writing a one to both bits of a pair disables that timer.

After enabling a timer, its TIMENx and TIMDISx bits are set (=1). The timer starts counting three peripheral clock cycles after the TIMENx bit is set. Setting (writing 1 to) the timer’s TIMDISx bit stops the timer without waiting for any additional event.

Each T_GSRx register also contains an interrupt latch bit (TIMILx) and an overflow/error indicator bit (OVF_ERRx) for each timer. These “sticky” bits are set by the timer hardware and may be watched by software. They need to be cleared in each timer’s corresponding T_GSRx register by software explicitly. To clear, write a “one” to the corresponding bit.

Interrupt and overflow bits may be cleared simultaneously with timer enable or disable.

To enable a timer’s interrupts, set the IRQ_ENA bit in the Timer Configuration (T_CFGRx) register and unmask the timer’s interrupt by setting the corresponding bit of the IMASK register. With the IRQ_ENA bit cleared, the timer does not set its 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, ensure that the interrupt service routine clears the TIMILx latch before the RTI instruction to assure that the interrupt is not re-issued. In external clock (EXT_CLK) mode, reset the latch at the beginning of the interrupt routine to not miss any timer event. To enable timer interrupts, set the IRQ_ENA bit in the proper Timer Configuration (T_CFGRx) register.

Table 12-1. Timer Global Status and Control (T_GSRx) Register Bits

Bit(s)	Name	Definition
0	TIMIL0 Timer 0 Interrupt Latch	Write one to clear (also an output)
1	TIMIL1 Timer 1 Interrupt Latch	Write one to clear (also an output)
2	TIMIL2 Timer 2 Interrupt Latch	Write one to clear (also an output)
3	Reserved	
4	OVF_ERR0 Timer 0 Overflow/Error	Write one to clear (also an output)
5	OVF_ERR1 Timer 1 Overflow/Error	Write one to clear (also an output)
6	OVF_ERR2 Timer 2 Overflow/Error	Write one to clear (also an output)
7	Reserved	
8	TIMEN0 Timer 0 Enable	Write one to enable Timer 0
9	TIMDIS0 Timer 0 Disable	Write one to disable Timer 0
10	TIMEN1 Timer 1 Enable	Write one to enable Timer 1
11	TIMDIS1 Timer 1 Disable	Write one to disable Timer 1
12	TIMEN2 Timer 2 Enable	Write one to enable Timer 2
13	TIMDIS2 Timer 2 Disable	Write one to disable Timer 2
14 - 15	Reserved	

To enable an individual timer, set the timer's TIMEN bit in the corresponding T_GSRx register. To disable an individual timer, set the timer's TIMDIS bit in the corresponding T_GSRx register. To enable all three timers in parallel, set each TIMEN bit in each timer's corresponding T_GSRx register.

Before enabling a timer, always program the corresponding timer's Configuration (T_CFGRx) register. This register defines the timer's operating mode, the polarity of the TMRx pin, and the timer's interrupt behavior. Do not alter the operating mode while the timer is running. For more information on the T_CFGRx register, see Figure B-34 on page B-92.

Timer enable/disable timing appears in Figure 12-2 on page 12-5.

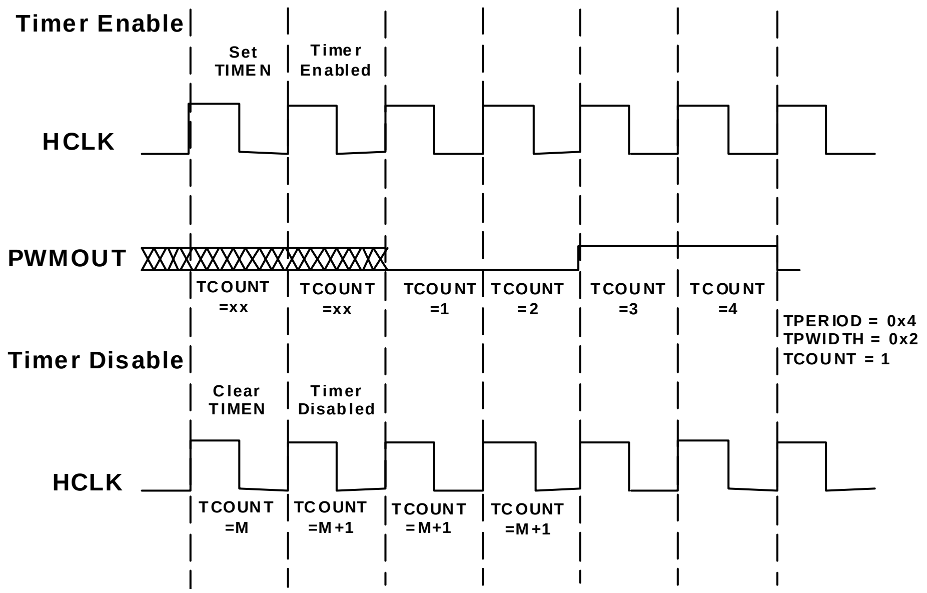


Figure 12-2. Timer Enable and Disable Timing

Because the timers are 32 bits, hardware support guarantees that high and low words are always coherent whenever the DSP accesses period or pulse-width registers. There is no similar support for DSP reads of the counter register itself. When a coherent read of the counter register's high and low words is needed, software should stop (disable) the timer before reading the 32-bit counter value.

When the timer is disabled, the counter registers retain their state. When the timer is re-enabled, the counter is re-initialized based on the operating mode. The counter registers are read-only. The software cannot overwrite or preset the counter value directly.

Any of the timers can be used to implement a watchdog functionality, which might be controlled by an internal or an external clock source.

For software to service the watchdog, disable the timer and re-enable it again. This resets the timer value. Servicing the watchdog periodically prevents the count register from reaching the period value and prevents the timer interrupt from being generated. Assign a very high interrupt priority to this watchdog timer. When the timer reaches the period value and generates the interrupt, reset the DSP within the corresponding watchdog's interrupt service routine.

Pulsewidth Modulation (PWMOUT) Mode

Setting the TMODE field to 01 in the Timer's Configuration (T_CFGRx) register enables PWMOUT mode. In PWMOUT mode, the timer's TMRx pin is an output. It is actively driven as long as the TMODE field remains 01.

The timer is clocked internally by HCLK. Depending on the PERIOD_CNT bit, PWMOUT mode generates pulsewidth modulation waveforms or generates a single pulse on the TMRx pin.

After setting TMODE to 01 but before enabling the timer, set the width and period registers to proper values. There are shadow registers for the T_PRDHx, T_PRDLx, and T_WHRx registers. A write to the T_WLRx register triggers the shadow registers to update the T_PRDHx, T_PRDLx, and T_WHRx values. This guarantees coherency between all four registers.

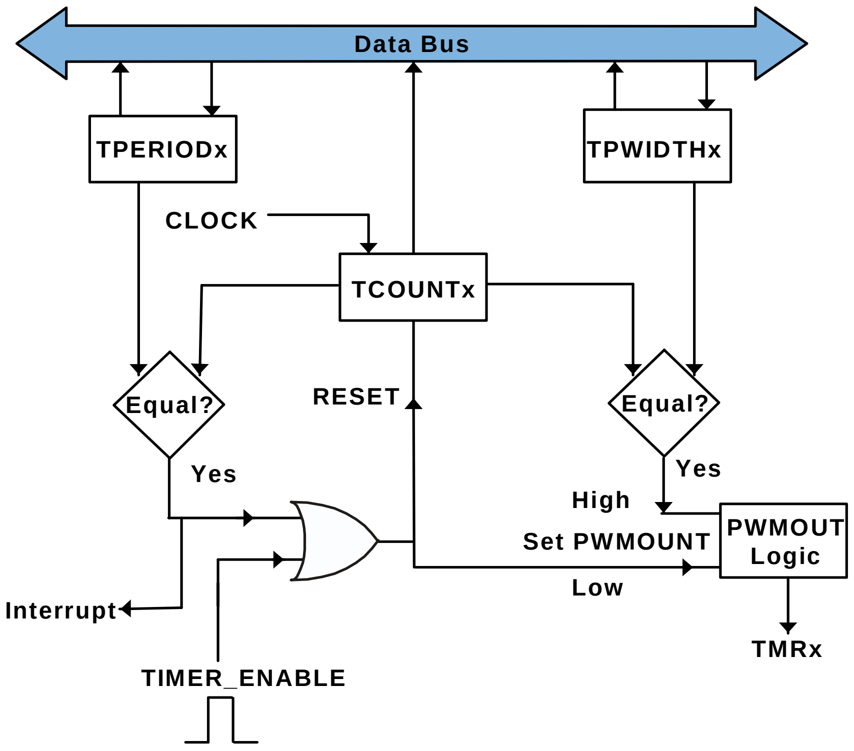


Figure 12-3. Timer Flow Diagram - PWMOUT Mode

When the timer is enabled, the timer checks the period and width values for plausibility (independent of `PERIOD_CNT`) and does *not* start to count when any of the following conditions is true:

- Width equals to zero
- Period value is lower than width value
- Width equals period

The timer module tests these conditions on writes to the `T_WLRx` register. Before writing to `T_WLRx`, ensure that `T_WHRx` and the period registers are set accordingly.

On invalid conditions, the timer sets the OVF_ERRx bit and the TIMILx bit after two HCLK cycles. The count register is not altered. Note that after reset, the timer registers are all zero.

If period and width values are valid after enabling, the count register is loaded with the value 0xFFFF FFFF – width. The timer counts upward to 0xFFFF FFFE. Instead of incrementing to 0xFFFF FFFF, the timer reloads the counter with 0xFFFF FFFF – (period – width) and repeats.

In PWMOUT mode, the TMRx pin is driven low when the timer is disabled, regardless of the state of the PULSE_HI bit. When the timer is running, however, the TMRx pin polarity corresponds to the PULSE_HI bit setting.

PWM Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with a well defined period and duty cycle. This mode also generates periodic interrupts for real-time DSP processing.

The 32-bit Period (T_PRDHx / T_PRDLx) and Width (T_WHRx / T_WLRx) registers are programmed with the values of the timer count period and pulsewidth modulated output pulsewidth.

When the timer is enabled in this mode, the TMRx pin is pulled to a de-asserted state each time the pulsewidth expires, and the pin is asserted again when the period expires (or when the timer is started).

To control the assertion sense of the TMRx pin, the PULSE_HI bit in the corresponding T_CFGRx register is cleared or set (clearing causes a low assertion level, setting causes a high assertion level).

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine must clear the interrupt latch bit (TIMILx) and may alter the period and/or width values. In pulsewidth modulation applications, the software needs to update period and pulsewidth value while the timer is running. To guarantee coherency between both the high and low words and between period and pulsewidth registers, a double buffer mechanism is in place.

DSP core writes to the T_PRDHx, T_PRDLx, and T_WHRx registers do not become active until the DSP core writes to the T_WLRx register. If the software would like to update one of these three registers only, it must rewrite the T_WLRx register afterward. When the T_WLRx value is not subject to change, the interrupt service routine may just read back the current value of the T_WLRx register and rewrite it again. On the next counter reload, all four registers are available to the timer.

In this mode, the counter is reloaded at the end of every period as well as at the end of every pulse. The generated waveform depends on whether T_WLRx is updated before or after the pulse width expires, due to the reload sequence previously described.

You can alter the pulse width on-the-fly while the timer is running by having an interrupt service routine write new values to the width registers. As illustrated in Figure 12-4 on page 12-9, one erroneous period is generated when the write to the TIMERx_WIDTH_LO register occurs before the on-going pulse width expires. This is very likely because the interrupt is requested at the end of a period.

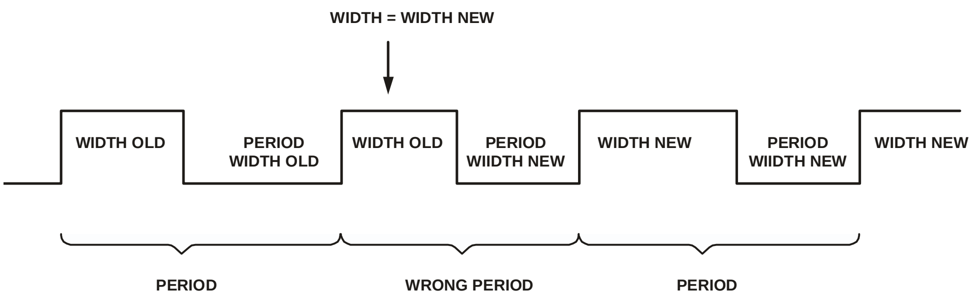


Figure 12-4. Possible Period Failure Due to On-the-Fly Width Update

If an application forbids single misaligned PWM patterns, use the procedure in Figure 12-4, which alters the period value temporary and restores the original period value at the very next PWM cycle to obtain constant PWM periods.

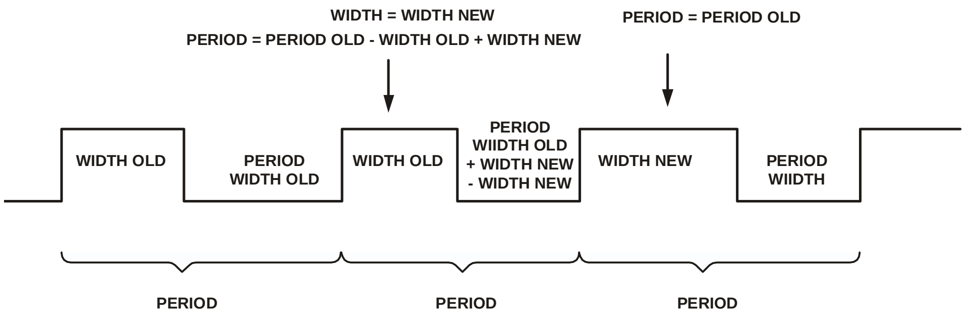


Figure 12-5. Recommended On-the-Fly Width Update Procedure

Period settings can be altered without similar impacts.

To generate the maximum frequency on the TMRx output pin, set the period value to 2 and set the pulsewidth to 1. This makes TMRx toggle each HCLK clock, producing a 50% duty cycle.

Single-Pulse Generation

If the PERIOD_CNT bit is cleared, PWMOUT mode generates a single pulse on the TMRx pin. This mode also can be used to implement a well-defined software delay often required by state-machines and so on. The pulsewidth is defined by the width register, and the period register is not used.

At the end of the pulse, the interrupt latch bit (TIMIRQx) is set and the timer is stopped automatically. Always set the PULSE_HI bit in single-pulse mode to generate an active-high pulse. Active-low pulses are not recommended in this mode because the TMRx pin drives low when the timer is not running.

Pulsewidth 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 period and pulsewidth of externally applied rectangular waveforms. Setting the TMODE field to 10 in the T_CFGRx register enables this mode. The period and width registers are read-only in WIDTH_CNT mode.

When enabled in this mode, the timer resets words of the count in the T_CNTHx and T_CNTLx registers to 0x0000 0001 and does not start counting until detecting the leading edge on the TMRx pin.

When the timer detects a first leading edge, it starts incrementing. When it detects the trailing edge of a waveform, the timer captures the current 32-bit value of the T_CNTHx and T_CNTLx count registers to the T_WHRx and T_WLRx width registers. At the next leading edge, the timer transfers the current 32-bit value of the T_CNTHx and T_CNTLx count registers to the T_PRDHx and T_PRDLx period register. The count registers are reset to 0x0000 0001 again, and the timer continues counting until it is disabled or the count value reaches 0xFFFF FFFF.

In this mode, software can measure the pulsewidth 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 T_CFGRx register is set or cleared. When the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the count register is captured to the width register on the rising edge and the period is captured on the next falling edge.

The PERIOD_CNT bit in the T_CFGRx register controls whether an enabled interrupt is generated when the pulsewidth or pulse period is captured. When the PERIOD_CNT bit is set, the interrupt latch bit (TIMILx) gets set when the pulse period value is captured. If the PERIOD_CNT bit is cleared, the TIMILx bit gets set when the pulse width value is captured.

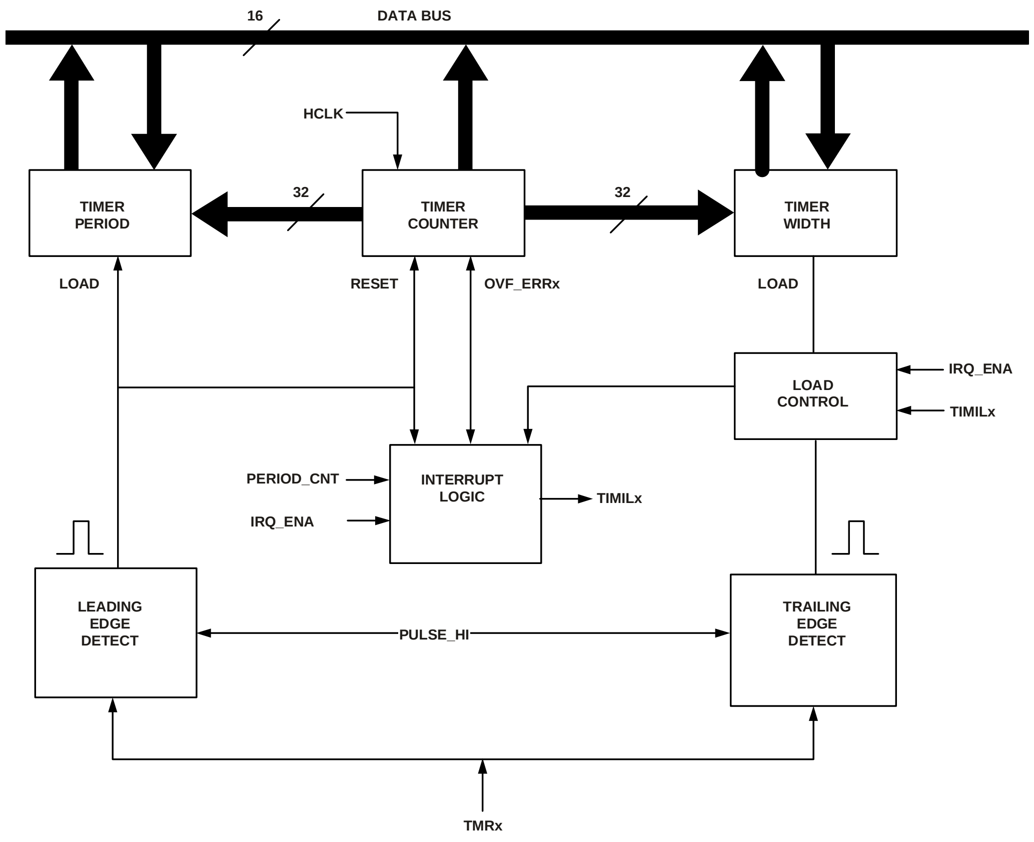


Figure 12-6. Timer Flow Diagram - WDTH_CAP Mode

If the `PERIOD_CNT` bit is cleared, the first period value has not yet been measured when the first interrupt is generated, so the period value is not valid. If the interrupt service routine reads the period value anyway, the timer returns a period zero value.

With the IRQ_ENA bit set, the width registers become sticky in WDTH_CAP mode. Once a pulsewidth event (trailing edge) has been detected and properly latched, the width registers do not update unless the IRQx bit is cleared by software. The period registers still update every time a leading edge is detected.

A timer interrupt (if enabled) is also generated when the count register reaches a value of 0xFFFF FFFF. At that point, the timer is disabled automatically, and the OVF_ERRx status bit is set, indicating a count overflow. TIMILx and OVF_ERRx are sticky bits, and software has to explicitly clear them.

The first width value captured in WDTH_CAP mode is erroneous due to synchronizer latency. To avert this error, software must issue two NOP instructions between setting WDTH_CAP mode and setting TIMEN. TIMEN is set subsequently without NOP instructions.

Auto-Baud Mode

Any one of the three timers may provide auto-baud detection for the UART port. The timer input select (TIN_SEL) bit in the T_CFGRx register causes the timer to sample the UART port receive data (RXD) pin instead of the TMRx pin while enabled for WDTH_CAP mode. A software routine can detect the pulse widths of serial stream bit-cells. Because the sample base of the timers are synchronous with the UART operation—all derived from the PLL clock—the pulse widths can be used to calculate the baud rate divider for the UART. If the pulsewidth of a single bit is captured, use the following formula:

BAUDDIV = (width registers) / (16 X number of captured bits)

For an auto-baud example that captures the period rather than the pulsewidth, see “Auto-Baud Rate Detection Using Timer 0” on page 11-15.

External Event Watchdog (EXT_CLK) Mode

In EXT_CLK mode, the TMRx pin is an input. The timer works as a counter clocked by any external source, which can also be asynchronous to the DSP clock. Setting the TMODE field to 11 in the T_CFGRx register enables this mode. The T_PRDHx and T_PRDLx period registers are programmed with the value of the maximum timer external count.

After the timer has been enabled, it waits for the first rising edge on the TMRx pin. This edge forces the count register to be loaded by the value 0xFFFF FFFF – Period. Every subsequent rising edge increments the count register. After reaching the count value 0xFFFF FFFE, the TIMILx bit is set and an interrupt is generated. The next rising edge reloads the count register again by 0xFFFF FFFF – Period.

The TIN_SEL, PULSE_HI, and PERIOD_CNT configuration bits do not affect this mode. Also, OVF_ERRx is never set. The width register is not used.

In this mode, an external clock source can use the timer to wake up the DSP from the sleeping mode even if HCLK has been stopped.

Code Examples

This section shows how to set up the timer. In PWMOUT mode, when Timer0 width expires, the counter is loaded with (period – width) and continues counting. When period expires, the counter is loaded with the width value again and the cycle repeats. The TMRx pin is alternately driven high/low, determined by PULSE_HI, at each zero. When the width or period expires, TIMIL0 (if enabled) is set depending on PERIOD_CNT bit in T_CFGR0.

Timer Example Steps

TIMER0 is set up in PWMOUT mode. It is intended to toggle general-purpose I/O's (GPIO) ON/OFF inside Timer0 interrupt service routine at a 1-Hz rate. This is done assuming a 160-MHz core clock (CCLK) and an 80-MHz peripheral clock (HCLK).

Prior to initializing or re-configuring the timer, it is best to reset TIMEN. Because the intended mode of operation in this example is PWMOUT, software sets the TMODE field to 01 in the T_CFGR0 register to select PWM_OUT operation. As a result, this configures TMRx pin as an output pin with its polarity determined by PULSE_HI.

- 1 generates a positive active width pulse waveform at the TMRx pin
- 0 generates a negative active width pulse waveform at the TMRx pin

This polarity is dependent on the application, but in this example, it is set to be positive active width pulse. As well, initialize to generate a PWMOUT output, and enable Timer0 interrupt requests.

```
AX0 = 0x001D;
    /* PWM_OUT mode, Positive Active Pulse, Count to end of */
    /* period, Int Request Enable, Timer_pin select */
IO(T_CFGR0) = AX0;
```

Next, initialize the period and width register values. Update the high-low period values first. Once the period value has been updated, update the high-word width value followed by the low-word width value. Updating the low-word width value actually transfers the period and width values to their respective Buffers.

Ensure that the period is greater than the width value.

```
AX0 = 0x0262;
IO(T_PRDH0) = AX0;
```

```
            /* Timer 0 Period register (high word) */
AX0 = 0x5A00;
IO(T_PRDL0) = AX0;
            /* Timer 0 Period register (low word) */
AX0 = 0x0131;
IO(T_WHR0) = AX0;
            /* Timer 0 Width register (high word) */
AX0 = 0x2D00;
IO(T_WLR0) = AX0;
            /* Timer 0 Width register (low word) */
```

Because TIMEN0 is sticky, enabling Timer0 requires that a 1 be written to bit 8 of the Timer 0 Global Status and Sticky (T_GSR0) register. The timer starts three cycles after software enables it. During those three seconds, the timer performs boundary exception checks for the following period and width values:

- If (width = 0, or period < width, or period = width) both OVF_ERR and TIMILx are set.
- If there are no exceptions, the width value is loaded in counter and it starts counting.

Writing bit 9 of T_GSR0 disables Timer0. When disabled, the counter and other registers retain their state. When the timer is re-enabled, the buffers and counter are re-initialized from the period/width registers based on the TMODE field in the T_CFGR0 register.

```
AX0 = 0x0100;
             /* Enable Timer0 */
IO(T_GSR0) = AX0;
```

Lastly, enable global interrupts.

```
ENA INT;
            /*Enable Interrupts */
RTS;
```

The PFx pins are toggled on/off inside the Timer0 interrupt service routine. The interrupt is generated when the period count expires.

```
AX0 = 0x000F;
AR = 0;
IOPG = General_Purpose_IO;
AX1 = DM(Timer__Flag_Polarity);
AR = TGLBIT 0x0 OF AX1;
            /* Toggle Status flag */
IF EQ JUMP Turn_Off;
            /* Determine whether GPIO was ON or OFF */

Turn_On:
IO(FLAGS) = AX0;
            /* Turn ON GPIOS 0, 1, 2, and 3 */
DM(Timer__Flag_Polarity) = AR;
IOPG = AY1;
DIS SR;
RTI;

Turn_Off:
IO(FLAGC) = AX0;
            /* Turn OFF GPIOS 0, 1, 2, and 3 */
DM(Timer__Flag_Polarity) = AR;
IOPG = AY1;
DIS SR;
RTI;
```

In the sections that follow, the code illustrates the Timer0 initialization and operation for the ADSP-2191 DSP.

Timer0 Initialization Routine

This example shows the initialization code for Timer0. This routine is intended for use with the ADSP-2191 EZ-KIT Lite evaluation system.

```
#include <def2191.h>

/* GLOBAL DECLARATIONS */
.GLOBAL    _main;
.GLOBAL    Start;

/* Program memory code */

.SECTION /pm program;
Start:
_main:
   call Program_Timer_Interrupt;
            /* Initialize Interrupt Priorities */
   call General_Purpose_Intitialization;
            /* Initialize General Purpose I/O */
   call Timer_register_Initialization;
            /* Initialize Timer0 */

wait_forever:
   NOP;
   NOP;
   NOP;
   NOP;
   JUMP wait_forever;

/* INTERRUPT PRIORITY CONFIGURATION */

.SECTION /pm program;
Program_Timer_Interrupt:
      IOPG = 0;
      AR=IO(SYSCR);     /* Map Interrupt Vector Table to Page 0*/
      AR = SETBIT 4 OF AR;
      IO(SYSCR) = AR;
   DIS INT;      /* Disable all interrupts */
      IRPTL = 0x0;   /* Clear all interrupts */
      ICNTL = 0x0;   /* Interrupt nesting disable */
      IMASK = 0;   /* Mask all interrupts */
      IOPG = Interrupt_Controller_Page;
   AR = 0xBB1B;   /* Assign Timer0 with priority of 1 */
   IO(IPR2) = AR;
   AR = 0xBBBB;   /* Assign remainder with lowest priority */
   IO(IPR0) = AR;
   IO(IPR1) = AR;
   IO(IPR3) = AR;
   AY0 = IMASK;
   AY1 = 0x0020;    /* Unmask Timer0 Interrupt */
   AR = AY0 OR AY1;
   IMASK = AR;
   RTS;

/* INITIALIZE GENERAL PURPOSE FLAGS */

.SECTION /pm program;
General_Purpose_Intitialization:
   IOPG = General_Purpose_IO;
   AY0 = IO(DIRS);
   AX0 = 0x000F;   /* Configure FLAGS 0, 1, 2, & 3 as outputs */
   AR = AX0 OR AY0;
   IO(DIRS) = AR;
   AX1 = 0x000F;   /* Turn OFF FLAGS 0, 1, 2, and 3 */
   IO(FLAGC) = AX1;
   AX1 = 0x000F;   /* Turn ON FLAGS 0, 1, 2, and 3 */
   IO(FLAGS) = AX1;
   RTS;
```

```
/* TIMER REGISTER INTIALIZATION */

.SECTION /pm program;
Timer_register_Initialization:
   IOPG = Timer_Page;
   AX0 = 0x001D;
      /* PWM_OUT mode, Positive Active Pulse, Count to end of */
   IO(T_CFGR0) = AX0;
   /* period, Int Request Enable, Timer_pin select */
   AX0 = 0x0262;
   IO(T_PRDH0) = AX0;
   /* Timer 0 Period register (high word) */
   AX0 = 0x5A00;
   IO(T_PRDL0) = AX0;
   /* Timer 0 Period register (low word) */
   AX0 = 0x0131;
   IO(T_WHR0) = AX0;
   /* Timer 0 Width register (high word) */
   AX0 = 0x2D00;
   IO(T_WLR0) = AX0;
   /* Timer 0 Width register (low word) */
   AX0 = 0x0100;   /* Enable Timer0 */
   IO(T_GSR0) = AX0;
   ENA INT;   /* Globally Enable Interrupts */
   RTS;
```

Timer Interrupt Service Routine

This example shows a timer interrupt service routine. This example is intended for use with the ADSP-2191 EZ-KIT Lite evaluation system.

```
#include <def2191.h>

/* EXTERNAL DECLARATIONS */
```

```
.EXTERN Start;
/* DM data */

.SECTION /dm data1;
.VAR counter_int5 = 0;
.VAR Timer__Flag_Polarity;

/* PM Reset interrupt vector code */

.section/pm IVreset;
JUMP Start;
NOP; NOP; NOP;

/* Timer ISR*/

.section/pm IVint5;
ENA SR;
AY1 = IOPG;
IOPG = Timer_Page;
AX0 = 0x0001;
/* Clear Timer0 TIMIL0 */
IO(T_GSR0) = AX0;

AR = DM(counter_int5);
/* Interrupt counter */
AR = AR + 1;
DM(counter_int5) = AR;

Timer0_Interrupt_Handler:
AX0 = 0x000F;
AR = 0;
IOPG = General_Purpose_IO;
AX1 = DM(Timer__Flag_Polarity);
```

```
AR = TGLBIT 0x0 OF AX1;
/* Toggle Status flag */
IF EQ JUMP Turn_Off;
/* Determine whether GPIO was ON or OFF */

Turn_On:
IO(FLAGS) = AX0;
/* Turn ON GPIOS 0, 1, 2, 3 */
DM(Timer__Flag_Polarity) = AR;
IOPG = ay1;
DIS SR;
RTI;

Turn_Off:
IO(FLAGC) = AX0;
/* Turn OFF GPIOS 0, 1, 2, and 3 */
DM(Timer__Flag_Polarity) = AR;
IOPG = AY1;
DIS SR;
RTI;
```

13 JTAG TEST-EMULATION PORT

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial Test Access Port (TAP).

This chapter provides the following topics:

Overview

The ADSP-2191 DSP contains a test access port compatible with the industry-standard IEEE 1149.1 (JTAG) specification. Only the IEEE 1149.1 features specific to the ADSP-2191 are described here. For more information, see the IEEE 1149.1 specification and other documents listed in "References" on page 13-5.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-2191 DSP. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (`TCK`) is asynchronous to the ADSP-2191 system clock (`CLKIN`).

JTAG Test Access Port

The emulator uses JTAG boundary scan logic for ADSP-2191 communications and control. This JTAG logic consists of a state machine, a five pin Test Access Port (TAP), and shift registers. The state machine and pins conform to the IEEE 1149.1 specification. The TAP pins appear in Table 13-1 on page 13-3.

Table 13-1. JTAG Test Access Port (TAP) Pins

Pin	Function
TCK	(input) Test Clock: pin used to clock the TAP state machine.[1]
TMS	(input) Test Mode Select: pin used to control the TAP state machine sequence.1
TDI	(input) Test Data In: serial shift data input pin.
TDO	(output) Test Data Out: serial shift data output pin.
$\overline{\texttt{TRST}}$	(input) Test Logic Reset: resets the TAP state machine

1 Asynchronous with CLKIN

A Boundary Scan Description Language (BSDL) file for the ADSP-2191 is available on Analog Devices' Web site. Set your browser to:

`http://www.analog.com/techsupt/documents/bsdl`

Refer to the IEEE 1149.1 JTAG specification for detailed information on the JTAG interface. The many sections of this chapter assume a working knowledge of the JTAG specification.

Instruction Register

The Instruction register allows an instruction to be shifted into the processor. This instruction selects the test to be performed and/or the test data register to be accessed. The Instruction register is 5 bits long with no parity bit. A value of `10000` binary is loaded (LSB nearest `TDO`) into the instruction register whenever the TAP reset state is entered.

Table 13-2 on page 13-4 lists the binary code for each instruction. Bit 0 is nearest `TDO` and bit 4 is nearest `TDI`. No data registers are placed into test modes by any of the public instructions. The instructions affect the ADSP-2191 DSP as defined in the 1149.1 specification. The optional instructions `RUNBIST` and `USERCODE` are not supported by the ADSP-2191.

Table 13-2. JTAG Instruction Register Codes

Code	Register	Instruction	Type
00000	BOUNDARY	EXTEST[1]	Public
00001	IDCODE	IDCODE1	Public
00010	BOUNDARY	SAMPLE/PRELOAD1	Public
11111	BYPASS	BYPASS1	Public
01110	BYPASS	CLAMP	Public
01101	BOUNDARY	Reserved (HIGHZ)	Public

1 Fixed IR value, can not be moved.

The entry under "Register" is the serial scan path, enabled by the instruction. No special values need be written into any register prior to selection of any instruction. The ADSP-2191 DSP does not support self-test functions.

The data registers are selected via the instruction register. Once a particular data register's value is written into the Instruction Register, and the TAP state is changed to `SHIFT-DR`, the particular data going into or out of the processor is dependent on the definition of the Data Register selected. See the IEEE 1149.1 specification for more details.

When registers are scanned out of the device, the `MSB` is the first bit to be out of the processor.

Bypass Register

The 1-bit Bypass register is fully defined in the 1149.1 specification.

Boundary Register

The Boundary register is used by multiple JTAG instructions. All four of the JTAG instructions that use the Boundary register are required by the 1149.1 specification.

IDCODE Register

The device identification register for the ADSP-2191 DSP is the 32-bit `IDCODE` register. This register includes three fields: the ADI identification code (`0x0E5`), the part identification code (`0x278B`), and the revision number (`0x0`) (Revision number changes with each silicon revision).

References

- IEEE Standard 1149.1-1990. Standard Test Access Port and Boundary-Scan Architecture.

  To order a copy, contact IEEE at 1-800-678-IEEE.

- Maunder, C.M. & R. Tulloss. Test Access Ports and Boundary Scan Architectures.

  IEEE Computer Society Press, 1991.

- Parker, Kenneth. The Boundary Scan Handbook.

  Kluwer Academic Press, 1992.

- Bleeker, Harry, P. van den Eijnden, & F. de Jong. Boundary-Scan Test—A Practical Approach.

  Kluwer Academic Press, 1993.

- Hewlett-Packard Co. HP Boundary-Scan Tutorial and BSDL Reference Guide.

  (HP part# E1017-90001.) 1992.

14 SYSTEM DESIGN

This chapter describes the basic system interface features of the ADSP-2191 family processors. The system interface includes various hardware and software features used to control the DSP processor.

Processor control pins include a $\overline{\text{RESET}}$ signal, clock signals, flag inputs and outputs, and interrupt requests. This chapter describes only the logical relationships of control signals; see the *ADSP-2191 DSP Microcomputer Data Sheet* for actual timing specifications.

This chapter provides the following topics:

Pin Descriptions

This section provides functional descriptions of the ADSP-2191 processor pins. Refer to the data sheet for the ADSP-2191 for more information, including pin numbers for the 144-lead LQFP and the 144-lead mini-BGA packages.

ADSP-2191 pin definitions are listed in Table 14-1 on page 14-3. All of the ADSP-2191 pins are asynchronous.

Unused inputs should be tied or pulled to `VDDEXT` or `GND`, except for `ADDR21-0`, `DATA15-0`, `PF7-0`, and inputs that have internal pull-up or pull-down resistors ($\overline{\texttt{TRST}}$, `BMODE0`, `BMODE1`, `OPMODE`, `BYPASS`, `TCK`, `TMS`, `TDI`, and $\overline{\texttt{RESET}}$); these pins can be left floating. These pins have a logic-level hold circuit that prevents them from floating internally.

The following symbols appear in the Type column of Table 14-1 on page 14-3: G = Ground, I = Input, O = Output, P = Power Supply, and T = Three-State.

Table 14-1. Pin Descriptions

Pin	Type	Description
A21–0	O/T	External Port Address Bus
D7–0	I/O/T	External Port Data Bus, least significant 8 bits
D15 /PF15 /SPI1SEL7	I/O/T I/O O	Data 15 (if 16-bit external bus)/Programmable Flag 15 (if 8-bit external bus)/SPI1 Slave Select output 7 (if 8-bit external bus, when SPI1 enabled)
D14 /PF14 /SPI0SEL7	I/O/T I/O O	Data 14 (if 16-bit external bus)/Programmable Flag 14 (if 8-bit external bus)/SPI0 Slave Select output 7 (if 8-bit external bus, when SPI0 enabled)
D13 /PF13 /SPI1SEL6	I/O/T I/O O	Data 13 (if 16-bit external bus)/Programmable Flag 13 (if 8-bit external bus)/SPI1 Slave Select output 6 (if 8-bit external bus, when SPI1 enabled)
D12 /PF12 /SPI0SEL6	I/O/T I/O O	Data 12 (if 16-bit external bus)/Programmable Flag 12 (if 8-bit external bus)/SPI0 Slave Select output 6 (if 8-bit external bus, when SPI0 enabled)
D11 /PF11 /SPI1SEL5	I/O/T I/O O	Data 11 (if 16-bit external bus)/Programmable Flag 11 (if 8-bit external bus)/SPI1 Slave Select output 5 (if 8-bit external bus, when SPI1 enabled)

Table 14-1. Pin Descriptions (Cont'd)

Pin	Type	Description
D10 /PF10 /SPI0SEL5	I/O/T I/O O	Data 10 (if 16-bit external bus)/Programmable Flag 10 (if 8-bit external bus)/SPI0 Slave Select output 5 (if 8-bit external bus, when SPI0 enabled)
D9 /PF9 /SPI1SEL4	I/O/T I/O O	Data 9 (if 16-bit external bus)/Programmable Flag 9 (if 8-bit external bus)/SPI1 Slave Select output 4 (if 8-bit external bus, when SPI1 enabled)
D8 /PF8 /SPI0SEL4	I/O/T I/O O	Data 8 (if 16-bit external bus)/Programmable Flag 8 (if 8-bit external bus)/SPI0 Slave Select output 4 (if 8-bit external bus, when SPI0 enabled)
PF7 /SPI1SEL3 /DF	I/O O I	Programmable Flag 7/SPI1 Slave Select output 3 (when SPI0 enabled)/Divisor Frequency (divisor select for PLL input during boot)
PF6 /SPI0SEL3 /MSEL6	I/O O I	Programmable Flag 6/SPI0 Slave Select output 3 (when SPI0 enabled)/Multiplier Select 6 (during boot)
PF5 /SPI1SEL2 /MSEL5	I/O O I	Programmable Flag 5/SPI1 Slave Select output 2 (when SPI0 enabled)/Multiplier Select 5 (during boot)
PF4 /SPI0SEL2 /MSEL4	I/O O I	Programmable Flag 4/SPI0 Slave Select output 2 (when SPI0 enabled)/Multiplier Select 4 (during boot)
PF3 /SPI1SEL1 /MSEL3	I/O O I	Programmable Flag 3/SPI1 Slave Select output 1 (when SPI1 enabled)/Multiplier Select 3 (during boot)
PF2 /SPI0SEL1 /MSEL2	I/O O I	Programmable Flag 2/SPI0 Slave Select output 1 (when SPI0 enabled)/Multiplier Select 2 (during boot)
PF1 /SPISS1 /MSEL1	I/O I I	Programmable Flag 1/SPI1 Slave Select input (when SPI1 enabled)/Multiplier Select 1 (during boot)
PF0 /SPISS0 /MSEL0	I/O I I	Programmable Flag 0/SPI0 Slave Select input (when SPI0 enabled)/Multiplier Select 0 (during boot)

Table 14-1. Pin Descriptions (Cont'd)

Pin	Type	Description
$\overline{\text{RD}}$	O/T	External Port Read Strobe
$\overline{\text{WR}}$	O/T	External Port Write Strobe
ACK	I	External Port Access Ready Acknowledge
$\overline{\text{BMS}}$	O/T	External Port Boot Space Select
$\overline{\text{IOMS}}$	O/T	External Port IO Space Select
$\overline{\text{MS3-0}}$	O/T	External Port Memory Space Selects
$\overline{\text{BR}}$	I	External Port Bus Request
$\overline{\text{BG}}$	O	External Port Bus Grant
$\overline{\text{BGH}}$	O	External Port Bus Grant Hang
HAD15–0	I/O/T	Host Port Multiplexed Address and Data Bus
HA16	I	Host Port MSB of Address Bus
HACK_P	I	Host Port ACK Polarity
HRD	I	Host Port Read Strobe
HWR	I	Host Port Write Strobe
HACK	O	Host Port Access Ready Acknowledge
HALE	I	Host Port Address Latch Strobe or Address Cycle Control
$\overline{\text{HCMS}}$	I	Host Port Internal Memory–Internal I/O Memory–Boot Memory Select
$\overline{\text{HCIOMS}}$	I	Host Port Internal I/O Memory Select
CLKIN	I	Clock Input/Oscillator input 0
XTAL	O	Crystal output
BMODE1–0	I	Boot Mode 1–0 (See "Resetting the Processor ("Hard Reset")" on page 14-12 for more information on how the BMODE1-0 pins are used.) The BMODE1 and BMODE0 pins have 85 kΩ internal pull-up resistors.

Table 14-1. Pin Descriptions (Cont'd)

Pin	Type	Description
OPMODE	I	Operating Mode (See "Resetting the Processor ("Hard Reset")" on page 14-12 for more information on how the OPMODE pin is used.) The OPMODE pin has a 85 kΩ internal pull-up resistor.
CLKOUT	O	Clock Output
BYPASS	I	Phase-Lock-Loop (PLL) Bypass mode. The BYPASS pin has a 85 kΩ internal pull-up resistor.
RCLK1–0	I/O/T	SPORT1–0 Receive Clock
RCLK2/SCK1	I/O/T	SPORT2 Receive Clock/SPI1 Serial Clock
RFS1–0	I/O/T	SPORT1–0 Receive Frame Sync
RFS2/MOSI1	I/O/T	SPORT2 Receive Frame Sync/SPI1 Master-Output, Slave-Input data
TCLK1–0	I/O/T	SPORT1–0 Transmit Clock
TCLK2/SCK0	I/O/T	SPORT2 Transmit Clock/SPI0 Serial Clock
TFS1–0	I/O/T	SPORT1–0 Transmit Frame Sync
TFS2/MOSI0	I/O/T	SPORT2 Transmit Frame Sync/SPI0 Master-Output, Slave-Input data
DR1–0	I/T	SPORT1–0 Serial Data Receive
DR2/MISO1	I/O/T	SPORT2 Serial Data Receive/SPI1 Master-Input, Slave-Output data
DT1–0	O/T	SPORT1–0 Serial Data Transmit
DT2/MISO0	I/O/T	SPORT2 Serial Data Transmit/SPI0 Master-Input, Slave-Output data
TMR2–0	I/O/T	Timer output or capture
RXD	I	UART Serial Receive Data
TXD	O	UART Serial Transmit Data

Table 14-1. Pin Descriptions (Cont'd)

Pin	Type	Description
$\overline{\text{RESET}}$	I	Processor Reset. Resets the ADSP-2191 to a known state and begins execution at the program memory location specified by the hardware reset vector address. The $\overline{\text{RESET}}$ input must be asserted (low) at power-up. The $\overline{\text{RESET}}$ pin has a 85 kΩ internal pull-up resistor.
TCK	I	Test Clock (JTAG). Provides a clock for JTAG boundary scan. The TCK pin has a 85 kΩ internal pull-up resistor.
TMS	I	Test Mode Select (JTAG). Used to control the test state machine. The TMS pin has a 85 kΩ internal pull-up resistor.
TDI	I	Test Data Input (JTAG). Provides serial data for the boundary scan logic. The TDI pin has a 85 kΩ internal pull-up resistor.
TDO	O	Test Data Output (JTAG). Serial scan output of the boundary scan path.
$\overline{\text{TRST}}$	I	Test Reset (JTAG). Resets the test state machine. $\overline{\text{TRST}}$ must be asserted (pulsed low) after power-up or held low for proper operation of the ADSP-2191. The $\overline{\text{TRST}}$ pin has a 65 kΩ internal pull-down resistor.
$\overline{\text{EMU}}$	O	Emulation Status (JTAG). Must be connected to the ADSP-2191 emulator target board connector only.
VDDINT	P	Core Power Supply. Nominally 2.5 V dc and supplies the DSP's core processor. (four pins).
VDDEXT	P	I/O Power Supply; Nominally 3.3 V dc. (nine pins).
GND	G	Power Supply Return. (twelve pins).
NC		Do Not Connect. Reserved pins that must be left open and unconnected.

Recommendations for Unused Pins

The following is a list of recommendations for unused pins.

- If the `CLKOUT` pin is not used, turn it off, by clearing bit 6 (`CKOUTEN`) of the PLL Control (`PLLCTL`) register.
- If the interrupt/programmable flag pins are not used, configure them as inputs at reset and function as interrupts and input flag pins, pull the pins to an inactive state, based on the `POLARITY` setting of the flag pin.
- If a flag pin is not used, configure it as an output. If for some reason, it cannot be configured as an output, configure it as an input. Use a 100 kΩ pull-up resistor to `VDD` (or, if this is not possible, use a 100 kΩ pull-down resistor to `GND`).
- If a SPORT is not used completely and if the SPORT pins do not have a second functionality, disable the SPORT and let the pins float.
- If the receiver on a SPORT is the only part being used, use resistors on the other pins. However, if the other pins are outputs, let them float.

Pin States at Reset

The following table shows the state of each pin during and after reset. See "Pin Descriptions" on page 14-2 for a description of each of these pins.

The following symbols appear in the Type column of Table 14-2 on page 14-9: G = Ground, I = Input, O = Output, P = Power Supply, and T = Three-State.

Table 14-2. Pin States at Reset

Pin	Type	State at Reset
A21–0	O/T	High Impedance
D7–0	I/O/T	High Impedance
D15 /PF15 /SPI1SEL7	I/O/T I/O O	High Impedance
D14 /PF14 /SPI0SEL7	I/O/T I/O O	High Impedance
D13 /PF12 /SPI1SEL6	I/O/T I/O O	High Impedance
D12 /PF12 /SPI0SEL6	I/O/T I/O O	High Impedance
D11 /PF11 /SPI1SEL5	I/O/T I/O O	High Impedance
D10 /PF10 /SPI0SEL5	I/O/T I/O O	High Impedance
D9 /PF9 /SPI1SEL4	I/O/T I/O O	High Impedance
D8 /PF8 /SPI0SEL4	I/O/T I/O O	High Impedance
PF7 /SPI1SEL3 /DF	I/O O I	Input
PF6 /SPI0SEL3 /MSEL6	I/O O I	Input

Table 14-2. Pin States at Reset (Cont'd)

Pin	Type	State at Reset
PF5 /SPI1SEL2 /MSEL5	I/O O I	Input
PF4 /SPI0SEL2 /MSEL4	I/O O I	Input
PF3 /SPI1SEL1 /MSEL3	I/O O I	Input
PF2 /SPI0SEL1 /MSEL2	I/O O I	Input
PF1 /SPISS1 /MSEL1	I/O I I	Input
PF0 /SPISS0 /MSEL0	I/O I I	Input
$\overline{\text{RD}}$	O/T	Driven High[1]
$\overline{\text{WR}}$	O/T	Driven High[1]
ACK	I	Input
$\overline{\text{BMS}}$	O/T	Driven High[1]
$\overline{\text{IOMS}}$	O/T	Driven High[1]
$\overline{\text{MS3-0}}$	O/T	Driven High[1]
$\overline{\text{BR}}$	I	Input[1]
$\overline{\text{BG}}$	O	Driven High[1]; Responds to $\overline{\text{BR}}$ during reset
$\overline{\text{BGH}}$	O	Driven High[1]
HAD15–0	I/O/T	Three-stated

Table 14-2. Pin States at Reset (Cont'd)

Pin	Type	State at Reset
HA16	I	Three-stated
HACK_P	I	Input[2]
$\overline{\text{HRD}}$	I	Input[2]
$\overline{\text{HWR}}$	I	Input[2]
HACK	O	Driven
HALE	I	Input[2]
$\overline{\text{HCMS}}$	I	Input[2]
$\overline{\text{HCIOMS}}$	I	Input[2]
CLKIN	I	Input
BMODE1–0	I	Input
OPMODE	I	Input
BYPASS	I	Input
RCLK1–0	I/O/T	Three-stated[3]
RCLK2/SCK1	I/O/T	Three-stated[3]
RFS1–0	I/O/T	Three-stated[3]
RFS2/MOSI1	I/O/T	Three-stated[3]
TCLK1–0	I/O/T	Three-stated[3]
TCLK2/SCK0	I/O/T	Three-stated[3]
TFS1–0	I/O/T	Three-stated[3]
TFS2/MOSI0	I/O/T	Three-stated[3]
DR1–0	I/T	Three-stated[3]
DR2/MISO1	I/O/T	Three-stated[3]

Table 14-2. Pin States at Reset (Cont’d)

Pin	Type	State at Reset
DT1–0	O/T	Three-stated[3]
DT2/MISO0	I/O/T	Three-stated[3]
TMR2–0	I/O/T	Three-stated[3]
RXD	I	Three-stated[3]
TXD	O	Not three-stated[3]
$\overline{\texttt{RESET}}$	I	Input[2]
TCK	I	Input[4]
TMS	I	Input[4]
TDI	I	Input[4]
TDO	O	Three-stated[4]
$\overline{\texttt{TRST}}$	I	Input[4] (pulled low by resistor)
$\overline{\texttt{EMU}}$	O	Three-stated[4] ((open drain with internal pullup)
NC		

1 Three-stated when the DSP is bus slave (held in $\overline{\texttt{BR}}$).
2 Bus Master independent.
3 SPI, SPORT, UART.
4 JTAG.

Resetting the Processor (“Hard Reset”)

The $\overline{\texttt{RESET}}$ signal halts execution and causes a hardware reset of the processor; the program control jumps to address `0xFF0000` and begins execution of the boot ROM code at that location. (If configured in “no boot” mode, the DSP begins execution from PM `0x10000`.)

The ADSP-2191 can be booted via the EPROM, UART, SPI, or Host port. The DSP looks at the values of three pins (`BMODE0`, `BMODE1`, and `OPMODE`) to determine the boot mode, as shown in the following table.

Table 14-3. Hard Reset Boot Mode Pins

OPMODE	BMODE1	BMODE0	Result
0	0	0	No boot; run from external 16-bit memory starting at address 0x10000
0	0	1	Boot from EPROM
0	1	0	Boot from Host
0	1	1	Reserved
1	0	0	No boot; run from external 8-bit memory, bypass ROM, starting at address 0x10000
1	0	1	Boot from UART
1	1	0	Boot from SPI, up to 4K bits
1	1	1	Boot from SPI, >4K bits up to 512K bits

After the DSP has determined the boot mode, it loads the headers and data blocks. For some booting modes, the boot process uses DMA. For more information about DMA, see "I/O Processor" on page 6-1.

The $\overline{\texttt{RESET}}$ signal must be asserted (held low) when the processor is powered up to assure proper initialization.

The internal clock on the ADSP-2191 requires approximately 500 clock cycles to stabilize. To maximize the speed of recovery from reset, `CLKIN` should run during the reset.

The power-up sequence is defined as the total time required for the crystal oscillator circuit to stabilize after a valid `VDD` is applied to the processor and for the internal PLL to lock onto the specific crystal frequency. A minimum of 500 CLKIN cycles ensures that the PLL has locked, but it does not include the crystal oscillator start-up time. During the power-up sequence the $\overline{\text{RESET}}$ signal should be held low.

If a clock has not been supplied during $\overline{\text{RESET}}$, the processor does not know it has been reset and the registers won't be initialized to the proper values.

At powerup, if $\overline{\text{RESET}}$ is held low (asserted) without any input clock signal, the states of the internal transistors are unknown and uncontrolled. This condition could lead to processor damage.

"ADSP-219x DSP Core Registers" on page A-1 and "ADSP-2191 DSP I/O Registers" on page B-1 contain tables showing the $\overline{\text{RESET}}$ states of various registers, including the processors' on-chip memory-mapped status/control registers. The values of any registers not listed are undefined at reset. The contents of on-chip memory are unchanged after $\overline{\text{RESET}}$, except as shown in the tables for the I/O memory-mapped control/status registers. The CLKOUT signal continues to be generated by the processor during $\overline{\text{RESET}}$, except when disabled.

In clock multiplier mode (not bypass mode), the MSELx pins, which define the clock multiplier ratios, are sampled during reset. For information on managing these pins, see "Managing DSP Clocks" on page 14-29.

The contents of the computation unit (ALU, MAC, shifter) and data address generator (DAG1, DAG2) registers are undefined following $\overline{\text{RESET}}$. When $\overline{\text{RESET}}$ is released, the processor's booting operation takes place, depending on the states of the processor's BMODEx and OPMODE pins. (Program booting is described in "Boot Mode DMA Transfers" on page 6-41.)

When the power supply and clock remain valid, the content of the on-chip memory is not changed by a software reset.

Resetting the Processor ("Soft Reset")

A software reset is generated by writing ones to the software reset (SWR) bits in the Software Reset (SWRST) register. A software reset affects only the state of the core and the peripherals (as defined by the peripheral registers documented in"ADSP-2191 DSP I/O Registers" on page B-1). During a soft reset, the DSP does not sample the boot mode pins, rather it gets its boot information from the Next System Configuration (NXTSCR) register.

In no-boot mode, the (RMODE) bit of the Next System Configuration Register has been set to 0, following a soft reset, program flow jumps to address 0xFF0000 and begins executing the boot ROM code at that location to reboot the DSP. A software reset can also be used to reset the boot mode without doing an actual reboot. If bit 4 of the Next System Configuration Register has been set to 1, following a soft reset, program flow jumps to address 0x000000 and completes reset without rebooting the DSP.

The ADSP-2191 can be booted via the EPROM, UART, SPI, or Host port. The DSP uses three bits of the System Configuration (SYSCR) register (loaded from NXTSCR on soft reset) to determine the boot mode, as shown in Figure B-2 on page B-21. (Note that these three bits correspond to the BMODE0, BMODE1, and OPMODE pins used to determine the boot mode for a hard reset, as described in "Resetting the Processor ("Hard Reset")" on page 14-12.)

"ADSP-219x DSP Core Registers" on page A-1 and "ADSP-2191 DSP I/O Registers" on page B-1 contain tables showing the state of the processor registers after a software reset that includes a DSP reboot. The values of any registers not listed are unchanged by a reboot.

The MSELx pins are *not* sampled during a software reset.

Because the ADSP-2191's shadow write FIFO automatically pushes the write to internal memory as soon as the write does not compete with a read, this FIFO's operation is completely transparent to programs, except in software reset/restart situations. To ensure correct operation after a software reset, software must perform two "dummy" writes to memory before writing the software reset bit. For more information, see "Shadow Write FIFO" on page 5-17.

Booting the Processor ("Boot Loading")

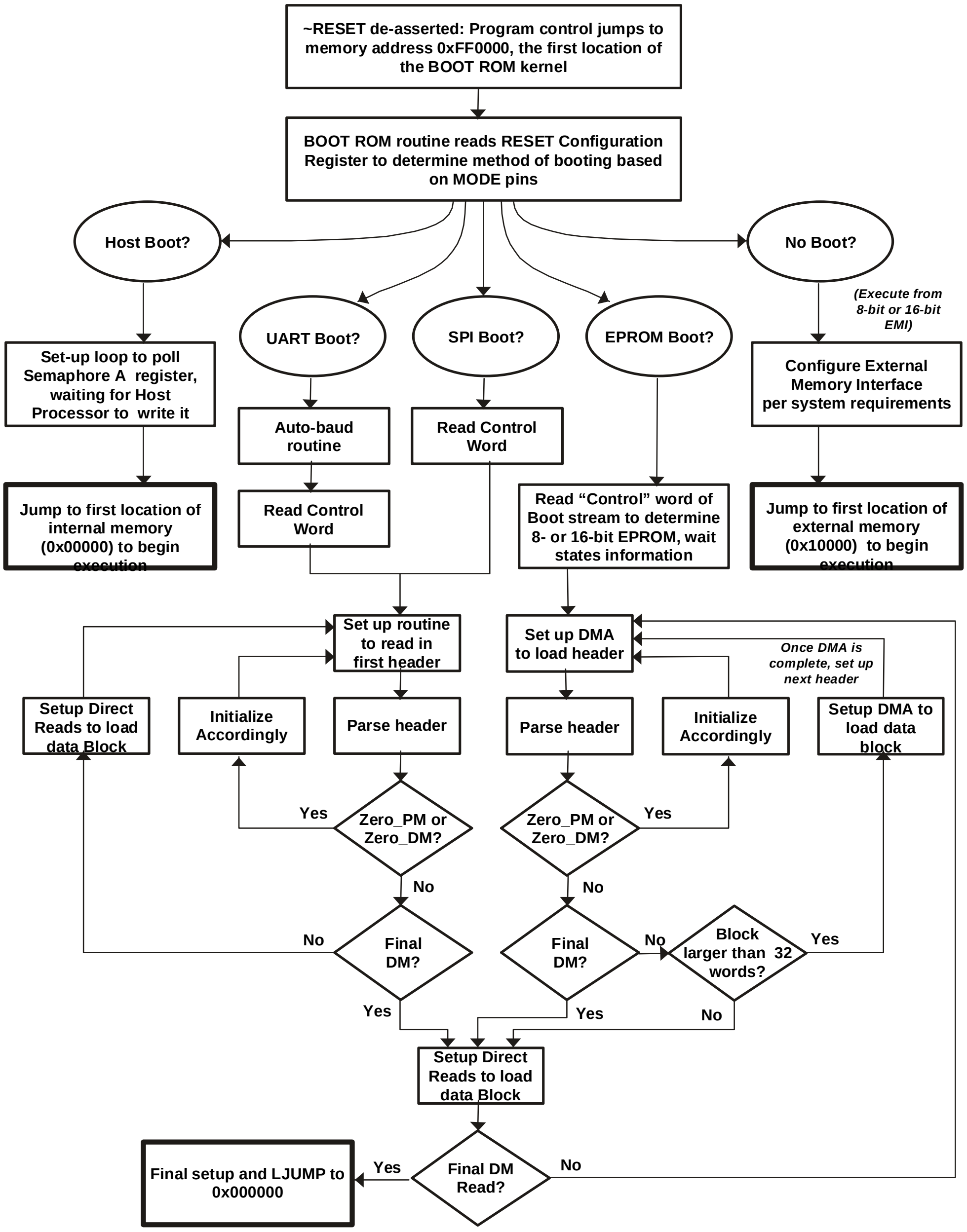
The ADSP-2191 has a booting scheme that is different from previous Analog Devices fixed-point DSP's, such as the ADSP-218x. When the ADSP-218x comes out of reset, it is configured to automatically boot in a "Loader Kernel" using DMA. This Loader Kernel then loads in corresponding "page loaders". On the ADSP-2191, the boot Kernel is located on-chip and stored in a 24-bit wide, 1K ROM. The starting address of this boot ROM begins at 0xFF0000 (i.e., the first location of page 255). For more information on the ADSP-2191 memory map, see Figure 5-4 on page 5-10.

Whether the DSP is reset with hardware ($\overline{\text{RESET}}$ pin) or software (SWR bits in the SWRST register), the boot kernel executes its process. For more information on reset, see "Resetting the Processor ("Hard Reset")" on page 14-12 and "Resetting the Processor ("Soft Reset")" on page 14-15.

Boot Modes

Figure 14-1 on page 14-17 shows the program flow for the boot kernel. Following a reset, the first operation performed by the boot Kernel is to read the System Configuration (SYSCR) register and determine the DSP boot mode.

In the event that the DSP is configured to boot from a peripheral, the first operation performed by the boot Kernel is to read in the first word of the bootstream. This control word contains information on the rest of the boot. The DSP performs this transfer using the modes that the DSP is


~RESET de-asserted: Program control jumps to memory address 0xFF0000, the first location of the BOOT ROM kernel
BOOT ROM routine reads RESET Configuration Register to determine method of booting based on MODE pins
Host Boot?
UART Boot?
SPI Boot?
EPROM Boot?
No Boot?
(Execute from 8-bit or 16-bit EMI)
Set-up loop to poll Semaphore A register, waiting for Host Processor to write it
Jump to first location of internal memory (0x00000) to begin execution
Auto-baud routine
Read Control Word
Read Control Word
Read “Control” word of Boot stream to determine 8- or 16-bit EPROM, wait states information
Configure External Memory Interface per system requirements
Jump to first location of external memory (0x10000) to begin execution
Set up routine to read in first header
Set up DMA to load header
Once DMA is complete, set up next header
Setup Direct Reads to load data Block
Initialize Accordingly
Parse header
Parse header
Initialize Accordingly
Setup DMA to load data block
Yes
Zero_PM or Zero_DM?
Zero_PM or Zero_DM?
Yes
No
No
No
Final DM?
Final DM?
No
Block larger than 32 words?
Yes
Yes
Yes
No
Setup Direct Reads to load data Block
Final setup and LJUMP to 0x000000
Yes
Final DM Read?
No

configure with at reset (e.g., 8-bit external to 16-bit internal packing mode in case of the external memory interface, with maximum wait states and base clock divisor).

If it is determined that the DSP is not going to boot in a program, but instead *run* a program from 8-bit or 16-bit external memory, the boot ROM routine sets up the External Memory Interface and the External Access Bridge register for the desired packing mode (8-bit external to 24-bit internal or 16-bit-external-to-24-bit internal). Then, execution jumps to the first location of external memory (0x10000), where the user program is executed.

SPI Port and UART Port Booting

If booting via a peripheral such as the SPI or UART, the Loader Kernel will set up the corresponding peripheral as follows:

- SPI: SPI0 is used, set up as master, and is set to receive 8-bit words received MS-Bit first, SCLK = peripheral clock/60, with an active-low serial clock to be compatible with commonly available serial EEPROMS.
- UART: In the case of UART boot, the Loader Kernel begins by first running an Auto-baud routine using a timer to determine the baud-rate of the external UART device. Once the baud-rate has been determined, the Loader Kernel will proceed with the rest of boot.

If booting via the SPI or UART, the corresponding DMA engines associated with the peripheral are not used at all, but rather all the data is read in through core reads a byte at a time and packed internally by the Boot Kernel.

The external device transmits the word “0xaa” to allow the timer to capture the pulse width of the device. Once the baud rate has been determined, the DSP UART will transmit the words 0x4F and 0x4B corresponding to “OK” in ASCII. The external device can now begin transmitting the boot file.

Host Port Booting

If booting via a Host processor, the Loader Kernel will relocate the Interrupt vector location to page 0 of memory. It will then sit in a loop polling the Semaphore A register `(IO:0x1CFC)`, waiting for a Host Processor to write to it. The Host processor has the responsibility of loading the code and data into the DSP.

The ADSP-2191 can be booted from either an 8-bit, 16-bit, or 32-bit Host processor. In the case of booting from a 32-bit Host, the Host must send data on the 16 least significant data lines (right-justified).

The Host boot is configured to always use little-endian format, as this is the default that the Host port comes up in.

Example: If the representation of decimal number 1025 is 00000100 00000001, the Table 14-4 on page 14-19 describes big and little endian representation of the number.

Table 14-4. Big and Little Endian Comparison

Address	Big-Endian Representation of 1025	Little-Endian Representation of 1025
00	-	00000001
01	-	00000100
02	00000100	-
03	00000001	-

After the Host processor has finished loading the ADSP-2191, it indicates this by writing a "1" to the Semaphore A Register (IO:0x1CFC). The Boot Kernel will then exit the polling loop and transfers program control to the first location of page 0.

External Memory Interface Booting

If booting via the EMI, the Loader Kernel sets up the corresponding system and control registers and the wait state control register accordingly. The Kernel will then set up a DMA transfer block to read in the first header of the bootstream via DMA.

For BMS to MSx boot sequences, the EMI bus widths must match. The DSP cannot boot 8-bit MSx space from a 16-bit BMS. Also, the DSP cannot boot 16-bit MSx space from an 8-bit BMS.

After a header is read in, the Loader Kernel will parse the header and set up another DMA transfer block to load in the actual data following this header. While this DMA is in progress, the Boot Kernel will poll the DMA ownership bit to determine whether the DMA has completed or not.

To optimize booting speed, due to the overhead of setting up and kicking off DMA sequence, if the size of a data block following the header is less than 32 words, that block is read/initialized using core-driven direct reads as opposed to using DMA.

Once a data block has been read/initialized, the next header is read in, and the process is repeated. This process repeats for all the blocks that need to be transferred.

The last block to be read/initialized will be the "final DM" block. This final block will not be loaded with DMA (even if it is larger than 32 words), but will rather be direct core accesses. The purpose of the final block is to clean up the "scratch area" used by the boot loader for storing temporary DMA control blocks and variables. When it has completed loading in the last piece of data, the interrupt service routine performs some housecleaning and transfers program control to the first location of page 0.

According to the DSP's memory map (Figure 5-4 on page 5-10), Boot Memory Space starts at address 0x10000, but this is the *runtime* map. During EPROM booting, the DSP starts Boot Memory Space at address 0x00000. When designing a system to boot from an EPROM, place the start address of the EPROM at address 0x00000 in Boot Memory Space.

Bootstream Format

The bootstream is comprised of a series of "headers" consisting of 4 words, followed by optional data blocks for non-zero data. Each header contains information on the type of data that immediately follows, the starting address and the word count. In case of booting via the SPI or UART, after a header is read in (the Loader Kernel will use interrupts and a simple-counter based loop to determine the number of words to read in) the Loader Kernel parses the header and sets up another counter-based loop to load in the actual data following this header. These transfers are interrupt-driven.

The first word in the boot-stream is a Control word that applies to all booting formats, with the exception of Host boot and No-Boot. Individual bits within this word are set or cleared based on the method of booting and specific command line options specified by the user and loader utility. This is a 16-bit field that contains among other things, information on the number of waitstates and the Width External port or serial EEPROM (8-bit or 16-bit). The control word appears in Figure 14-2 on page 14-22 and Figure 14-3 on page 14-22.

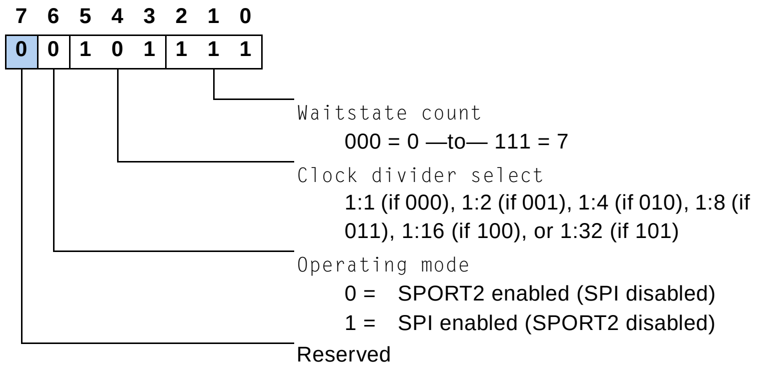

7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 1
Waitstate count
000 = 0 —to— 111 = 7
Clock divider select
1:1 (if 000), 1:2 (if 001), 1:4 (if 010), 1:8 (if 011), 1:16 (if 100), or 1:32 (if 101)
Operating mode
0 = SPORT2 enabled (SPI disabled)
1 = SPI enabled (SPORT2 disabled)
Reserved

Figure 14-2. First Byte of Boot Control Word

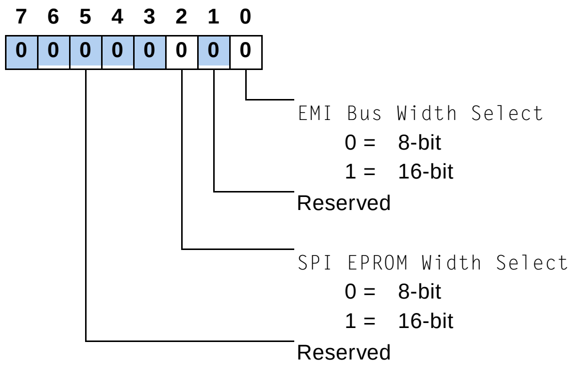


Figure 14-3. Second Byte of Boot Control Word

Following the control word is the regular bootstream, that is, a series of "headers" and data payloads or "blocks", with each header optionally followed by a corresponding block of data. An example bootstream appears in Table 14-5 on page 14-23.

Each header will consist of four 16-bit words: Flag, 24-bit starting address (uses two 16-bit words), and 16-bit word count.

Table 14-5. Sample Bootstream

Word Type	Description
Control Word	16-bit field (Wait State Information, EPROM/SPI Width)
Flag	16-bit field (PM/DM/Final PM/Final DM)
24-bit Starting Address	32-bit field (24-bit padded to yield 32-bits)
16-bit Word Count	16-bit field
Data Word	16-bit field if 16-bit data 32-bit field if 24-bit EMI data 24-bit field if 24-bit SPI/UART data
Data Word	(see above)
:	
:	
Flag	(see above)
24-bit Starting Address	(see above)
16-bit Word Count	(see above)
Data Word	(see above)
Data Word	(see above)
:	

The first word of a header is a 16-bit field consisting of a flag that indicates whether the block of data to follow is either a 24-bit or 16-bit payload or zero-initialized data. The flag also uniquely identifies the last block that needs to be transferred. Table 14-6 on page 14-24 lists the Flags with associated function. While data blocks always have to follow a header, data blocks do not follow headers indicate regions of memory that are to be “zero-filled”.

Table 14-6. Bootstream Flags

Flag Values	Payload Type
0x00	24-bit data/PM
0x01	16-bit data/DM
0x02	Final PM
0x03	Final DM
0x04	zero-init PM
0x05	zero-init DM
0x06	zero-init Final PM
0x07	zero-init Final DM
0x08 through 0xFF	Reserved

The second word of a header (16-bit field) contains the lower 16 bits of the 24-bit start address to begin loading the data (destination). The first octet will be the 8 LSBs, followed by the next most significant bits (8-15), and so on.

The third word (16-bit field) contains the upper-most 8 bits of the 24-bit destination address, padded (suffixed) with a byte of zeros.

The fourth word (16-bit field) contains the word count of the payload. As with the address, the first octet will be the 8 LSBs, the second octet will be the 8 MSBs.

These four words constitute the header. Following the header is the data block. 16-bit data is sent in a 16-bit field while 24-bit data is sent in a 32-bit field.

24-bit data is represented differently in the bootstream from 24-bit addresses. 32-bit data will be transmitted the following way – a byte of zeros, bits 0-7, followed by bits 8-15, and finally bits 16-24. Refer to Figure 5.1(a) for details.

Table 14-7 on page 14-25 and Table 14-8 on page 14-26 show example bootstreams when booting via the EMI, from an 8-bit device and a 16-bit device respectively. Since the DMA engine does not support 8-bit transfers (internal packing has to be one of either 8-16, or 8-24, or 16-16, or 16-24 bits), to load in the 4-word header, the word count needs to be set to 4 in either case.

Table 14-7. 8-bit Device External Memory Interface Bootstream Format in Little-Endian Style

D15 – D8	D7 – D0
Not used	Wait states
Not used	Width
Not used	LSB of Flag
Not used	MSB of Flag
Not used	LSB of Addr
Not used	8-15 of Addr
Not used	MSB of Addr
Not used	00
Not used	LSB of Word count
Not used	MSB of Word count
Not used	LSB of Word
Not used	MSB of Word
:	:
Not used	00
Not used	LSB of Data Word
Not used	8-15 of Data Word
Not used	MSB of Data Word

Table 14-8. 16-bit Device External Memory Interface Bootstream Format in Little-Endian Style

D15 – D8	D7 – D0
00	Wait states
00	Width
MSB of Flag	LSB of Flag
15-8 of Addr	LSB of Addr
00	MSB of Addr
MSB of Word count	LSB of Word count
MSB of Word	LSB of Word
:	:
:	:
MSB of Word	LSB of Word
LSB of Data Word	00
MSB of Data Word	15-8 of Word

Unlike EMI booting, 24-bit data is now represented as three bytes. Table 14-9 on page 14-26 shows the bootstream format when booting via the SPI or UART.

Table 14-9. Bootstream Format for 8-bit SPI Port and UART Port Booting

D15 – D8	D7 – D0
Not used	Wait states
Not used	Width

Table 14-9. Bootstream Format for 8-bit SPI Port and UART Port Booting (Cont'd)

D15 – D8	D7 – D0
Not used	LSB of Flag
Not used	MSB of Flag
Not used	LSB of Addr
Not used	8-15 of Addr
Not used	MSB of Addr
Not used	00
Not used	LSB of Word count
Not used	MSB of Word count
Not used	LSB of Word
Not used	MSB of Word
:	:
Not used	LSB of Data Word
Not used	8-15 of Data Word
Not used	MSB of Data Word

The last block to be read/initialized will be the "final DM" block. This final block is also read in with direct core accesses. Following the final transfer, the interrupt service routine performs some housecleaning and transfers program control to the first location of page 0.

Configuring and Servicing Interrupts

Internal interrupts, including serial port, timer, and DMA interrupts, are discussed in other chapters and appendixes in this manual. For additional information about interrupt masking, set up, and operation, see "Program Sequencer" on page 3-1 and "I/O Processor" on page 6-1.

A variety of interrupts are available on the ADSP-2191. They include core interrupts, user-programmable interrupts, DMA interrupts, and interrupts triggered from a programmable flag (PFx) pin.

User-Mappable Interrupts

The ADSP-2191 includes a set of 12 interrupt lines connecting peripherals to the DSP core. The interrupt lines have fixed priorities, but the peripherals can be remapped to different interrupt lines as needed for a DSP application. The 12 interrupt lines are named DSPIRQ[n] where n has a value from 0 to 11. The peripherals can be mapped to these lines for interrupt priorities from 0 (the highest priority) through 11 (the lowest priority). By default the interrupt lines are mapped to peripherals in numerical order matching the values of n (an interrupt priority of 0 assigned to DSPIRQ[0], an interrupt priority of 1 assigned to DSPIRQ[1], and so on).

The DSP defines four Interrupt Priority (IPRx) registers. Each register defines four of the 12 user-mappable interrupts, as shown in IPRx register diagrams (Figure B-3 on page B-22, Figure B-4 on page B-23, Figure B-5 on page B-23, and Figure B-6 on page B-23). Table 14-10 on page 14-28 shows the relationship of IPRx registers to peripheral interrupts.

Table 14-10. Interrupt Priority (IPRx) Registers and Peripherals

Priority Register	**Priority Fields/Peripherals**
Interrupt Priority Register 0 (IPR0)	Bits 3–0, HOSTIP (Host interrupt priority)
	Bits 7–4, SP0RXIP (SPORT0 RX interrupt priority)
	Bits 11–8, SP0TXIP (SPORT0 TX interrupt priority)
	Bits 15–12, SP1RXIP (SPORT1 RX interrupt priority)

Table 14-10. Interrupt Priority (IPRx) Registers and Peripherals

Priority Register	Priority Fields/Peripherals
Interrupt Priority Register 1 (IPR1)	Bits 3–0, SP1TXIP (SPORT1 TX interrupt priority)
	Bits 7–4, SP2RXIP (SPORT2 RX interrupt priority)
	Bits 11–8, SP2TXIP (SPORT2 TX interrupt priority)
	Bits 15:12, UARRXIP (UART RX interrupt priority)
Interrupt Priority Register 2 (IPR2)	Bits 3–0 , UARTXIP (UART TX interrupt priority)
	Bits 7–4, TIMER0IP (Timer0 interrupt priority)
	Bits 11–8, TIMER1IP (Timer1 interrupt priority)
	Bits 15–12, TIMER2IP (Timer2 interrupt priority)
Interrupt Priority Register 3 (IPR3)	Bits 3–0, FLAGAIP (Flag A interrupt priority)
	Bits 7–4, FLAGBIP (Flag B interrupt priority)
	Bits 11–8, MDMAIP (MemDMA interrupt priority)
	Bits 15–12, Reserved—Must write 0

To change the default interrupt priority for one of the remappable interrupts, set the correct bits of its interrupt priority register to the desired interrupt priority value.

Managing DSP Clocks

Systems can drive the ADSP-2191's clock inputs with a crystal oscillator; sine wave input; or a buffered, shaped clock derived from an external clock oscillator. If the system uses a crystal oscillator, the crystal should be connected across the CLKIN and XTAL pins, with two capacitors connected as shown in Figure 14-4 on page 14-30. Capacitor values are dependent on crystal type and should be specified by the crystal manufacturer. Use a parallel-resonant, fundamental frequency, microprocessor-grade crystal for this configuration.

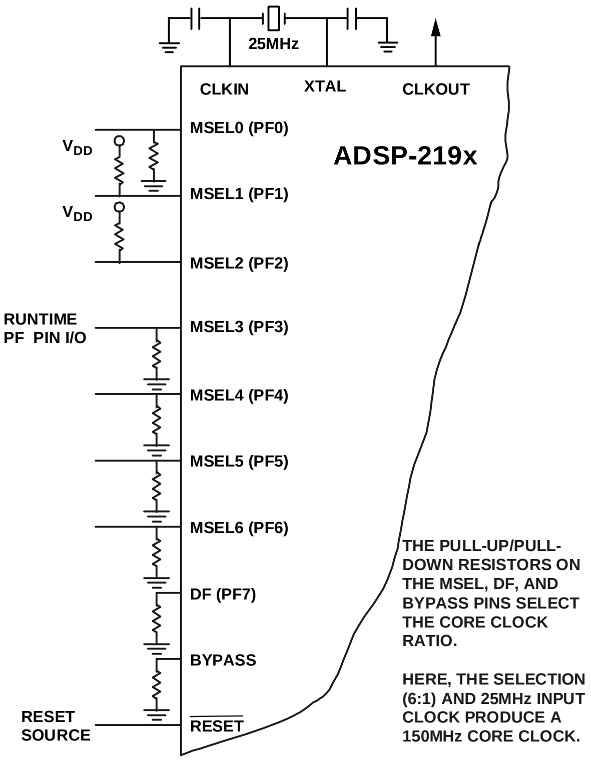

25MHz
CLKIN
XTAL
CLKOUT
ADSP-219x
MSEL0 (PF0)
MSEL1 (PF1)
MSEL2 (PF2)
MSEL3 (PF3)
MSEL4 (PF4)
MSEL5 (PF5)
MSEL6 (PF6)
DF (PF7)
BYPASS
RESET
VDD
VDD
RUNTIME PF PIN I/O
RESET SOURCE
THE PULL-UP/PULL-DOWN RESISTORS ON THE MSEL, DF, AND BYPASS PINS SELECT THE CORE CLOCK RATIO.
HERE, THE SELECTION (6:1) AND 25MHz INPUT CLOCK PRODUCE A 150MHz CORE CLOCK.

Figure 14-4. External Crystal Connections

If a buffered, shaped clock is used, this external clock connects to the DSP's `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. When an external clock is used, the `XTAL` input must be left unconnected.

Unlike previous ADSP-218x DSPs, The ADSP-2191 processor does not support the fixed 2x `CLKIN` clock mode. Instead, the ADSP-2191 includes a clock multiplier which provides more flexibility for multiple clock modes and permits changing the core-to-peripheral clock ratio.

The DSP uses the clock input (CLKIN) to generate on chip clocks. These on chip clocks include the core clock (CCLK), the peripheral clock (HCLK), and external memory interface clocks (EMICLK, one per bank).

Depending on the clock multiplier mode (multiplier mode or bypass mode), the DSP's on chip phase lock loop (PLL) can apply a programmable 1x to 32x multiplication factor to the clock input as the PLL generates the on chip clocks. At reset, the MSEL6-0, BYPASS, and DF pins select the PLL functionality. At runtime, the PLLCTL register permits controlling and changing the clock modes (including the multiplication factor) in software. The following code examples shows how to use the PLLCTL register.

```
/* Example PLL control code */
IOPG = 0x00;             /* Init IOPG to Clk/Sys Control */
AX1 = 0x0352;            /* Stop the PLL - In Bypass */
IO(0x200) = AX1;         /* Write the PLLCTL register */
AX1 = 0x0552;            /* Reprogram to 2x - In Bypass */
IO(0x200) = AX1;         /* Write the PLLCTL register */
AX1 = 0x0550;            /* Start the PLL - In Bypass */
IO(0x200) = AX1;         /* Write the PLLCTL register */
AX1 = 0x0450;            /* Exit Bypass- ~500 clkin cycles */
IO(0x200) = AX1;         /* Write the PLLCTL register */
CNTR = 1100;             /* Wait for a duration of time */
DO wt UNTIL CE;
wt: NOP;
```

The wait loop above is optional. As an alternative, poll the LOCKCNT register. The DSP continues running in bypass mode until the lock counter expires. Then, core and peripherals are connected back to the PLL automatically.

To support input clocks greater than 100 MHz, the PLL uses an additional input: the divide frequency (DF) pin. Setting the DF pin divides CLKIN/2 prior to the PLL; therefore, if the input clock is greater than 100 MHz, DF must be high. If the input clock is less than 100 MHz, DF must be low.

The combination of pull-up and pull-down resistors in Figure 14-4 on page 14-30 put the DSP in clock multiplier mode (BYPASS=0), select an input clock <100 MHz (DF=0), and select a 6x clock multiplier (MSELx=core clock ratio of 6:1), producing a 150 MHz core clock from the 25 MHz input.

All on-chip peripherals for the ADSP-2191 operate at the rate set by the peripheral clock. The peripheral clock is either equal to the core clock rate or one-half the DSP core clock rate. This selection is controlled by the IOSEL bit in the PLLCTL register. The maximum core clock is 160 MHz, and the maximum peripheral clock is 80 MHz—the combination of the input clock and core/peripheral clock ratios may not exceed these limits.

ⓘ The clock on the ADSP-2191 requires approximately 500 clock cycles to stabilize. To maximize the speed of recovery from reset, CLKIN must run during the $\overline{\text{RESET}}$.

Using the PLL Control (PLLCTL) Register

The PLL Control (PLLCTL) register selects the DSP's core clock (CCLK) frequency and select powerdown modes. The PLL multiplies the clock frequency of the input clock with a programmable ratio.

At reset, the PLL starts in bypass mode, running CCLK directly from CLKIN. The reset must be active for sufficient time to allow full initialization of the synchronizer chain. For timing information, see the *ADSP-2191 DSP Microcomputer Data Sheet*. After reset, software can switch to a clock multiplier mode as discussed in "Using Clock Modes" on page 14-37.

The bits in the PLLCTL register are illustrated in Figure B-1 on page B-18. The bits in the PLLCTL register are as follows:

- **Divide Frequency.** Bit 0 (DF) is read/write. A value of 0 for the DF bit disables the input divider; when the DF bit has a value of 1, CLKIN is divided by 2. This bit lets the system configure the PLL to use a high frequency clock input (80–160 MHz; PLL divides input by 2) or low frequency clock input (1–80 MHz; no divide).

- **PLL Off.** Bit 1(PO) is read/write. PO lets software shut off the PLL. A value of 0 for the PO bit turns the PLL on; a value of 1 turns it off.

- **Stop All PLL Output.** Bit 2 (STOPALL) is read/write. A value of 0 for the STOPALL bit enables PLL output; a value of 1 sets the CCLK and HCLK clocks high and disables their output.

- **Stop Core Clock.** Bit 3 (STOPCK) is read/write. A value of 0 for the STOPCK bit enables CCLK output; a value of 1 sets CCLK high and disables its output.

- **Core:Peripheral Clock Ratio.** Bit 4 (IOSEL) is read/write. A value of 0 for the IOSEL bit sets CCLK to the value of HCLK; a value of 1 for the IOSEL bit sets HCLK to CCLK/2.

- **Powerdown.** Bit 5 (PDWN) is read/write. A value of 0 for the PDWN bit means the PLL is running; a value of 1 puts the PLL into low power mode, which shuts off the PLL circuitry and stops the input clock to the PLL.

- **CLKOUT Enable.** Bit 6 (CKOUTEN) is read/write. A value of 0 for the CKOUTEN bit sets CLKOUT to 0; a value of 1 for the CKOUTEN bit sets CLKOUT to the value of HCLK.

- **Divide CLKIN/2 in Bypass Enable.** Bit 7 (DIV2) is read/write. A value of 0 for the DIV2 bit specifies no divide; a value of 1 for the DIV2 bit sets CCLK to CLKIN/2 in bypass mode.

- **Bypass PLL Multiplier.** Bit 8 (BYPASS) is read/write. A value of 0 for the BYPASS bit puts the PLL into multiplier mode; a value of 1 puts the PLL into bypass mode.
- **Multiplier Select.** Bits 15–9 (MSEL6-0). These bits are latched from the mode pins at hardware reset. These bits select the CLKIN multiplier as shown in Table 14-11 on page 14-34.

There are a number of restrictions on the relationship between the input clock (CLKIN), the core clock (CCLK) and the phase lock loop clock (PLLCK). These restrictions are identified in the notes following Table 14-11 on page 14-34.

Table 14-11. CLKIN Multiplier Values (xCLKIN=CCLK)[1,2,3,4,5,6]

Pin name	MSEL6:DF = 00		MSEL6:DF= 01		MSEL6:DF= 10		MSEL6:DF= 11	
MSEL4:0	**CCLK**	**PLLCK**	**CCLK**	**PLLCK**	**CCLK**	**PLLCK**	**CCLK**	**PLLCK**
0	32x	32x	16x	16x	16x	32x	8x	16x
1	1x	1x	0.5x	0.5x	0.5x	1x	0.25x	0.5x
2	2x	2x	1x	1x	1x	2x	0.5x	1x
3	3x	3x	1.5x	1.5x	1.5x	3x	0.75x	1.5x
4	4x	4x	2x	2x	2x	4x	1x	2x
5	5x	5x	2.5x	2.5x	2.5x	5x	1.25x	2.5x
6	6x	6x	3x	3x	3x	6x	1.5x	3x
7	7x	7x	3.5x	3.5x	3.5x	7x	1.75x	3.5x
8	8x	8x	4x	4x	4x	8x	2x	4x
9	9x	9x	4.5x	4.5x	4.5x	9x	2.25x	4.5x
10	10x	10x	5x	5x	5x	10x	2.5x	5x
11	11x	11x	5.5x	5.5x	5.5x	11x	2.75x	5.5x
12	12x	12x	6x	6x	6x	12x	3x	6x
13	13x	13x	6.5x	6.5x	6.5x	13x	3.25x	6.5x
14	14x	14x	7x	7x	7x	14x	3.5x	7x
15	15x	15x	7.5x	7.5x	7.5x	15x	3.75x	7.5x
16	16x	16x	8x	8x	8x	16x	4x	8x
17	17x	17x	8.5x	8.5x	8.5x	17x	4.25x	8.5x

Table 14-11. CLKIN Multiplier Values (xCLKIN=CCLK)[1,2,3,4,5,6]

Pin name	MSEL6:DF = 00		MSEL6:DF= 01		MSEL6:DF= 10		MSEL6:DF= 11	
MSEL4:0	CCLK	PLLCK	CCLK	PLLCK	CCLK	PLLCK	CCLK	PLLCK
18	18x	18x	9x	9x	9x	18x	4.5x	9x
19	19x	19x	9.5x	9.5x	9.5x	19x	4.75x	9.5x
20	20x	20x	10x	10x	10x	20x	5x	10x
21	21x	21x	10.5x	10.5x	10.5x	21x	5.25x	10.5x
22	22x	22x	11x	11x	11x	22x	5.5x	11x
23	23x	23x	11.5x	11.5x	11.5x	23x	5.75x	11.5x
24	24x	24x	12x	12x	12x	24x	6x	12x
25	25x	25x	12.5x	12.5x	12.5x	25x	6.25x	12.5x
26	26x	26x	13x	13x	13x	26x	6.5x	13x
27	27x	27x	13.5x	13.5x	13.5x	27x	6.75x	13.5x
28	28x	28x	14x	14x	14x	28x	7x	14x
29	29x	29x	14.5x	14.5x	14.5x	29x	7.25x	14.5x
30	30x	30x	15x	15x	15x	30x	7.5x	15x
31	31x	31x	15.5x	15.5x	15.5x	31x	7.75x	15.5x

1 The values in this table apply for MSEL5==0; For MSEL5==1, see note 6.

2 The same DSP core clock frequency (CCLK) can be obtained with different combinations of MSEL[6:0]/DF; One combination may work better in a given application either to run at lower power (DF=1) or to satisfy the PLLCK minimum frequency (10MHz).

3 The PLLCK minimum frequency is 10 MHz, and therefore for any MSEL value, for which the PLLCK frequency is going to be less than 10 MHz, the user needs to select the PLL BYPASS mode. For e.g., if CLKIN = 3.33 MHz and MSEL = 0x01(hex) for a 1x operation, BYPASS mode should be selected. On the other hand if CLKIN = 3.33 MHz, and MSEL = 26(hex) for 3x operation, BYPASS mode is not required as the PLLCK will be 6x (20 MHz).

4 The PLLCK maximum frequency is 400 MHz. If the system uses high values for the clock ratio, then care should be taken that the PLLCK frequency doesn't exceed 400 MHz. For example, if 50x is the PLLCK ratio to achieve a 25x CCLK/CLKIN ratio, then the CLKIN should not exceed 400/50 = 8MHz.

5 Although the PLLCK supports Core clocks (CCLK) up to 200 MHz, the output frequency for a given ADSP-219x part is limited by the core speed.

6 Note that when MSEL[5]==1, DF must also be set to 1 by the user; in this case, output clock frequency ratio on CCLK and PLLCK are the same as with MSEL[5]==0 and DF==0.

Designing for Multiplexed Clock Pins

The ADSP-2191's MSEL6-0 pins and PF6-0 pins are multiplexed. During reset, these pins act as multiplier selects (if in multiplier mode) and act as programmable flags after reset. This multiplexing influences system design as follows:

- For systems selecting bypass mode during reset, MSELx pin states do not need to be managed during reset. The multiplexed nature of these pins does not influence system design for the PFx pins when using bypass mode during reset.
- For systems using multiplier mode during reset and not using PFx pins at runtime, use pullup or pulldown resistors to select the MSELx value. Do not leave these pins unconnected.
- For systems using multiplier mode during reset and using the PFx pins at runtime, use pullup or pulldown resistors to select the MSELx value during reset and ensure that the system permits the MSELx pins to stabilize to a valid multiplier value in compliance with the timing for $\overline{\mathtt{RESET}}$ in the datasheet.

The timing for the MSELx, BYPASS, and DF pins during reset is identical and has the following features (as showing in Figure 14-7 on page 14-25):

- t_{PFD}—Delay from $\overline{\mathtt{RESET}}$ asserted to PFx input or output is terminated. From this point, the MSELx values begin stabilizing to a valid state.
- t_{MSD}—Delay from $\overline{\mathtt{RESET}}$ asserted to MSELx must have valid values. The values can change from this point, but only from one valid value to another.
- t_{MSS}—Setup for MSELx value before $\overline{\mathtt{RESET}}$ deasserted. The value must be held from this point until hold time completes.
- t_{MSH}—Hold for MSELx value after $\overline{\mathtt{RESET}}$ deasserted.

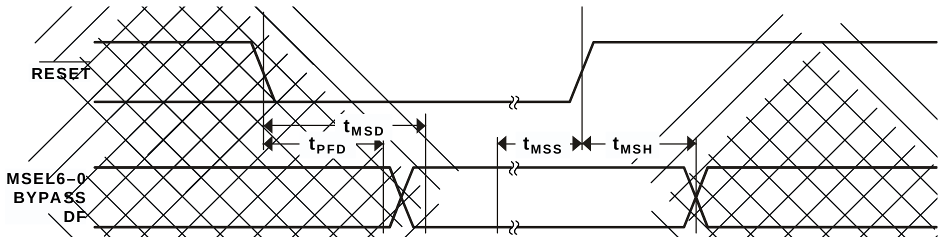


Figure 14-5. MSELx, BYPASS, and DF Timing

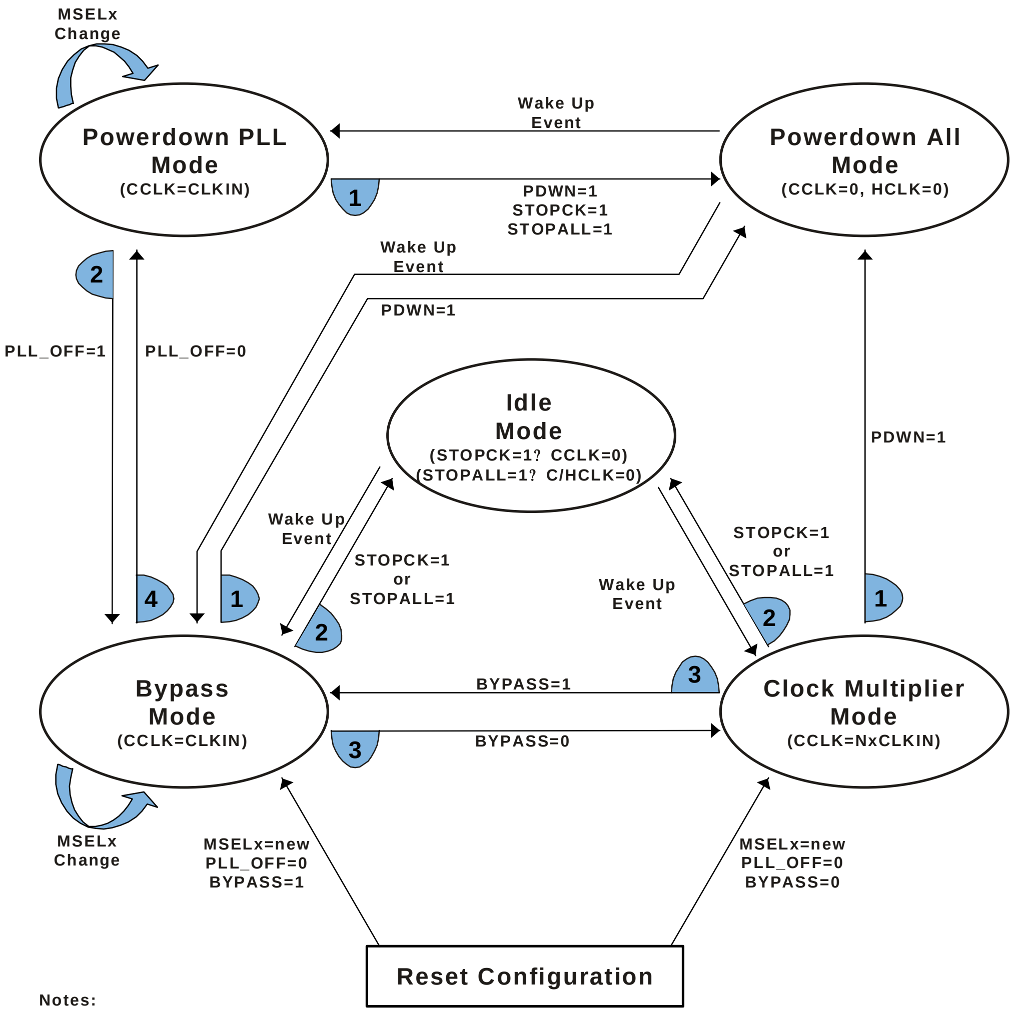
Using Clock Modes

Figure 14-6 on page 14-38 shows a state diagram for the DSP's clock modes. Note the following key points that this diagram illustrates:

- The MSELx pins provide input for the reset configuration (at reset).
- The MSELx bits in the PLLCTL register provide input when the DSP is in bypass or power-down PLL modes (at runtime).
- The PWDN, PLL_OFF, STOPCK, STOPALL, and BYPASS bits in the PLLCTL register control movement between clock modes.
- The PLL determines the mode change from the bits and the bits' priorities.

These modes are provided in the DSP in order to cutoff clock signals to the core and/or to the peripherals. This is an important requirement for the low-cost power sensitive applications.

In power-down all mode, BYPASS and PLL are off, and there are no output clocks. An asynchronous wake-up (FIO_WAKEUP, TMR_WAKE0, TMR_WAKE1 or TMR_WAKE2) is expected to trigger the wake-up sequence.



Notes:

1) A **Wake Up Event** is an interrrupt that prompts the DSP to return from Idle or Powerdown All Mode.

2) The **PLL mode arbitration priority** (number tags on outbound paths from each mode) determine the mode change. If the DSP is in Bypass mode and both the PDWN and PLL_OFF bits where set in the same cycle (for example), the PLL puts the DSP in Powerdown All mode because that mode change has the higher priority (1).

Figure 14-6. Clock PLL Modes Flowchart

In Idle mode the PLL is on, but CCLK is off and/or HCLK, too. An asynchronous wake-up or in the case the HCLK is running, an interrupt (IRQ_INT) can trigger the wake-up sequence.

The PLL can be in five different transition states, as specified by the bits in the PLLCTL register.

- *Clock Multiplier Mode.* BYPASS is off and PLL is on. The output clock is generated by the PLL with the desired frequency ratio. While in clock multiplier mode, the PLLCTL register's STOPCK, STOPALL, BYPASS, PDWN, DIV2, and IOSEL bits can be changed; the other bits of the PLLCTL register cannot be changed in clock multiplier mode. If more than one of these bits is updated, there is a predefined order for the update, as shown in the state diagram. Any change of these bits leads to a state transition.

- *Bypass Mode.* BYPASS and PLL are both on. The PLL is in BYPASS mode, and the input clock is directly used to generate the clocks for the core and the peripherals. In this mode the multiplication ratio can be changed. The lock counter defines when the PLL is locked to the new ratio and can get to clock multiplier mode. While in Bypass mode, the PLLCTL register's STOPCK, STOPALL, PLLOFF, BYPASS, IOSEL, and PDWN bits can be written. The DIV2, MSEL, and DF bits can also be written when BYPASS=1. If more than one of these bits is updated, there is a predefined order for the update, as shown in Figure 14-6 on page 14-38.

- *Idle Mode.* The PLL is on, and the core is in idle mode. The PLLCTL register cannot be written when in Idle mode. An external event or some peripheral activity is expected to generate the wake-up interrupt. There are two configurations for this mode; only CCLK may be turned off, or both CCLK and HCLK may be turned off. The PLLCTL register's BYPASS bit determines the next state of the PLL after wake-up: BYPASS=1 means the next state will be Bypass mode, otherwise the next state will be multiplier mode.

- *Power-down PLL Mode.* BYPASS is on and PLL is off. The DSP is in bypass mode, and the input clock is directly used to generate the clocks for the core and the peripherals. In this mode the multiplication ratio (MSEL) can be changed. The lock counter defines when the PLL is locked to the new ratio and can transition to the bypass mode before switching to multiplier mode. While in powerdown PLL mode, the PLLCTL register's PDWN, PLLOFF, DF, DIV2, and IOSEL bits can be changed; the BYPASS bit cannot be changed. When the STOPCK and STOPALL bits change, the next state will be power-down all mode.
- *Power-down All Mode.* The PLL is off, and the output clocks CCLK and HCLK are off. Because the clocks are off, the PLLCTL register state cannot change in this mode. An asynchronous event is expected to trigger the wake-up sequence for the DSP. The PLLOFF bit of the PLLCTL register determines the next state of the PLL after wake-up: PLLOFF=0 means the next state will be powerdown PLL mode; otherwise the next state will be bypass mode. If the PLL was in multiply mode before it went into power-down all mode, it will wake up in bypass mode but will transition to multiplier mode as soon as the PLL is locked.

Using Programmable Flags

This section includes a detailed example of how to set up the programmable flags; configure the flag interrupt sources, sensitivities, and polarities; configure the interrupt priorities; and mask and enable interrupts. See "Programmable Flags Example" on page 14-53.

The ADSP-2191 DSP has 16 general-purpose flag pins (PF15-0) shown in Table 14-1 on page 14-3. Eight of these flag pins are available when the DSP is using an 8- or 16-bit bus. When the DSP is using an 8-bit bus, eight additional general-purpose flag pins are also available. The lower eight pins (the ones that are always available) can be used in three different ways: as general-purpose flag pins (referred to as PFx or PF7-0), as clock multiplier select pins (MSELx or MSEL6-0), or as select pins for external SPI devices (SPInSELx and SPISSn).

As multiplier select (MSELx) pins, these pins define the clock multiplier ratios. If the BYPASS pin is set (=1), the MSELx pins are bypassed and the clock is passed straight through to the DSP. If the BYPASS pin is cleared (=0), the values of the MSELx pins are used to determine the clock multiplier value. The DF pin (alternately named PF7) controls the input divider. The input divider is disabled when DF is cleared (=0); when DF is set (=1), CLKIN is divided by two before being used.

For information on working with the MSELx/PFx pins, see "Designing for Multiplexed Clock Pins" on page 14-36.

Data being read from a pin configured as an input is synchronized to the processor's clock (HCLK). Pins configured as outputs drive the appropriate output value.

Flag Configuration Registers

The PFx flags on the ADSP-2191 are programmed with a group of flag configuration registers: the Flag Direction (DIR) register, the Flag Control registers (FLAGC and FLAGS), the Flag Interrupt Mask registers (MASKAC, MASKAS, MASKBC, and MASKBS), the Flag Interrupt Polarity (FSPR) register, and the Flag Sensitivity registers (FSSR and FSBER). These registers are described in the following sections.

Several precautions should be observed when programming these flag configuration registers:

- To avoid unwanted interrupts, software should only change a FLAGx[n] bit while its respective interrupt bit, MASKx[n], is masked.
- Five NOPs or instructions must follow an FSPRx[n] bit change, and the respective FLAG[n] bit must be cleared before its interrupt bit is unmasked.
- At reset, all flag configuration registers are initialized to zero; all flag pins are configured as level-sensitive inputs with no inversion, all flag interrupts are masked, and all interrupts are disabled.
- Narrow positive active input [n] pulses are only detectable if FSPRx[n]=0; narrow negative active input [n] pulses are only detectable if FSPRx[n]=1.

For more information about the programmable flag registers, see "ADSP-2191 DSP I/O Registers" on page B-1.

Flag Direction (DIR) Register

The Flag Direction (DIR) register configures a flag pin as an input or output. The DIR register is located at I/O memory page 0x06, I/O address 0x000. (The DIR register is also aliased to I/O memory page 0x06, I/O address 0x001.) Writing a "1" to a bit of the DIR register (at either I/O address) configures the corresponding flag pin as an output; writing a "0" configures the corresponding flag pin as an input. Each bit of the DIR register corresponds with each of the 16 available flag pins of the ADSP-2191.

Flag Control (FLAGC and FLAGS) Registers

The Flag Control registers set or clear a flag pin.

The Flag Clear (FLAGC) register is used to clear the flag pin when it is configured as either an input or an output. FLAGC is located at I/O memory page 0x06, I/O address 0x0002. Writing a “1” to the FLAGC register clears the corresponding flag pin; writing a “0” has no effect on the value of the flag pin. The 16 bits of the FLAGC register correspond to the 16 available flag pins of the ADSP-2191.

The Flag Set (FLAGS) register is used to set the flag pin when it is configured as either an input or an output. Setting a flag pin that is configured as an input allows for software configurable interrupts. FLAGS is located at I/O memory page 0x06, I/O address 0x0003. Writing a “1” to the FLAGS register sets the corresponding flag pin; writing a “0” has no effect on the value of the flag pin. The 16 bits of the FLAGS register correspond to the 16 available flag pins of the ADSP-2191.

The Flag Set (FLAGS) register is used to set the flag pin when it is configured as an output. Software interrupts may be implemented this way. Writes to FLAGS/FLAGC are ignored if the corresponding pin is configured as an input.

When switching from input to output, the current value of the FLAGS/FLAGC registers are applied to the corresponding output pin. While a pin is configured as input, writes to FLAGS/FLAGC are not latched and the content of FLAGS/FLAGC is controlled by the input signal and the polarity register (FSPR). Assuming FSPR cleared, a pin change from input to output initially outputs the same state as the input had before. This behavior guarantees a clean transition in case an external pull-up or pull-down resistor defines the state during start-up.

Flag Interrupt Mask Registers (MASKAC, MASKAS, MASKBC, and MASKBS)

The Flag Interrupt Mask registers enable a flag pin as an interrupt source. The flag pin can be configured as either an input or an output signal. The MASKA and MASKB registers allow for two different programmable flag 0 and 1 interrupt priority levels for all of the flag pins.

The Flag Interrupt MASKA and MASKB Set registers (MASKAS and MASKBS, respectively) are used to “unmask” or enable the servicing of the flag interrupt. The MASKAS register is located at I/O memory page 0x06, I/O address 0x005. The MASKBS register is located at I/O memory page 0x06, I/O address 0x007. Writing a “1” to the MASKAS or MASKBS register unmasks the interrupt capability of the corresponding flag pin; writing a “0” has no effect on the masking of the flag pin. The 16 bits of the MASKAS and MASKBS registers correspond to the 16 available flag pins of the ADSP-2191.

The Flag Interrupt MASKA and MASKB Clear registers (MASKAC and MASKBC, respectively) are used to “mask” or disable the servicing of the flag interrupt. The MASKAC register is located at I/O memory page 0x06, I/O address 0x004. The MASKBC register is located at I/O memory page 0x06, I/O address 0x006. Writing a “1” to the MASKAC or MASKBC register masks the interrupt capability of the corresponding flag pin; writing a “0” has no effect on the masking of the flag pin. The 16 bits of the MASKAC and MASKBC registers correspond to the 16 available flag pins of the ADSP-2191.

Flag Interrupt Polarity (FSPR) Register

The Flag Interrupt Polarity (FSPR) register selects either a high or low polarity of an interrupt signal. The flag polarity applies for input flag pins only (DIR[n]=0).

The Flag Interrupt Polarity (FSPR) register is located at I/O memory page 0x06, I/O address 0x008. (The FSPR register is also aliased to I/O memory page 0x06, I/O address 0x009.) Writing a “0” to a bit of the FSPR register configures the corresponding flag pin as an active high input signal; writing a “1” configures the corresponding flag pin as an active low input signal. The 16 bits of the FSPR register correspond to the 16 available flag pins of the ADSP-2191.

Flag Sensitivity (FSSR) Register and Flag Sensitivity Both Edges (FSBER) Register

The Flag Sensitivity (FSSR) register determines edge- or level-sensitivity when the flag pin is configured as an input (DIR[n]=0). If the flag pin is configured for edge-sensitivity, the FSSR register also specifies the flag pin's sensitivity for rising edge, falling edge, or both edges.

FSSR is located at I/O memory page 0x06, I/O address 0x00A. (The FSSR register is also aliased to I/O memory page 0x06, I/O address 0x00B.) Writing a "0" to a bit of the FSSR register configures the corresponding flag pin as a level sensitive input; writing a "1" configures the corresponding flag pin as an edge sensitive input. The 16 bits of the FSSR register correspond to the 16 available flag pins of the ADSP-2191.

The Flag Sensitivity Both Edges (FSBER) register configures the sensitivity of the flag pin for rising-edge, falling-edge, or both edges sensitivity (depending on the value of the FSPR[n] bit).

FSBER is located at I/O memory page 0x06, I/O address 0x00C. (The FSBER register is also aliased to I/O memory page 0x06, I/O address 0x00D.) Writing a "0" to a bit of the FSBER register configures the corresponding flag pin for rising-edge or falling-edge sensitivity (as determined by the value of the corresponding bit of the FSPR register); writing a "1" configures the corresponding flag pin for both-edges sensitivity. The 16 bits of the FSBER register correspond to the 16 available flag pins of the ADSP-2191.

For more information, see "Programmable Flags Example" on page 14-53.

Power-Down Modes

The ADSP-2191 has four low-power options that significantly reduce the power dissipation. To enter any of these modes, the DSP executes an IDLE instruction. The ADSP-2191 uses configuration of the PDWN, STOPCK, and

STOPALL bits in the PLLCTL register to select between the low-power modes as the DSP executes the IDLE. Depending on the mode, an IDLE shuts off clocks to different parts of the DSP in the different modes. The low power modes are:

- Idle
- Power-down core
- Power-down core/peripherals
- Power-down all

Idle Mode

When the ADSP-2191 is in idle mode, the DSP core stops executing instructions, retains the contents of the instruction pipeline, and waits for an interrupt. The core clock and peripheral clock continue running.

To enter IDLE mode, the DSP can execute the Idle instruction anywhere in code. To exit IDLE mode, the DSP responds to an interrupt and upon RTI, resumes executing the instruction after the IDLE.

Power-Down Core Mode

When the ADSP-2191 is in power-down core mode, the DSP core clock (CCLK) is off, but the PLL is running. The peripheral clock (HCLK) keeps running, letting the peripherals receive data. The peripherals cannot do DMA, because the on-chip memory is controlled by the CCLK. The peripherals can issue an interrupt to exit power-down.

To enter power-down core mode, the DSP executes an `IDLE` instruction after performing the following tasks:

- Check for pending interrupts and I/O service routines
- Clear (= 0) the `PDWN` bit in the `PLLCTL` register
- Clear (= 0) the `STOPALL` bit in the `PLLCTL` register
- Set (= 1) the `STOPCK` bit in the `PLLCTL` register
- The PLL will issue a power-down interrupt
- ADSP-2191 enters power-down upon encountering an `Idle` instruction in the ISR

To exit power-down core mode, the DSP responds to an interrupt and resumes executing instructions with the instruction after the `IDLE`.

Power-Down Core/Peripherals Mode

When the ADSP-2191 is in power-down core/peripherals mode, the DSP core clock and peripheral clock are off, but the DSP keeps the PLL running. The peripheral clock is stopped, so the peripherals cannot receive data.

To enter power-down core/peripherals mode, the DSP executes an `IDLE` instruction after performing the following tasks:

- Check for pending interrupts and I/O service routines
- Clear (= 0) the `PDWN` bit in the `PLLCTL` register

- Set (= 1) the `STOPALL` bit in the `PLLCTL` register
- The PLL will issue a power-down interrupt
- ADSP-2191 enters power-down upon encountering an `Idle` instruction in the ISR

To exit power-down core/peripherals mode, the DSP responds to an interrupt and (after five to six cycles of latency) resumes executing instructions with the instruction after the `IDLE`.

Power-Down All Mode

When the ADSP-2191 is in power-down all mode, the DSP core clock, the peripheral clock, and the PLL are all stopped. The peripheral clock is stopped, so the peripherals cannot receive data.

To enter power-down all mode, the DSP executes an `IDLE` instruction after performing the following tasks:

- Check for pending interrupts and I/O service routines
- Set (= 1) the `PDWN` bit in the `PLLCTL` register
- The PLL will issue a power-down interrupt
- ADSP-2191 enters power-down upon encountering an `Idle` instruction in the ISR

To exit power-down core/peripherals mode, the DSP responds to an interrupt and (after 500 cycles to re-stabilize the PLL) resumes executing instructions with the instruction after the Idle.

Working with External Bus Masters

The ADSP-2191 processor can relinquish control of data and address buses to an external device. The external device requests the bus by asserting (low) the bus request signal, $\overline{\texttt{BR}}$. The $\overline{\texttt{BR}}$ signal is an asynchronous input, arbitrated with core and peripheral requests. External bus requests have the lowest priority inside the DSP. If no other internal request is pending, the external bus request is granted. Due to synchronizer and arbitration delays, bus grants are provided with a minimum of three peripheral clock delays. The ADSP-2191 responds to the bus grant by:

1. Three-stating the data and address buses and the $\overline{\texttt{MS3-0}}$, $\overline{\texttt{BMS}}$, $\overline{\texttt{IOMS}}$, $\overline{\texttt{RD}}$, and $\overline{\texttt{WR}}$ output drivers.
2. Asserting the bus grant ($\overline{\texttt{BG}}$) signal.

Please make sure to include 10 kΩ pull-up resistors on the $\overline{\texttt{MSx}}$, $\overline{\texttt{BMS}}$, $\overline{\texttt{IOMS}}$, $\overline{\texttt{RD}}$, and $\overline{\texttt{WR}}$ signals, to ensure that they are held in a valid inactive state if these signals are used in the system's design.

The ADSP-2191 halts program execution if the bus is granted to an external device and an instruction fetch or data read/write request is made to external general-purpose or peripheral memory spaces. If an instruction requires two external memory read accesses, the bus is not granted between the two accesses. If an instruction requires an external memory read and an external memory write access, the bus may be granted between the two accesses. The external memory interface can be configured so that the core will have exclusive use of the interface. DMA and bus requests will be granted. When the external device releases $\overline{\texttt{BR}}$, the DSP releases $\overline{\texttt{BG}}$ and continues program execution from the point at which it stopped.

The bus request feature operates at all times, including when the processor is booting and when $\overline{\texttt{RESET}}$ is active. During $\overline{\texttt{RESET}}$, $\overline{\texttt{BG}}$ is asserted in the same cycle that $\overline{\texttt{BR}}$ is recognized. During booting, the bus is granted after completion of loading of the current byte (including any waitstates). Using bus request during booting is one way to bring the booting operation under control of a host.

The ADSP-2191 processor also has a Bus Grant Hung ($\overline{\texttt{BGH}}$) output, which lets it operate in a multiprocessor system with a minimum number of wasted cycles. The $\overline{\texttt{BGH}}$ pin asserts when the ADSP-2191 processor is ready to execute an instruction but is stopped because the external bus is granted to another device. The other device can release the bus by de-asserting bus request. Once the bus is released, the ADSP-2191 processor de-asserts $\overline{\texttt{BG}}$ and $\overline{\texttt{BGH}}$ and executes the external access.

If the ADSP-2191 processor is performing an external access when the $\overline{\texttt{BR}}$ signal is asserted, it will not grant the buses until the cycle after the access completes. The sequence of events is illustrated in Figure 14-5 on page 14-37. The entire instruction does not need to be completed when the bus is granted.

When the $\overline{\texttt{BR}}$ input is released, the ADSP-2191 processor releases the $\overline{\texttt{BG}}$ signal, reenables the output drivers and continues program execution from the point where it stopped. $\overline{\texttt{BG}}$ is always de-asserted in the same cycle that the removal of $\overline{\texttt{BR}}$ is recognized. Refer to the data sheet for exact timing relationships.

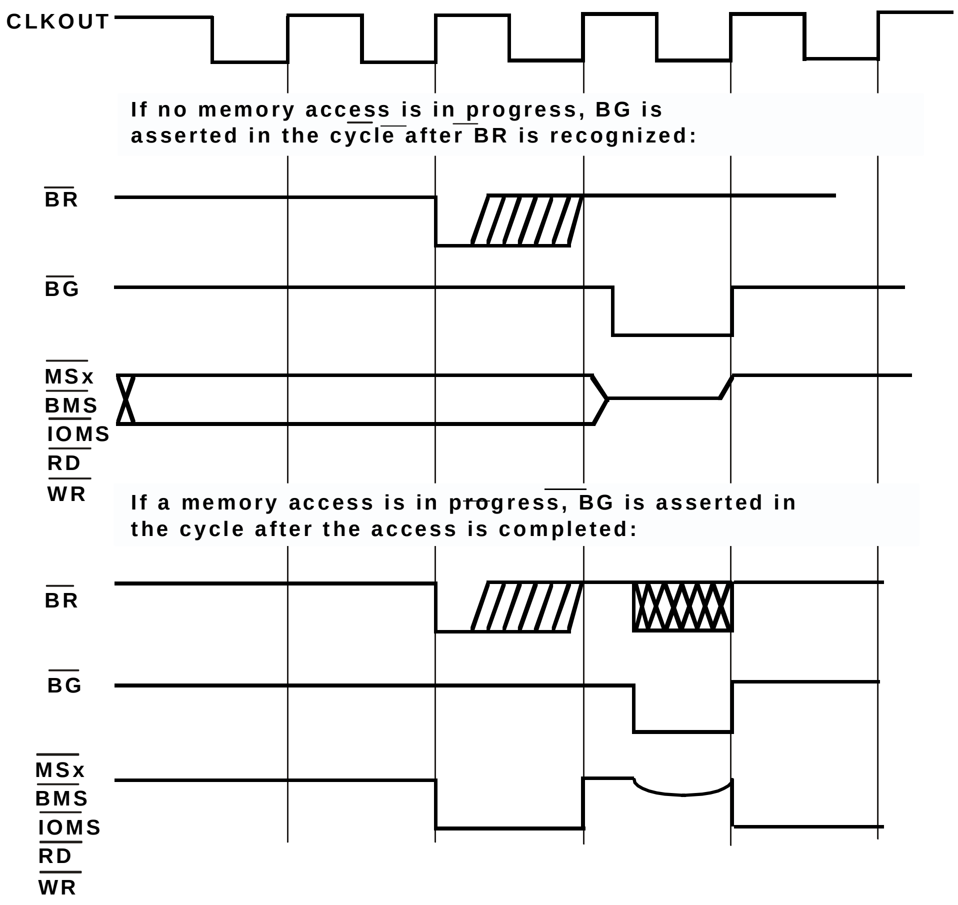

CLKOUT
If no memory access is in progress, BG is asserted in the cycle after BR is recognized:
BR
BG
MSx
BMS
IOMS
RD
WR
If a memory access is in progress, BG is asserted in the cycle after the access is completed:
BR
BG
MSx
BMS
IOMS
RD
WR

Figure 14-7. Bus Request (with or without External Access)

Recommended Reading

The text *High-Speed Digital Design: A Handbook of Black Magic* is recommended for further reading. This book is a technical reference that covers the problems encountered in state-of-the-art, high-frequency digital circuit design, and 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

Reference: Johnson & Graham, *High-Speed Digital Design: A Handbook of Black Magic*, Prentice Hall, Inc., ISBN 0-13-395724-1

Programmable Flags Example

The following sample code shows how to set up the programmable flags; configure the flag interrupt sources, sensitivities, and polarities; configure the interrupt priorities; and mask and enable interrupts.

2191_GPIO.asm

```
#include <def2191.h>

/* GLOBAL & EXTERNAL DECLARATIONS */

.GLOBAL Start;

/* DM data */
.section/data data1;

/* Program memory code */

.SECTION /pm program;
Start:
_main:
   CALL Initialize_GPIO; /* Initialize GPIO */
   CALL Initialize_Interrupts; /* Initialize Interrupts */

wait_forever:
   JUMP wait_forever;

/* INITIALIZE GENERAL PURPOSE FLAGS */

.SECTION /pm program;
Initialize_GPIO:
   IOPG = General_Purpose_IO;
   AR = 0x000F;    /* Configure FLAGS 0, 1, 2, 3 as outputs, */
                   /* and 4, 5 as inputs */
```

```
    IO(DIR) = AR;

    AR = 0x0010;
    IO(MASKAS) = AR;         /* Unmask FLAG 4 */
                             /* as GPIO interrupt source 0 */

    AR = 0x0020;
    IO(MASKBS) = AR;         /* Unmask FLAG 5 */
                             /* as GPIO interrupt source 1 */

    AR = 0x0000;
    IO(FSPRC) = AR;   /* Select FLAG polarity as active High - */
                   /* NOTE: Depends on hardware implementation */

    AR = 0x0000;
    IO(FSSRC) = AR;   /* Select Flag input as level sensitive */

    RTS;

 /* INTERRUPT PRIORITY CONFIGURATION */

    Initialize_Interrupts:
    IOPG = 0;
    AR = IO(SYSCR);   /* Map Interrupt Vector Table to Page 0 */
    AR = SETBIT 4 OF AR;
    IO(SYSCR) = AR;

    DIS INT;   /* Disable all interrupts */
    IRPTL = 0x0;   /* Clear all interrupts */
    ICNTL = 0x0;   /* Interrupt nesting disable */
    IMASK = 0;   /* Mask all interrupts */

    /* Set up Interrupt Priorities */
    IOPG = Interrupt_Controller_Page;
```

```
   ar = 0xBB21;    /* Assign GPIO Interrupt 0 priority of 1, */
                   /* GPIO Interrupt 1 priority of 2 */
   IO(IPR3) = AR;
   AR = 0xBBBB;   /* Assign the remainder with lowest priority */
   IO(IPR0) = AR;
   IO(IPR1) = AR;
   IO(IPR2) = AR;

   AY0 = IMASK;
   AY1 = 0x0060;   /* Unmask GPIO 0 and 1 Interrupts */
   AR = AY0 OR AY1;
   IMASK = AR;

   ENA INT;   /* Globally enable all interrupts */

   RTS;
```

GPIO_ISR.asm

```
#include <def2191.h>

/* EXTERNAL DECLARATIONS */

.EXTERN Start;

/* DM data */

.SECTION /dm data1;
.VAR counter_int5 = 0;
.VAR counter_int6 = 0;
.VAR Int_0_Polarity;
.VAR Int_1_Polarity;

/* PM Reset interrupt vector code */
```

```
.section/pm IVreset;
   JUMP Start;
   NOP; NOP; NOP;

/* GPIO 0 ISR */

.section/pm IVint5;
   ENA SR;
   AY1 = IOPG;

   AR = DM(counter_int5);     /* Interrupt counter */
                              /* for debug purposes */
   AR = AR + 1;
   DM(counter_int5) = AR;

   IOPG = General_Purpose_IO;
   AY0 = 0x0010;
Wait_0_Depressed:
   AX0 = IO(FLAGC);
   AR =AX0 AND AY0;
   IF NE JUMP Wait_0_Depressed;

   AX0 = 0x0003;
   AX1 = DM(Int_0_Polarity);
   AR = TGLBIT 0x0 OF AX1;    /* Toggle Status flag */
   IF EQ JUMP TURN_0_OFF;   /* Determine if GPIO was ON or OFF */
TURN_0_ON:
   IO(FLAGS) = AX0;   /* Turn ON GPIOS 0, 1 */
   DM(Int_0_Polarity) = AR;
   IOPG = AY1;
   DIS SR;
   RTI;
```

```
TURN_0_OFF:
   IO(FLAGC) = AX0;   /* Turn OFF GPIOS 0, 1 */
   DM(Int_0_Polarity) = AR;
   IOPG = AY1;
   DIS SR;
   RTI;

/* GPIO 1 ISR */

.section/pm IVint6;
   ENA SR;
   AY1 = IOPG;

   AR = DM(counter_int6);         /* Interrupt counter */
                                  /* for debug purposes */
   AR = AR + 1;
   DM(counter_int6) = AR;

   IOPG = General_Purpose_IO;
   AY0 = 0x0020;
Wait_1_Depressed:
   AX0 = IO(FLAGC);
   AR = AX0 AND AY0;
   IF NE JUMP Wait_1_Depressed;

   AX0 = 0x000C;
   AX1 = DM(Int_1_Polarity);
   AR = TGLBIT 0x0 OF AX1; /* Toggle Status flag */
   IF EQ JUMP TURN_1_OFF; /* Determine if GPIO was ON or OFF */
TURN_1_ON:
   AX0 = 0x000C;
   IO(FLAGS) = AX0;    /* Turn ON GPIOS 2, 3 */
   DM(Int_1_Polarity) = AR;
   IOPG = AY1;
```

```
    DIS SR;
    RTI;

TURN_1_OFF:
    IO(FLAGC) = AX0; /* Turn OFF GPIOS 2, 3 */
    DM(Int_1_Polarity) = AR;
    IOPG = AY1;
    DIS SR;
    RTI;
```

A ADSP-219X DSP CORE REGISTERS

The DSP core has general-purpose and dedicated registers in each of its functional blocks. The register reference information for each functional block includes bit definitions, initialization values, and (for system control registers) memory-mapped addresses. The following information on each type of register is provided:

Overview

Outside of the DSP core, a set of registers control the I/O peripherals. For information on these product specific registers, see "ADSP-2191 DSP I/O Registers" on page B-1.

When writing DSP programs, it is often necessary to set, clear, or test bits in the DSP's registers. While these bit operations can all be done by referring to the bit's location within a register or (for some operations) the register's address with a hexadecimal number, it is much easier to use symbols that correspond to the bit's or register's name. For convenience and consistency, Analog Devices provides a header file that provides these bit and registers definitions. For core register definitions, see the "Register and Bit #Defines File (def219x.h)" on page A-22. For off-core register definitions, see the "Register and Bit #define File (def2191.h)" on page B-115.

Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) the register's reserved bits.

Core Registers Summary

The DSP has three categories of registers:

- Core registers
- System control registers
- I/O registers

Table A-1 on page A-3 lists and describes the DSP's core registers and system control registers. Also, the DSP core registers divide into register group (Dreg, Reg1, Reg2, and Reg3) based on their opcode identifiers and functions as shown in Table A-2 on page A-4. For more information on how registers may be used within instructions, see the ADSP-219x *DSP Instruction Set Reference*.

Register Load Latencies

An effect latency occurs when some instructions write or load a value into a register, which changes the value of one or more bits in the register. Effect latency refers to the time it takes after the write or load instruction for the effect of the new value to become available for other instructions to use.

Table A-1. Core Registers

Type	Registers	Function
Status	ASTAT MSTAT SSTAT (read-only)	Arithmetic status flags Mode control and status flags System status
Computational Units	AX0, AX1, AY0, AY1, AR, AF, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SI, SE, SB, SR0, SR1, SR2	Data register file registers provide Xop and Yop inputs for computations. AR, SR, and MR receive results. In this text, the word Dreg denotes unrestricted use of data registers as a data register file, while the words XOP and YOP denote restricted use. The data registers (except AF, SE, and SB) serve as a register file, for unconditional, single-function instructions.
Shifter	SE SB	Shifter exponent register Shifter block exponent register
Program flow	CCODE LPSTACKA LPSTACKP STACKA STACKP	Software condition register Loop stack address register, 16 address LSBs Loop stack page register, 8 address MSBs PC stack address register, 16 address LSBs PC stack page register, 8 address MSBs
Interrupt	ICNTL IMASK IRPTL	Interrupt control register Interrupt mask register Interrupt latch register
DAG address	I0, I1, I2, I3 I4, I5, I6, I7	DAG1 index registers DAG2 index registers
	M0, M1, M2, M3 M4, M5, M6, M7	DAG1 modify registers DAG2 modify registers
	L0, L1, L2, L3 L4, L5, L6, L7	DAG1 length registers DAG2 length registers
System control	B0, B1, B2, B3, B4, B5, B6, B7, CACTL	DAG1 base address registers (B0-3), DAG2 base address registers (B4-7), Cache control

Table A-1. Core Registers (Cont'd)

Type	Registers	Function
Page	DMPG1 DMPG2 IJPG IOPG	DAG1 page register, 8 address MSBs DAG2 page register, 8 address MSBs Indirect jump page register, 8 address MSBs I/O memory page register, 8 address MSBs
Bus exchange	PX	Holds eight LSBs of 24-bit memory data for transfers between memory and data registers only.

Table A-2. ADSP-219x DSP Core Registers

RGP/Address	Register Groups (RGP)			
Address	**00 (DREG)**	**01 (REG1)**	**10 (REG2)**	**11 (REG3)**
0000	AX0	I0	I4	ASTAT
0001	AX1	I1	I5	MSTAT
0010	MX0	I2	I6	SSTAT
0011	MX1	I3	I7	LPSTACKP
0100	AY0	M0	M4	CCODE
0101	AY1	M1	M5	SE
0110	MY0	M2	M6	SB
0111	MY1	M3	M7	PX
1000	MR2	L0	L4	DMPG1
1001	SR2	L1	L5	DMPG2
1010	AR	L2	L6	IOPG
1011	SI	L3	L7	IJPG
1100	MR1	IMASK	Reserved	Reserved
1101	SR1	IRPTL	Reserved	Reserved
1110	MR0	ICNTL	CNTR	Reserved
1111	SR0	STACKA	LPCSTACKA	STACKP

Effect latency values are given in terms of instruction cycles. A `0` latency means that the effect of the new value is available on the next instruction following the write or load instruction. For register changes that have an effect latency greater than `0`, do not try to use the register right after writing or loading a new value to avoid using the old value. Table A-3 on page A-6 gives the effect latencies for writes or loads of various interrupt and status registers.

Table A-3. Effect Latencies for Register Changes

Register	Bits	REG = value	ENA/DIS mode	POP STS	SET/CLR INT
ASTAT	All	1 cycle	NA	0 cycles	NA
CCODE	All	1 cycle	NA	NA	NA
CNTR	All	1 cycle[1]	NA	NA	NA
ICNTL	All	1 cycle	NA	NA	0 cycles
IMASK	All	1 cycle	NA	0 cycles	NA
MSTAT	SEC_REG	1 cycle	0 cycles	1 cycle	NA
	BIT_REV	3 cycles	0 cycles	3 cycles	NA
	AV_LATCH	0 cycles	0 cycles	0 cycles	NA
	AR_SAT	1 cycle	0 cycles	1 cycle	NA
	M_MODE	1 cycle	0 cycles	1 cycle	NA
	SEC_DAG	3 cycles	0 cycles	3 cycles	NA
CACTL	CPE	5 cycles	NA	NA	NA
	CDE	5 cycles	NA	NA	NA
	CFZ	4 cycles	NA	NA	NA

1 This latency applies only to `IF COND` instructions, not to the `DO UNTIL` instruction. Loading the `CNTR` register has 0 effect latency for the `DO UNTIL` instruction.

A `PUSH` or `POP PC` has one cycle of latency for all `SSTAT` register bits, but a `PUSH` or `POP LOOP` or `STS` has one cycle of latency only for the `STKOVERFLOW` bit in the `SSTAT` register.

When loading some group 2 and group 3 registers (see Table A-3 on page A-6), the effect of the new value is not immediately available to subsequent instructions that might use it. For interlocked registers (DAG address and page registers, IOPG, IJPG), the DSP automatically inserts stall cycles as needed, but for noninterlocked registers (to accommodate the required latency) programs must insert either the necessary number of NOP instructions or other instructions that are not dependent upon the effect of the new value.

The noninterlocked registers are:

- Status registers ASTAT and MSTAT
- Condition code register CCODE
- Interrupt control register ICNTL

The number of NOP instructions to insert is specific to the register and the load instruction as shown in Table A-3 on page A-6. A zero (0) latency indicates that the new value is effective on the next cycle after the load instruction executes. An *n* latency indicates that the effect of the new value is available up to *n* cycles after the load instruction executes. When using a modified register before the required latency, the DSP provides the register's old value.

Since unscheduled or unexpected events (such as interrupts and DMA operations) often interrupt normal program flow, do not rely on these load latencies when structuring program flow. A delay in executing a subsequent instruction based on a newly loaded register could result in erroneous results—whether the subsequent instruction is based on the effect of the register's new or old value.

Load latency applies only to the time it takes the loaded value to affect the change in operation, not to the number of cycles required to load the new value. A loaded value is always available to a read access on the next instruction cycle.

Core Status Registers

The DSP's control and status system registers configure how the processor core operates and indicate the status of many processor core operations. Table A-4 on page A-8 lists the processor core's control and status registers with their initialization values. Descriptions of each register follow.

Table A-4. Core Status Registers

Register Name & Page Reference	Initialization After Reset
"Arithmetic Status (ASTAT) Register" on page A-8	b#0 0000 0000
"Mode Status (MSTAT) Register" on page A-8	b#000 000
"System Status (SSTAT) Register" on page A-10	b#0000 0000

Arithmetic Status (ASTAT) Register

Figure A-1 on page A-9 shows this is a non-memory mapped, register group 3 register (REG3). The DSP updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

Mode Status (MSTAT) Register

Figure A-2 on page A-10 shows this is a non-memory mapped, register group 3 register (REG3). For more information on using bits in this register, see "Secondary (Alternate) Data Registers" on page 2-63, "Addressing with Bit-Reversed Addresses" on page 4-16, "Latching ALU Result Overflow Status" on page 2-12,"Saturating ALU Results on Overflow" on page 2-12 "Numeric Formats" on page C-1, and "Secondary (Alternate) Data Registers" on page 2-63.

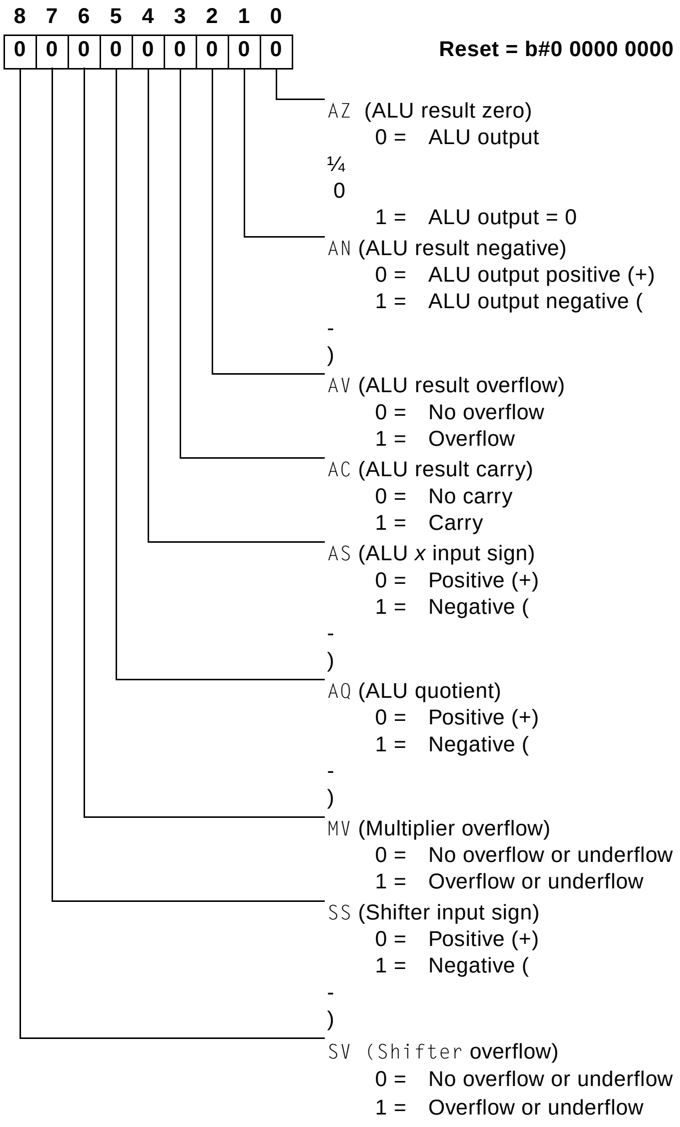


Figure A-1. ASTAT Register Bit Definitions

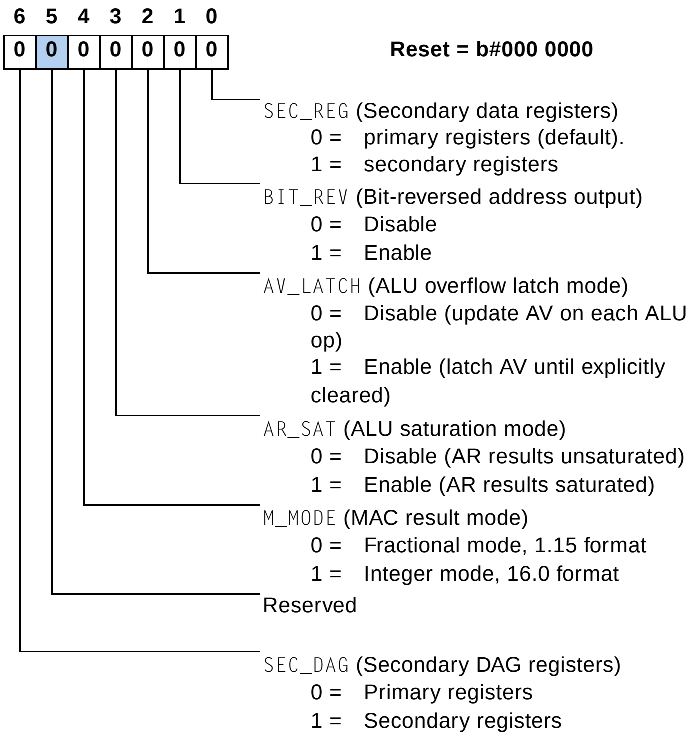

6 5 4 3 2 1 0
0 0 0 0 0 0 0
Reset = b#000 0000
SEC_REG (Secondary data registers)
0 = primary registers (default).
1 = secondary registers
BIT_REV (Bit-reversed address output)
0 = Disable
1 = Enable
AV_LATCH (ALU overflow latch mode)
0 = Disable (update AV on each ALU op)
1 = Enable (latch AV until explicitly cleared)
AR_SAT (ALU saturation mode)
0 = Disable (AR results unsaturated)
1 = Enable (AR results saturated)
M_MODE (MAC result mode)
0 = Fractional mode, 1.15 format
1 = Integer mode, 16.0 format
Reserved
SEC_DAG (Secondary DAG registers)
0 = Primary registers
1 = Secondary registers

Figure A-2. MSTAT Register Bit Definitions

System Status (SSTAT) Register

Figure A-3 on page A-11 shows this is a non-memory mapped, register group 3 register (REG3).

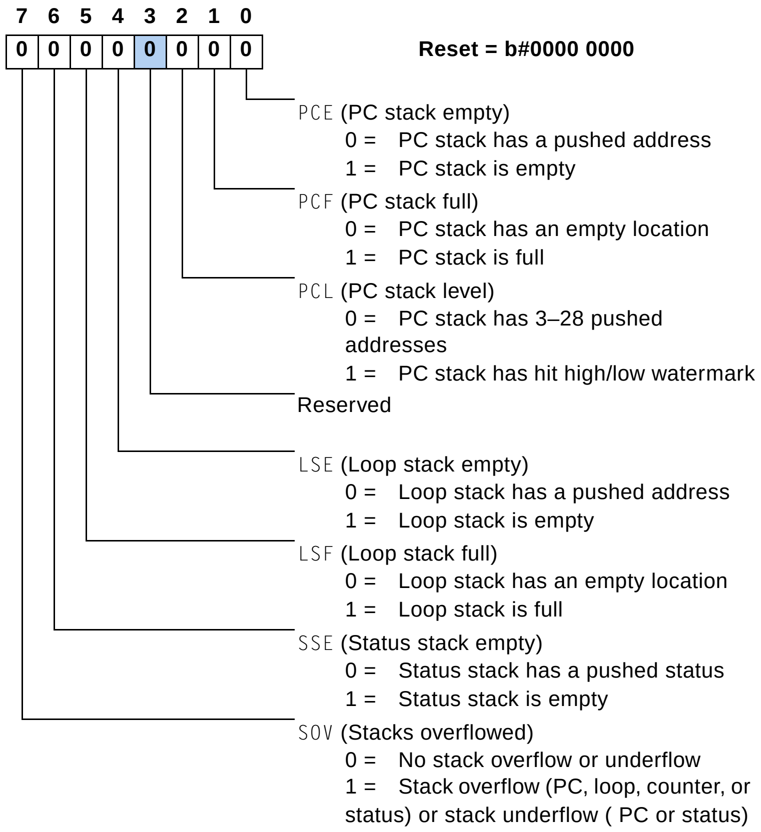

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
Reset = b#0000 0000
PCE (PC stack empty)
0 = PC stack has a pushed address
1 = PC stack is empty
PCF (PC stack full)
0 = PC stack has an empty location
1 = PC stack is full
PCL (PC stack level)
0 = PC stack has 3–28 pushed addresses
1 = PC stack has hit high/low watermark
Reserved
LSE (Loop stack empty)
0 = Loop stack has a pushed address
1 = Loop stack is empty
LSF (Loop stack full)
0 = Loop stack has an empty location
1 = Loop stack is full
SSE (Status stack empty)
0 = Status stack has a pushed status
1 = Status stack is empty
SOV (Stacks overflowed)
0 = No stack overflow or underflow
1 = Stack overflow (PC, loop, counter, or status) or stack underflow (PC or status)

Figure A-3. SSTAT Register Bit Definitions

Computational Unit Registers

The DSP's computational registers store data and results for the ALU, multiplier, and shifter. The inputs and outputs for processing element operations go through these registers.

The PX register lets programs transfer data between the data buses, but cannot be an input or output in a calculation.

Table A-5. Computational Unit Registers

Register	Initialization After Reset
"Data Register File (Dreg) Registers" on page A-12	Undefined
"ALU X Input (AX0, AX1) Registers and ALU Y Input (AY0, AY1) Registers" on page A-13	Undefined
"ALU Results (AR) Register" on page A-13	Undefined
"ALU Feedback (AF) Register" on page A-13	Undefined
"Multiplier X Input (MX0, MX1) Registers and Multiplier Y Input (MY0, MY1) Registers" on page A-13	Undefined
"Multiplier Results (MR2, MR1, MR0) Registers" on page A-14	Undefined
"Shifter Input (SI) Register" on page A-14	Undefined
"Shifter Exponent (SE) Register and Shifter Block Exponent (SB) Register" on page A-14	Undefined
"Shifter Result (SR2, SR1, SR0) Registers" on page A-14	Undefined

Data Register File (Dreg) Registers

These are non-memory mapped, register group 0 registers (DREG). For unconditional, single-function instructions, the DSP has a data register file—a set of 16-bit data registers that transfer data between the data buses and the computation units. These registers also provides local storage for operands and results. For more information on how to use these registers, see "Data Register File" on page 2-61. The registers in the data register file include: AX0, AX1, MX0, MX1, AY0, AY1, MY0, MY1, MR2, SR2, AR, SI, MR1, SR1, MR0, and SR0.

ALU X Input (AX0, AX1) Registers and ALU Y Input (AY0, AY1) Registers

These are non-memory mapped, register group 0 registers. For conditional and/or multifunction instructions, some restrictions apply to data register usage. The registers that may provide Xop and Yop input to the ALU for conditional and/or multifunction instructions include: AX0, AX1, AY0, and AY1. For more information on how to use these registers, see "Multifunction Computations" on page 2-64.

ALU Results (AR) Register

This is a non-memory mapped, register group 0 register.The ALU places its results in the 16-bit AR register. For more information on how to use this register, see "Arithmetic Logic Unit (ALU)" on page 2-18.

ALU Feedback (AF) Register

This is a non-memory mapped, register group 0 register.The ALU can place its results in the 16-bit AF register. For more information on how to use this register, see "Arithmetic Logic Unit (ALU)" on page 2-18.

Multiplier X Input (MX0, MX1) Registers and Multiplier Y Input (MY0, MY1) Registers

These are non-memory mapped, register group 0 registers. For conditional and/or multifunction instructions, some restrictions apply to data register usage. The registers that may provide Xop and Yop input to the multiplier for conditional and/or multifunction instructions include: MX0, MX1, MY0, and MY1. For more information on how to use these registers, see "Multifunction Computations" on page 2-64.

Multiplier Results (MR2, MR1, MR0) Registers

These are non-memory mapped, register group 0 registers. The multiplier places results in the combined multiplier result register, MR. For more information on result register fields, see "Multiply/Accumulates (Multiplier)" on page 2-30.

Shifter Input (SI) Register

This is a non-memory mapped, register group 0 register. For conditional and/or multifunction instructions, some restrictions apply to data register usage. SI is the only registers that may provide input to the shifter for conditional and/or multifunction instructions. For more information on how to use this register, see "Multifunction Computations" on page 2-64.

Shifter Exponent (SE) Register and Shifter Block Exponent (SB) Register

These are non-memory mapped, register group 3 registers. These register hold exponent information for the shifter. For more information on how to use these registers, see "Barrel Shifter (Shifter)" on page 2-39.

The SB and SE registers are 16 bits in length, but all shifter instructions that use these registers as operands or update these registers with result values do not use the full width of these registers. Shifter instructions treat SB as being a 5-bit twos complement register and treat SE as being an 8-bit twos complement register.

Shifter Result (SR2, SR1, SR0) Registers

These are non-memory mapped, register group 0 registers. The Shifter places results in the shift result register, SR. Optionally, the multiplier can use SR as a second (dual) accumulator. For more information on how to use this registers, see "Barrel Shifter (Shifter)" on page 2-39.

Program Sequencer Registers

The DSP's Program Sequencer registers hold page addresses, stack addresses, and other information for determining program execution.

Refer to "System Interrupt Controller Registers" on page B-21 for additional information.

Table A-6. Program Sequencer Registers

Register	**Initialization After Reset**
"Interrupt Mask (IMASK) Register and Interrupt Latch (IRPTL) Register" on page A-15	0x0000
"Interrupt Control (ICNTL) Register" on page A-15	0x0000
"Indirect Jump Page (IJPG) Register" on page A-17	0x00
"PC Stack Page (STACKP) Register and PC Stack Address (STACKA) Register" on page A-17	Undefined
"Loop Stack Page (LPSTACKP) Register and Loop Stack Address (LPSTACKA) Register" on page A-17	Undefined
"Counter (CNTR) Register" on page A-18	Undefined
"Condition Code (CCODE) Register" on page A-18	Undefined
"Cache Control (CACTL) Register" on page A-19	b#101n nnnn

Interrupt Mask (IMASK) Register and Interrupt Latch (IRPTL) Register

Figure A-4 on page A-16 shows the bits for these are a non-memory mapped, register group 1 registers (REG1).

Interrupt Control (ICNTL) Register

Figure A-5 on page A-16 shows this is a non-memory mapped, register group 1 register (REG1). The reset value for this register is `0x0000`.

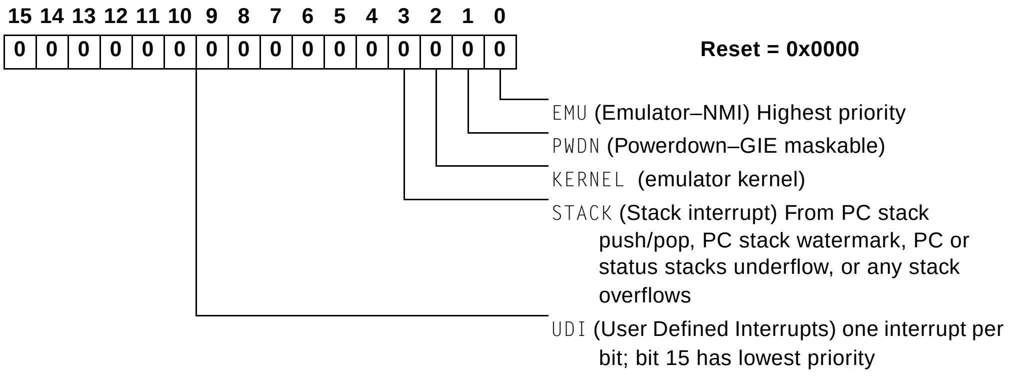

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EMU (Emulator–NMI) Highest priority
PWDN (Powerdown–GIE maskable)
KERNEL (emulator kernel)
STACK (Stack interrupt) From PC stack push/pop, PC stack watermark, PC or status stacks underflow, or any stack overflows
UDI (User Defined Interrupts) one interrupt per bit; bit 15 has lowest priority

Figure A-4. IMASK and IRPTL Registers Bit Definitions

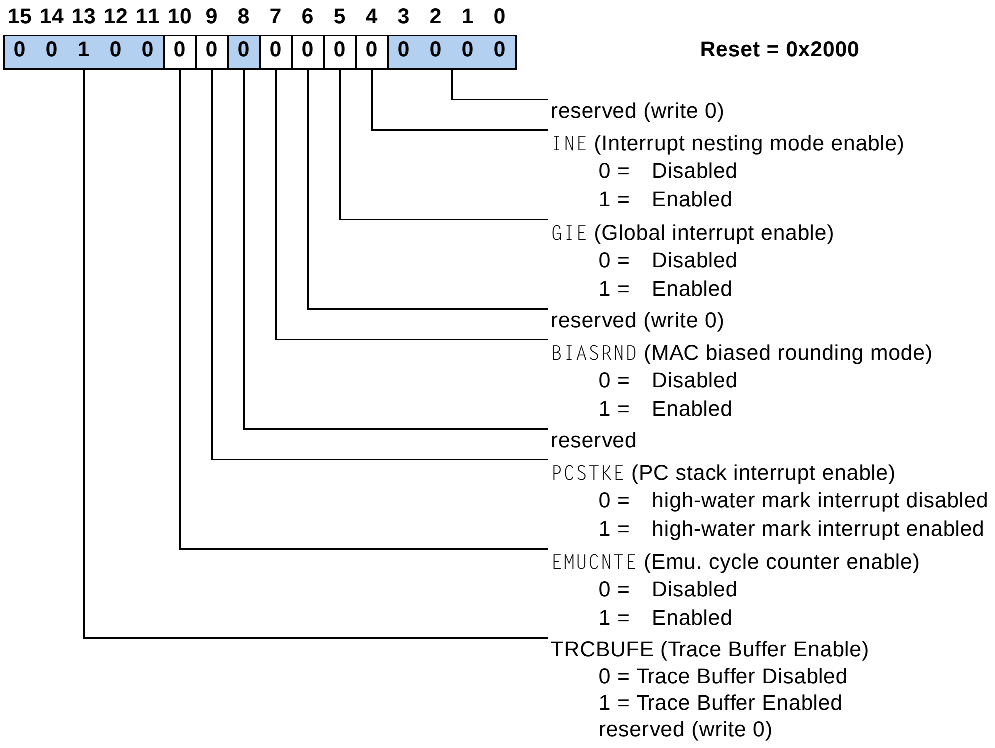

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x2000
reserved (write 0)
INE (Interrupt nesting mode enable)
0 = Disabled
1 = Enabled
GIE (Global interrupt enable)
0 = Disabled
1 = Enabled
reserved (write 0)
BIASRND (MAC biased rounding mode)
0 = Disabled
1 = Enabled
reserved
PCSTKE (PC stack interrupt enable)
0 = high-water mark interrupt disabled
1 = high-water mark interrupt enabled
EMUCNTE (Emu. cycle counter enable)
0 = Disabled
1 = Enabled
TRCBUFE (Trace Buffer Enable)
0 = Trace Buffer Disabled
1 = Trace Buffer Enabled
reserved (write 0)

Figure A-5. ICNTL Register Bit Definitions

Indirect Jump Page (IJPG) Register

This is a non-memory mapped, register group 3 register (REG3). The reset value for this register is 0x00. For information on using this register, see "Indirect Jump Page (IJPG) Register" on page 3-16.

PC Stack Page (STACKP) Register and PC Stack Address (STACKA) Register

These are non-memory mapped, register group 1 (REG1) and register group 3 registers (REG3). The PC Stack Page (STACKP) and PC Stack Address (STACKA) registers hold the top entry in the Program Counter (PC) address stack. The upper 8 bits of the address go into STACKP, and the lower 16 bits go into STACKA. The PC stack is 33 levels deep.

On JUMP, CALL, DO/UNTIL (loop), and PUSH PC instructions, the DSP pushes the PC address onto this stack, loading the STACKP and STACKA registers. On RTS/RTI (return) and POP PC instructions, the DSP pops the STACKP:STACKA address off of this stack, loading the PC register.

For information on using these registers, see "Stacks and Sequencing" on page 3-36.

Loop Stack Page (LPSTACKP) Register and Loop Stack Address (LPSTACKA) Register

These are non-memory mapped, register group 2 and 3 registers (REG2, REG3). The Loop Stack Page (LPSTACKP) and Loop Stack Address (LPSTACKA) registers hold the top entry in the loop stack. The upper 8 bits of the address go into LPSTACKP, and the lower 16 bits go into LPSTACKA. The loop stack is 8 levels deep.

On DO/UNTIL (loop) instructions, the DSP pushes the end of loop address onto this stack, loading the LPSTACKP and LPSTACKA registers. On PUSH LOOP instructions, the DSP pushes the (explicitly loaded) contents of the LPSTACKP and LPSTACKA registers onto this stack.

At the end of a loop (counter decrements to zero), the DSP pops the LPSTACKP:LPSTACKA address off of this stack, loading the PC register with the next address after the end of the loop. On POP LOOP instructions, the DSP pops the contents of the LPSTACKP and LPSTACKA registers off of this stack.

At the start of a loop the PC (start of loop address) is pushed onto the loop begin stack (STACKP:STACKA registers) and the end of loop address is pushed onto the loop end stack (LPSTACKP:LPSTACKA registers). If it is a counter-based loop (DO/UNTIL CE), the loop count (CNTR register) is pushed onto the counter stack.

For information on using these registers, see "Stacks and Sequencing" on page 3-36.

Counter (CNTR) Register

This is a non-memory mapped, register group 2 register (REG2). The DSP loads the loop counter stack from CNTR or DO/UNTIL or PUSH LOOP instructions. For information on using this register, see "Loops and Sequencing" on page 3-20 and "Stacks and Sequencing" on page 3-36.

Condition Code (CCODE) Register

This is a non-memory mapped, register group 3 register (REG3). Using the CCODE register, conditional instructions may base execution on a comparison of the CCODE value (user-selected) and the SWCOND condition (DSP status). The CCODE register holds a value between 0x0 and 0xF, which the instruction tests against when the conditional instruction uses SWCOND or NOT SWCOND. Note that the CCODE register has a one cycle effect latency.

Table A-7. CCODE Register Bit Definitions

CCODE	Software Condition	
Value	SWCOND (1010)	NOT SWCOND (1011)
0x00	PF0 pin high	PF0 pin low
0x01	PF1 pin high	PF1 pin low
0x02	PF2 pin high	PF2 pin low
0x03	PF3 pin high	PF3 pin low
0x04	PF4 pin high	PF4 pin low
0x05	PF5 pin high	PF5 pin low
0x06	PF6 pin high	PF6 pin low
0x07	PF7 pin high	PF7 pin low
0x08	AS (ALU result signed)	NOT AS (ALU input not signed)
0x09	SV (SR result overflow)	NOT SV (SR result not overflow)
0x0A	PF8 pin high	PF8 pin low
0x0B	PF9 pin high	PF9 pin low
0x0C	PF10 pin high	PF10 pin low
0x0D	PF11 pin high	PF11 pin low
0x0E	PF12 pin high	PF12 pin low
0x0F	PF13 pin high	PF13 pin low

Cache Control (CACTL) Register

Figure A-6 on page A-20 shows this is a register-memory mapped register at address `Reg(0x0F)`.

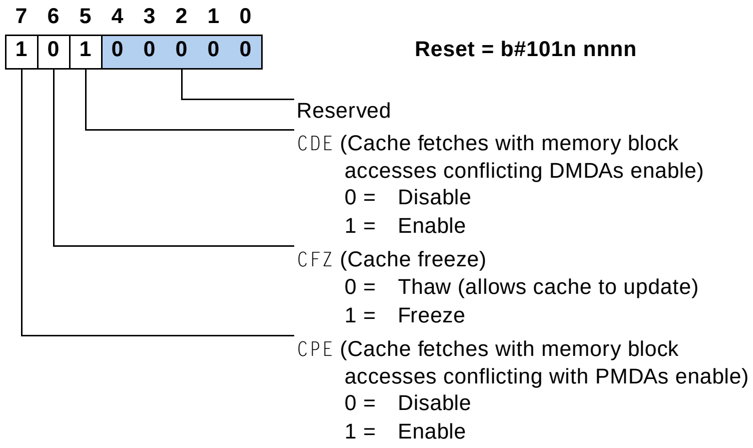

7 6 5 4 3 2 1 0
1 0 1 0 0 0 0 0
Reset = b#101n nnnn
Reserved
CDE (Cache fetches with memory block accesses conflicting DMDAs enable)
0 = Disable
1 = Enable
CFZ (Cache freeze)
0 = Thaw (allows cache to update)
1 = Freeze
CPE (Cache fetches with memory block accesses conflicting with PMDAs enable)
0 = Disable
1 = Enable

Figure A-6. CACTL Register Bit Definitions

Data Address Generator Registers

The DSP's Data Address Generator (DAG) registers hold data addresses, modify values, and circular buffer configurations. Using these registers, the DAGs can automatically increment addressing for ranges of data locations (a buffer).

Table A-8. Data Address Generator Registers

Register	Initialization After Reset
"Index (Ix) Registers" on page A-21	Undefined
"Modify (Mx) Registers" on page A-21	Undefined
"Length (Lx) Registers and Base (Bx) Registers" on page A-21	Undefined
"Data Memory Page (DMPGx) Registers" on page A-21	0x00

Index (Ix) Registers

These are non-memory mapped, register group 1 and 2 registers (REG1 and REG2). The Data Address Generators store addresses in Index registers (`I0-I3` for DAG1 and `I4-I7` for DAG2). An index register holds an address and acts as a pointer to memory. For more information, see "DAG Operations" on page 4-9.

Modify (Mx) Registers

These are non-memory mapped, register group 1 and 2 registers (REG1 and REG2). The Data Address Generators update stored addresses using Modify registers (`M0-M3` for DAG1 and `M4-M7` for DAG2). A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For more information, see "DAG Operations" on page 4-9.

Length (Lx) Registers and Base (Bx) Registers

The Length registers are non-memory mapped, register group 1 and 2 registers (REG1 and REG2). The Base registers are memory-mapped in register-memory at addresses: `B0=Reg(0x00)` through `B7=Reg(0x07)`.

The Data Address Generators control circular buffering operations with Length and Base registers (`L0-L3` and `B0-B3` for DAG1 and `L4-L7` and `B4-B7` for DAG2). Length and base registers setup the range of addresses and the starting address for a circular buffer. For more information, see "DAG Operations" on page 4-9.

Data Memory Page (DMPGx) Registers

This is a non-memory mapped, register group 3 register (REG3). The reset value for this register is 0x00. For information on using this register, see "Data Memory Page Registers (DMPGx)" on page 4-7.

Memory Interface Registers

The DSP's memory interface registers set up page access to I/O memory and provide an interface between the 24-bit and 16-bit data buses.

Table A-9. Memory Interface Registers

Register	Initialization After Reset
"PM Bus Exchange (PX) Register" on page A-22	Undefined
"I/O Memory Page (IOPG) Register" on page A-22	0x00

PM Bus Exchange (PX) Register

This is a non-memory mapped, register group 3 register (REG3). The PM Bus Exchange (PX) register permits data to flow between the PM and DM data buses. For more information on PX register usage, see "Internal Data Bus Exchange" on page 5-7.

I/O Memory Page (IOPG) Register

This is a non-memory mapped, register group 3 register (REG3). The reset value for this register is 0x00.

Register and Bit #Defines File (def219x.h)

The following example definitions file is for items common to all ADSP-219x DSPs. For the most current definitions file, use the version of this file that comes with the software development tools.

The version that appears in this appendix is included as a guide only.

```
/************************************************************
 *
 * def219x.h : $Revision: 1.4.12.1 $
 *
 * (c) Copyright 2000-2002 Analog Devices, Inc.
 *               All rights reserved.
 *
*************************************************************/

/*
** System register bit and address defines to symbolic names
** for DSP-219x DSPs.

/*---------------------------------------------------------------
The def219x.h file defines ADSP-219x DSP family common symbolic
names; for names that are unique to particular ADSP-219x family
DSPs, see that DSP's definitions file (such as the def2192-12.h)
instead. This (def219x.h) contains a list of macro "defines" that
let programs use symbolic include file names for the following
ADSP-219x facilities:

- system register bit definitions
- system register map

Here are some example uses:

mstat = ASTAT_AR_SAT | ASTAT_M_MODE;
     >> this ORs together the bitmask macros
  -OR-
ax0 = 0x0000;
ar = setbit ASTAT_AR_SAT_P of ax0;
     >> this uses the define of AR_SAT bit
ar = setbit ASTAT_M_MODE_P of ar;
     >> this uses the define of M_MODE bit
mstat = ar;
```

```
ccode = cond_SV;     >> uses the define of SV condition
ax0 = 0;
ar = 0;
ar = setbit ASTAT_SV_P of ax0;  >> uses the define of SV bit
astat = ar;
if swcond ar = ax0 xor 0x1000;

AR = setbit CACTL_CFZ_P of AX0;
     >> this uses the define of CACTL_CFZ_P bit
REG(CACTL) = AR;  >> uses def. for CACTL register's address

ax0 = 0x0800;
REG(B0) = ax0;  >> uses the define for B0 register's address

-------------------------------------------------------------*/

#ifndef __DEF219x_H_
#define __DEF219x_H_

#define MK_BMSK_( x ) (1<<x)
     /* Make a bit mask from a bit position */

//------------------------------------------------------------
//           System Register bit definitions
//------------------------------------------------------------

//*************************************************
//   ASTAT register
//*************************************************

// Bit Positions
#define ASTAT_AZ_P  0   // Bit 0: ALU result zero
#define ASTAT_AN_P  1   // Bit 1: ALU result negative
```

```
#define ASTAT_AV_P  2   // Bit 2: ALU overflow
#define ASTAT_AC_P  3   // Bit 3: ALU carry
#define ASTAT_AS_P  4   // Bit 4: ALU X input sign (ABS ops)
#define ASTAT_AQ_P  5   // Bit 5: ALU quotient (DIV ops)
#define ASTAT_MV_P  6   // Bit 6: Multiplier overflow
#define ASTAT_SS_P  7   // Bit 7: Shifter input sign
#define ASTAT_SV_P  8   // Bit 8: Shifter overflow

// Bit Masks
#define ASTAT_AZ  MK_BMSK_(ASTAT_AZ_P)  // ALU result zero
#define ASTAT_AN  MK_BMSK_(ASTAT_AN_P)  // ALU result negative
#define ASTAT_AV  MK_BMSK_(ASTAT_AV_P)  // ALU overflow
#define ASTAT_AC  MK_BMSK_(ASTAT_AC_P)  // ALU carry
#define ASTAT_AS  MK_BMSK_(ASTAT_AS_P)
           // ALU X input sign (ABS ops)
#define ASTAT_AQ  MK_BMSK_(ASTAT_AQ_P)  // ALU quotient (DIV ops)
#define ASTAT_MV  MK_BMSK_(ASTAT_MV_P)  // Multiplier overflow
#define ASTAT_SS  MK_BMSK_(ASTAT_SS_P)  // Shifter input sign
#define ASTAT_SV  MK_BMSK_(ASTAT_SV_P)  // Shifter overflow


//***************************************************
//   MSTAT register
//***************************************************

// Bit Positions
#define MSTAT_SEC_REG_P   0  // Bit  0: Sec. data reg enable
#define MSTAT_BIT_REV_P   1
         // Bit 1: Bit-reversed address output enable
#define MSTAT_AV_LATCH_P  2
         // Bit 2: ALU overflow latch mode select
#define MSTAT_AR_SAT_P   3
         // Bit 3: ALU saturation mode select
#define MSTAT_M_MODE_P   4
```

```
         // Bit 4: MAC result mode select
#define MSTAT_TIMER_P   5
         // Bit 5: Timer enable
#define MSTAT_SEC_DAG_P   6
         // Bit 6: Secondary DAG registers enable

// Bit Masks
#define MSTAT_SEC_REG    MK_BMSK_(MSTAT_SEC_REG_P )
     // Secondary data registers enable
#define MSTAT_BIT_REV    MK_BMSK_(MSTAT_BIT_REV_P )
     // Bit-reversed address output enable
#define MSTAT_AV_LATCH   MK_BMSK_(MSTAT_AV_LATCH_P)
     // ALU overflow latch mode select
#define MSTAT_AR_SAT     MK_BMSK_(MSTAT_AR_SAT_P  )
     // ALU saturation mode select
#define MSTAT_M_MODE     MK_BMSK_(MSTAT_M_MODE_P  )
     // MAC result mode select
#define MSTAT_TIMER      MK_BMSK_(MSTAT_TIMER_P   )
     // Timer enable
#define MSTAT_SEC_DAG    MK_BMSK_(MSTAT_SEC_DAG_P )
     // Secondary DAG registers enable


//**************************************************
//   SSTAT register
//**************************************************

// Bit Positions
#define SSTAT_PCEM_P   0   // Bit 0: PC stack empty
#define SSTAT_PCFL_P   1   // Bit 1: PC stack full
#define SSTAT_PCLV_P   2   // Bit 2: PC stack level
#define SSTAT_LSEM_P   4   // Bit 4: Loop stack empty
#define SSTAT_LSFL_P   5   // Bit 5: Loop stack full
#define SSTAT_SSEM_P   6   // Bit 6: Status stack empty
```

```
#define SSTAT_SSOV_P   7   // Bit 7: Stacks overflowed

// Bit Masks
#define SSTAT_PCEM  MK_BMSK_(SSTAT_PCEM_P)  // PC stack empty
#define SSTAT_PCFL  MK_BMSK_(SSTAT_PCFL_P)  // PC stack full
#define SSTAT_PCLV  MK_BMSK_(SSTAT_PCLV_P)  // PC stack level
#define SSTAT_LSEM  MK_BMSK_(SSTAT_LSEM_P)  // Loop stack empty
#define SSTAT_LSFL  MK_BMSK_(SSTAT_LSFL_P)  // Loop stack full
#define SSTAT_SSEM  MK_BMSK_(SSTAT_SSEM_P)  // Status stack empty
#define SSTAT_SSOV  MK_BMSK_(SSTAT_SSOV_P)  // Stacks overflowed


//**************************************************
//   ICNTL register
//**************************************************

// Bit Positions
#define ICNTL_INE_P      4   // Bit 4: Int nesting mode enable
#define ICNTL_GIE_P      5   // Bit 5: Global interrupt enable
#define ICNTL_BIASRND_P 7   // Bit 7: MAC biased rounding mode
#define ICNTL_PCSTKE_P 10   // Bit 10: PC stack interrupt enable
#define ICNTL_CCNTE_P  11   // Bit 11: Cycle counter enable

// Bit Masks
#define ICNTL_INE      MK_BMSK_(ICNTL_INE_P    )
     // Interrupt nesting mode enable
#define ICNTL_GIE      MK_BMSK_(ICNTL_GIE_P    )
     // Global interrupt enable
#define ICNTL_BIASRND  MK_BMSK_(ICNTL_BIASRND_P)
     // MAC biased rounding mode
#define ICNTL_PCSTKE   MK_BMSK_(ICNTL_PCSTKE_P )
     // PC stack interrupt enable
#define ICNTL_CCNTE       MK_BMSK_(ICNTL_CCNTE_P  )
     // Cycle counter enable
```

```
//***************************************************
//   IRPTL and IMASK registers
//***************************************************

// Bit Positions
#define INT_EMU_P      0   // Bit  0: Offset: 00: Emulator int
#define INT_PWDN_P     1   // Bit  1: Offset: 04: Powerdown intpt
#define INT_KRNL_P     2   // Bit  2: Offset: 08: Kernel interrupt
#define INT_STKI_P     3   // Bit  3: Offset: 0c: Stack interrupt
#define INT_INT4_P     4   // Bit  4: Offset: 10: Off-Core
#define INT_INT5_P     5   // Bit  5: Offset: 14: Off-Core
#define INT_INT6_P     6   // Bit  6: Offset: 18: Off-Core
#define INT_INT7_P     7   // Bit  7: Offset: 1c: Off-Core
#define INT_INT8_P     8   // Bit  8: Offset: 20: Off-Core
#define INT_INT9_P     9   // Bit  9: Offset: 24: Off-Core
#define INT_INT10_P  10   // Bit 10: Offset: 28: Off-Core
#define INT_INT11_P  11   // Bit 11: Offset: 2c: Off-Core
#define INT_INT12_P  12   // Bit 12: Offset: 30: Off-Core
#define INT_INT13_P  13   // Bit 13: Offset: 34: Off-Core
#define INT_INT14_P  14   // Bit 14: Offset: 38: Off-Core
#define INT_INT15_P  15   // Bit 15: Offset: 3c: Off-Core

// Bit Masks
#define INT_EMU           MK_BMSK_(INT_EMU_P  )
     // Offset: 00: Emulator interrupt
#define INT_PWDN          MK_BMSK_(INT_PWDN_P )
     // Offset: 04: Powerdown interrupt
#define INT_KRNL          MK_BMSK_(INT_KRNL_P )
     // Offset: 08: Kernel interrupt
#define INT_STKI         MK_BMSK_(INT_STKI_P )
     // Offset: 0c: Stack interrupt
#define INT_INT4         MK_BMSK_(INT_INT4_P )
```

```
     // Offset: 10: Off-Core
#define INT_INT5         MK_BMSK_(INT_INT5_P )
     // Offset: 14: Off-Core
#define INT_INT6         MK_BMSK_(INT_INT6_P )
     // Offset: 18: Off-Core
#define INT_INT7         MK_BMSK_(INT_INT7_P )
     // Offset: 1c: Off-Core
#define INT_INT8         MK_BMSK_(INT_INT8_P )
     // Offset: 20: Off-Core
#define INT_INT9         MK_BMSK_(INT_INT9_P )
     // Offset: 24: Off-Core
#define INT_INT10        MK_BMSK_(INT_INT10_P)
     // Offset: 28: Off-Core
#define INT_INT11        MK_BMSK_(INT_INT11_P)
     // Offset: 2c: Off-Core
#define INT_INT12        MK_BMSK_(INT_INT12_P)
     // Offset: 30: Off-Core
#define INT_INT13        MK_BMSK_(INT_INT13_P)
     // Offset: 34: Off-Core
#define INT_INT14        MK_BMSK_(INT_INT14_P)
     // Offset: 38: Off-Core
#define INT_INT15        MK_BMSK_(INT_INT15_P)
     // Offset: 3c: Off-Core


//***************************************************
//   CACTL register
//***************************************************

// Bit Positions
#define CACTL_CDE_P  5  // Bit 5 Cache conflict DM access enable
#define CACTL_CFZ_P  6  // Bit 6 Cache freeze
#define CACTL_CPE_P  7  // Bit 7 Cache conflict PM access enable
```

```
// Bit Masks
#define CACTL_CDE   MK_BMSK_(CACTL_CDE_P)
     // Cache conflicting DM access enable
#define CACTL_CFZ   MK_BMSK_(CACTL_CFZ_P)
     // Cache freeze
#define CACTL_CPE   MK_BMSK_(CACTL_CPE_P)
     // Cache conflicting PM access enable


//**************************************************
//   CCODE register
//**************************************************
#define cond_PF0  0x00  // if PF0 pin high, SWCOND true
#define cond_PF1  0x01  // if PF1 pin high, SWCOND true
#define cond_PF2  0x02  // if PF2 pin high, SWCOND true
#define cond_PF3  0x03  // if PF3 pin high, SWCOND true
#define cond_PF4  0x04  // if PF4 pin high, SWCOND true
#define cond_PF5  0x05  // if PF5 pin high, SWCOND true
#define cond_PF6  0x06  // if PF6 pin high, SWCOND true
#define cond_PF7  0x07  // if PF7 pin high, SWCOND true
#define cond_AS   0x08  // if AS, SWCOND true
#define cond_SV   0x09  // if SV, SWCOND true
#define cond_PF8  0x0A  // if PF8 pin high, SWCOND true
#define cond_PF9  0x0B  // if PF9 pin high, SWCOND true
#define cond_PF10 0x0C  // if PF10 pin high, SWCOND true
#define cond_PF11 0x0D  // if PF11 pin high, SWCOND true
#define cond_PF12 0x0E  // if PF12 pin high, SWCOND true
#define cond_PF13 0x0F  // if PF13 pin high, SWCOND true

//--------------------------------------------------------------
//           System Register address definitions
//--------------------------------------------------------------

#define B0  0x00  // DAG Base register 0 (for circ buf only)
```

```
#define B1  0x01  // DAG Base register 1 (for circ buff only)
#define B2  0x02  // DAG Base register 2 (for circ buff only)
#define B3  0x03  // DAG Base register 3 (for circ buff only)
#define B4  0x04  // DAG Base register 4 (for circ buff only)
#define B5  0x05  // DAG Base register 5 (for circ buff only)
#define B6  0x06  // DAG Base register 6 (for circ buff only)
#define B7  0x07  // DAG Base register 7 (for circ buff only)

#define CACTL 0x0F // Cache control register

#define DBGCTRL 0x60  // Emulation Debug Control Register
#define DBGSTAT 0x61  // Emulation Debug Status Register
#define CNT0    0x62  // Counter Register 0 (LSB) (read this 1st)
#define CNT1    0x63  // Counter Register 1
#define CNT2    0x64  // Cycle Counter Register 2
#define CNT3    0x65  // Cycle Counter Register 3 (MSB)

#endif
```

B ADSP-2191 DSP I/O REGISTERS

The DSP has general-purpose and dedicated registers in each of its functional blocks. The register reference information for each functional block includes bit definitions, initialization values, and (for I/O processor registers) memory-mapped address. Information on each type of register is available at the following locations:

- "Core Status Registers" on page A-8
- "Computational Unit Registers" on page A-11
- "Program Sequencer Registers" on page A-15
- "Data Address Generator Registers" on page A-20
- "I/O Processor Registers" on page B-2

When writing DSP programs, it is often necessary to set, clear, or test bits in the DSP's registers. While these bit operations can all be done by referring to the bit's location within a register or (for some operations) the register's address with a hexadecimal number, it is much easier to use symbols that correspond to the bit's or register's name. For convenience and consistency, Analog Devices provides a header file that provides these bit and register definitions. For more information, see the "Register and Bit #define File (def2191.h)" on page B-115 and "Register and Bit #define File (def2191.h)" on page B-115.

Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) a register's reserved bits.

I/O Processor Registers

The DSP's memory map includes the following groups of I/O processor registers:

- "Clock and System Control Registers" on page B-17
- "System Interrupt Controller Registers" on page B-21
- "DMA Controller Registers" on page B-27
- "SPORT Registers" on page B-35
- "Serial Peripheral Interface Registers" on page B-60
- "UART Registers" on page B-72
- "Timer Registers" on page B-88
- "Programmable Flag Registers" on page B-96
- "External Memory Interface Registers" on page B-99
- "Host Port Registers" on page B-107

The I/O processor registers are accessible as part of the DSP's memory map. Table B-1 on page B-3 lists the I/O processor's memory-mapped registers in address order and provides a cross reference to a description of each register. These registers occupy addresses `0x00` through `0xFF` of the memory map and control I/O operations, including:

- External port DMA
- Link port DMA
- Serial port DMA

I/O processor registers have a one-cycle effect latency (changes take effect on the second cycle after the change).

Because the I/O processor's registers are part of the DSP's I/O memory map, buses access these registers as locations in I/O memory. While these registers act as memory-mapped locations, they are separate from the DSP's internal memory.

To read or write I/O processor registers, programs must use the `IO( )` instruction. The following example code shows a value being transferred from the `AX0` register to the `DMACW_CP` register in I/O memory. The `IOPG` register is loaded to select the correct page in I/O memory. Because the page and address are necessary for accessing any I/O memory register, the I/O memory map in Table B-1 on page B-3 shows these as `IOPG`:Address.

```
IOPG = Memory_DMA_Controller_Page;    /* set the I/O mem page */
AX0 = WR_DMA_WORD_CONFIG;          /* Load AX0 with the cfg word */
IO(DMACW_CP) = ax0;          /* Load DMACW_CP with the cfg word */
```

The register names for I/O processor registers are not part of the DSP's assembly syntax. To ease access to these registers, programs should use the #include command to incorporate a file containing the registers' symbolic names and addresses. An example #include file appears in the "Register and Bit #define File (def2191.h)" on page B-115.

Table B-1. I/O Processor Registers Memory Map

DSP I/O Address (IOPG:Address)	**Host I/O Address[1] (on HA16–HAD1)**	**Register Name**	**Initialization After Reset**	**Page Cross Reference**
"Clock and System Control Registers" on page B-17				
0x00:0x200	0x0400	PLLCTL	0x0010	page B-17
0x00:0x201	0x0402	LOCKCNT	ni	page B-19
0x00:0x202	0x0404	SWRST	ni	page B-19
0x00:0x203	0x0406	NXTSCR	0x0000	page B-19
0x00:0x204	0x0408	SYSCR	0x0000	page B-20

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
"System Interrupt Controller Registers" on page B-21				
0x01:0x200	0x0C00	IPR0	Per interrupt request	page B-22
0x01:0x201	0x0C02	IPR1	Per interrupt request	page B-22
0x01:0x202	0x0C04	IPR2	Per interrupt request	page B-22
0x01:0x203	0x0C06	IPR3	Per interrupt request	page B-22
0x01:0x204	0x0C08	INTRD0	Per interrupt request	page B-25
0x01:0x205	0x0C0A	INTRD1	Per interrupt request	page B-25
0x01:0x206	0x0C0C	INTRD2	Per interrupt request	page B-25
0x01:0x207	0x0C0E	INTRD3	Per interrupt request	page B-25
0x01:0x208	0x0C0F	INTRD4	Per interrupt request	page B-25
0x01:0x209	0x0C12	INTRD5	Per interrupt request	page B-25
0x01:0x20A	0x0C14	INTRD6	Per interrupt request	page B-25
0x01:0x20B	0x0C16	INTRD7	Per interrupt request	page B-25
0x01:0x20C	0x0C18	INTRD8	Per interrupt request	page B-25
0x01:0x20D	0x0C1A	INTRD9	Per interrupt request	page B-25
0x01:0x20E	0x0C1C	INTRD10	Per interrupt request	page B-25
0x01:0x20F	0x0C1E	INTRD11	Per interrupt request	page B-25
"DMA Controller Registers" on page B-27				
0x02:0x100	0x1200	DMACW_PTR	0x0000	page B-29
0x02:0x101	0x1202	DMACW_CFG	0x0000	page B-29
0x02:0x102	0x1204	DMACW_SRP	0x0000	page B-31
0x02:0x103	0x1206	DMACW_SRA	0x0000	page B-31

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x02:0x104	0x1208	DMACW_CNT	0x0000	page B-31
0x02:0x105	0x120A	DMACW_CP	0x0000	page B-32
0x02:0x106	0x120C	DMACW_CPR	0x0000	page B-32
0x02:0x107	0x120E	DMACW_IRQ	0x0000	page B-32
0x02:0x180	0x1300	DMACR_PTR	0x0000	page B-33
0x02:0x181	0x1302	DMACR_CFG	0x0000	page B-33
0x02:0x182	0x1304	DMACR_SRP	0x0000	page B-33
0x02:0x183	0x1306	DMACR_SRA	0x0000	page B-34
0x02:0x184	0x1308	DMACR_CNT	0x0000	page B-34
0x02:0x185	0x130A	DMACR_CP	0x0000	page B-34
0x02:0x186	0x130C	DMACR_CPR	0x0000	page B-35
0x02:0x00187	0x130E	DMACR_IRQ	0x0000	page B-35
"SPORT Registers" on page B-35				
0x02:0x200	0x1400	SP0_TCR	0x0000	page B-38
0x02:0x201	0x1402	SP0_RCR	0x0000	page B-38
0x02:0x202	0x1404	SP0_TX	0x0000	page B-41
0x02:0x203	0x1406	SP0_RX	0x0000	page B-41
0x02:0x204	0x1408	SP0_TSCKDIV	0x0000	page B-42
0x02:0x205	0x140A	SP0_RSCKDIV	0x0000	page B-42
0x02:0x206	0x140C	SP0_TFSDIV	0x0000	page B-43
0x02:0x207	0x140E	SP0_RFSDIV	0x0000	page B-42
0x02:0x208	0x1410	SP0_STATR	0x0000	page B-43

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x02:0x209	0x1412	SP0_MTCS0	0x0000	page B-44
0x02:0x20A	0x1414	SP0_MTCS1	0x0000	page B-44
0x02:0x20B	0x1416	SP0_MTCS2	0x0000	page B-44
0x02:0x20C	0x1418	SP0_MTCS3	0x0000	page B-44
0x02:0x20D	0x141A	SP0_MTCS4	0x0000	page B-44
0x02:0x20E	0x141C	SP0_MTCS5	0x0000	page B-44
0x02:0x20F	0x141E	SP0_MTCS6	0x0000	page B-44
0x02:0x210	0x1420	SP0_MTCS7	0x0000	page B-44
0x02:0x211	0x1422	SP0_MRCS0	0x0000	page B-46
0x02:0x212	0x1424	SP0_MRCS1	0x0000	page B-46
0x02:0x213	0x1426	SP0_MRCS2	0x0000	page B-46
0x02:0x214	0x1428	SP0_MRCS3	0x0000	page B-46
0x02:0x215	0x142A	SP0_MRCS4	0x0000	page B-46
0x02:0x216	0x142C	SP0_MRCS5	0x0000	page B-46
0x02:0x217	0x142E	SP0_MRCS6	0x0000	page B-46
0x02:0x218	0x1430	SP0_MRCS7	0x0000	page B-46
0x02:0x219	0x1432	SP0_MCMC1	0x0000	page B-47
0x02:0x21A	0x1434	SP0_MCMC2	0x0000	page B-47
0x02:0x300	0x1600	SP0DR_PTR	0x0000	page B-48
0x02:0x301	0x1602	SP0DR_CFG	0x0000	page B-48
0x02:0x302	0x1604	SP0DR_SRP	0x0000	page B-50
0x02:0x303	0x1606	SP0DR_SRA	0x0000	page B-53

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x02:0x304	0x1608	SP0DR_CNT	0x0000	page B-53
0x02:0x305	0x160A	SP0DR_CP	0x0000	page B-53
0x02:0x306	0x160C	SP0DR_CPR	0x0000	page B-54
0x02:0x307	0x160E	SP0DR_IRQ	0x0000	page B-54
0x02:0x380	0x1700	SP0DT_PTR	0x0000	page B-55
0x02:0x381	0x1702	SP0DT_CFG	0x0000	page B-56
0x02:0x382	0x1704	SP0DT_SRP	0x0000	page B-57
0x02:0x383	0x1706	SP0DT_SRA	0x0000	page B-56
0x02:0x384	0x1708	SP0DT_CNT	0x0000	page B-57
0x02:0x385	0x170A	SP0DT_CP	0x0000	page B-58
0x02:0x386	0x170C	SP0DT_CPR	0x0000	page B-58
0x02:0x387	0x170E	SP0DT_IRQ	0x0000	page B-59
0x03:0x000	0x1800	SP1_TCR	0x0000	page B-38
0x03:0x001	0x1802	SP1_RCR	0x0000	page B-38
0x03:0x002	0x1804	SP1_TX	0x0000	page B-41
0x03:0x003	0x1806	SP1_RX	0x0000	page B-41
0x03:0x004	0x1808	SP1_TSCKDIV	0x0000	page B-42
0x03:0x005	0x180A	SP1_RSCKDIV	0x0000	page B-42
0x03:0x006	0x180C	SP1_TFSDIV	0x0000	page B-43
0x03:0x007	0x180E	SP1_RFSDIV	0x0000	page B-42
0x03:0x008	0x1810	SP1_STATR	0x0000	page B-43
0x03:0x009	0x1812	SP1_MTCS0	0x0000	page B-44

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x03:0x00A	0x1814	SP1_MTCS1	0x0000	page B-44
0x03:0x00B	0x1816	SP1_MTCS2	0x0000	page B-44
0x03:0x00C	0x1818	SP1_MTCS3	0x0000	page B-44
0x03:0x00D	0x181A	SP1_MTCS4	0x0000	page B-44
0x03:0x00E	0x181C	SP1_MTCS5	0x0000	page B-44
0x03:0x00F	0x181E	SP1_MTCS6	0x0000	page B-44
0x03:0x010	0x1820	SP1_MTCS7	0x0000	page B-44
0x03:0x011	0x1822	SP1_MRCS0	0x0000	page B-46
0x03:0x012	0x1824	SP1_MRCS1	0x0000	page B-46
0x03:0x013	0x1826	SP1_MRCS2	0x0000	page B-46
0x03:0x014	0x1828	SP1_MRCS3	0x0000	page B-46
0x03:0x015	0x182A	SP1_MRCS4	0x0000	page B-46
0x03:0x016	0x182C	SP1_MRCS5	0x0000	page B-46
0x03:0x017	0x182E	SP1_MRCS6	0x0000	page B-46
0x03:0x018	0x1830	SP1_MRCS7	0x0000	page B-46
0x03:0x019	0x1832	SP1_MCMC1	0x0000	page B-47
0x03:0x01A	0x1834	SP1_MCMC2	0x0000	page B-47
0x03:0x100	0x1A00	SP1DR_PTR	0x0000	page B-48
0x03:0x101	0x1A02	SP1DR_CFG	0x0000	page B-48
0x03:0x102	0x1A04	SP1DR_SRP	0x0000	page B-50
0x03:0x103	0x1A06	SP1DR_SRA	0x0000	page B-53
0x03:0x104	0x1A08	SP1DR_CNT	0x0000	page B-53

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x03:0x105	0x1A0A	SP1DR_CP	0x0000	page B-53
0x03:0x106	0x1A0C	SP1DR_CPR	0x0000	page B-54
0x03:0x107	0x1A0E	SP1DR_IRQ	0x0000	page B-54
0x03:0x180	0x1A10	SP1DT_PTR	0x0000	page B-55
0x03:0x181	0x1A12	SP1DT_CFG	0x0000	page B-56
0x03:0x182	0x1A14	SP1DT_SRP	0x0000	page B-57
0x03:0x183	0x1A16	SP1DT_SRA	0x0000	page B-56
0x03:0x184	0x1A18	SP1DT_CNT	0x0000	page B-57
0x03:0x185	0x1A1A	SP1DT_CP	0x0000	page B-58
0x03:0x186	0x1A1C	SP1DT_CPR	0x0000	page B-58
0x03:0x187	0x1A1E	SP1DT_IRQ	0x0000	page B-59
0x03:0x200	0x1C00	SP2_TCR	0x0000	page B-38
0x03:0x201	0x1C02	SP2_RCR	0x0000	page B-38
0x03:0x202	0x1C04	SP2_TX	0x0000	page B-41
0x03:0x203	0x1C06	SP2_RX	0x0000	page B-41
0x03:0x204	0x1C08	SP2_TSCKDIV	0x0000	page B-42
0x03:0x205	0x1C0A	SP2_RSCKDIV	0x0000	page B-42
0x03:0x206	0x1C0C	SP2_TFSDIV	0x0000	page B-43
0x03:0x207	0x1C0E	SP2_RFSDIV	0x0000	page B-42
0x03:0x208	0x1C10	SP2_STATR	0x0000	page B-43
0x03:0x209	0x1C12	SP2_MTCS0	0x0000	page B-44
0x03:0x20A	0x1C14	SP2_MTCS1	0x0000	page B-44

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x03:0x20B	0x1C16	SP2_MTCS2	0x0000	page B-44
0x03:0x20C	0x1C18	SP2_MTCS3	0x0000	page B-44
0x03:0x20D	0x1C1A	SP2_MTCS4	0x0000	page B-44
0x03:0x20E	0x1C1C	SP2_MTCS5	0x0000	page B-44
0x03:0x20F	0x1C1E	SP2_MTCS6	0x0000	page B-44
0x03:0x210	0x1C20	SP2_MTCS7	0x0000	page B-44
0x03:0x211	0x1C22	SP2_MRCS0	0x0000	page B-46
0x03:0x212	0x1C24	SP2_MRCS1	0x0000	page B-46
0x03:0x213	0x1C26	SP2_MRCS2	0x0000	page B-46
0x03:0x214	0x1C28	SP2_MRCS3	0x0000	page B-46
0x03:0x215	0x1C2A	SP2_MRCS4	0x0000	page B-46
0x03:0x216	0x1C2C	SP2_MRCS5	0x0000	page B-46
0x03:0x217	0x1C2E	SP2_MRCS6	0x0000	page B-46
0x03:0x218	0x1C30	SP2_MRCS7	0x0000	page B-46
0x03:0x219	0x1C32	SP2_MCMC1	0x0000	page B-47
0x03:0x21A	0x1C34	SP2_MCMC2	0x0000	page B-47
0x03:0x300	0x1E00	SP2DR_PTR	0x0000	page B-48
0x03:0x301	0x1E02	SP2DR_CFG	0x0000	page B-48
0x03:0x302	0x1E04	SP2DR_SRP	0x0000	page B-50
0x03:0x303	0x1E06	SP2DR_SRA	0x0000	page B-53
0x03:0x304	0x1E08	SP2DR_CNT	0x0000	page B-53
0x03:0x305	0x1E0A	SP2DR_CP	0x0000	page B-53

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x03:0x306	0x1E0C	SP2DR_CPR	0x0000	page B-54
0x03:0x307	0x1E0E	SP2DR_IRQ	0x0000	page B-54
0x03:0x380	0x1E10	SP2DT_PTR	0x0000	page B-55
0x03:0x381	0x1E12	SP2DT_CFG	0x0000	page B-56
0x03:0x382	0x1E14	SP2DT_SRP	0x0000	page B-57
0x03:0x383	0x1E16	SP2DT_SRA	0x0000	page B-56
0x03:0x384	0x1E18	SP2DT_CNT	0x0000	page B-57
0x03:0x385	0x1E1A	SP2DT_CP	0x0000	page B-58
0x03:0x386	0x1E1C	SP2DT_CPR	0x0000	page B-58
0x03:0x387	0x1E1E	SP2DT_IRQ	0x0000	page B-59
"Serial Peripheral Interface Registers" on page B-60				
0x04:0x000	0x2000	SPICTL0	0x0400	page B-61
0x04:0x001	0x2002	SPIFLG0	0xFF00	page B-63
0x04:0x002	0x20024	SPIST0	0x01	page B-65
0x04:0x003	0x2006	TDBR0	0x0000	page B-65
0x04:0x004	0x2008	RDBR0	0x0000	page B-67
0x04:0x005	0x200A	SPIBAUD0	0x0000	page B-68
0x04:0x006	0x200C	RDBRS0	0x0000	page B-67
0x04:0x100	0x2200	SPI0D_PTR	0x0000	page B-68
0x04:0x101	0x2202	SPI0D_CFG	0x0000	page B-68
0x04:0x102	0x2204	SPI0D_SRP	0x0000	page B-70
0x04:0x103	0x2206	SPI0D_SRA	0x0000	page B-70

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x04:0x104	0x2208	SPI0D_CNT	0x0000	page B-70
0x04:0x105	0x220A	SPI0D_CP	0x0000	page B-71
0x04:0x106	0x220C	SPI0D_CPR	0x0000	page B-71
0x04:0x107	0x220E	SPI0D_IRQ	0x0000	page B-71
0x04:0x200	0x2400	SPICTL1	0x0400	page B-61
0x04:0x201	0x2402	SPIFLG1	0xFF00	page B-63
0x04:0x202	0x2404	SPIST1	0x01	page B-65
0x04:0x203	0x2406	TDBR1	0x0000	page B-65
0x04:0x204	0x2408	RDBR1	0x0000	page B-67
0x04:0x205	0x240A	SPIBAUD1	0x0000	page B-68
0x04:0x206	0x240C	RDBRS1	0x0000	page B-67
0x04:0x300	0x2600	SPI1D_PTR	0x0000	page B-68
0x04:0x301	0x2602	SPI1D_CFG	0x0000	page B-68
0x04:0x302	0x2604	SPI1D_SRP	0x0000	page B-70
0x04:0x303	0x2606	SPI1D_SRA	0x0000	page B-70
0x04:0x304	0x2608	SPI1D_CNT	0x0000	page B-70
0x04:0x305	0x260A	SPI1D_CP	0x0000	page B-71
0x04:0x306	0x260C	SPI1D_CPR	0x0000	page B-71
0x04:0x307	0x260E	SPI1D_IRQ	0x0000	page B-71
"UART Registers" on page B-72				
0x05:0x000	0x2800	THR	0x01	page B-74
0x05:0x000	0x2800	RBR	0x0000	page B-74

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x05:0x000	0x2800	DLL	0x0000	page B-76
0x05:0x001	0x2802	IER	0x0000	page B-75
0x05:0x001	0x2802	DLH	0x0000	page B-76
0x05:0x002	0x2804	IIR	0x01	page B-77
0x05:0x003	0x2806	LCR	0x0000	page B-77
0x05:0x004	0x2808	MCR	See register	page B-77
0x05:0x005	0x280A	LSR	0x0060	page B-78
0x05:0x006	0x280C	MSR	0x0000	page B-78
0x05:0x007	0x280E	SCR	0x0000	page B-80
0x05:0x100	0x2A00	UARDR_PTR	0x0000	page B-81
0x05:0x101	0x2A02	UARDR_CFG	0x0000	page B-81
0x05:0x102	0x2A04	UARDR_SRP	0x0000	page B-83
0x05:0x103	0x2A06	UARDR_SRA	0x0000	page B-83
0x05:0x104	0x2A08	UARDR_CNT	0x0000	page B-84
0x05:0x105	0x2A0A	UARDR_CP	0x0000	page B-84
0x05:0x106	0x2A0C	UARDR_CPR	0x0000	page B-84
0x05:0x107	0x2A0E	UARDR_IRQ	0x0000	page B-84
0x05:0x180	0x2B00	UARDT_PTR	0x0000	page B-86
0x05:0x181	0x2B02	UARDT_CFG	0x0000	page B-86
0x05:0x182	0x2B04	UARDT_SRP	0x0000	page B-86
0x05:0x183	0x2B06	UARDT_SRA	0x0000	page B-87
0x05:0x184	0x2B08	UARDT_CNT	0x0000	page B-87

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x05:0x185	0x2B0A	UARDT_CP	0x0000	page B-87
0x05:0x186	0x2B0C	UARDT_CPR	0x0000	page B-87
0x05:0x187	0x2B0E	UARDT_IRQ	0x0000	page B-88
"Timer Registers" on page B-88				
0x05:0x200	0x2C00	T_GSR0	0x0000	page B-90
0x05:0x201	0x2C02	T_CFGR0	0x0000	page B-90
0x05:0x202	0x2C04	T_CNTL0	0x0000	page B-92
0x05:0x203	0x2C06	T_CNTH0	0x0000	page B-92
0x05:0x204	0x2C08	T_PRDL0	0x0000	page B-93
0x05:0x205	0x2C0A	T_PRDH0	0x0000	page B-93
0x05:0x206	0x2C0C	T_WLR0	0x0000	page B-95
0x05:0x207	0x2C0E	T_WHR0	0x0000	page B-95
0x05:0x208	0x2C10	T_GSR1	0x0000	page B-90
0x05:0x209	0x2C12	T_CFGR1	0x0000	page B-90
0x05:0x20A	0x2C14	T_CNTL1	0x0000	page B-92
0x05:0x20B	0x2C16	T_CNTH1	0x0000	page B-92
0x05:0x20C	0x2C18	T_PRDL1	0x0000	page B-93
0x05:0x20D	0x2C1A	T_PRDH1	0x0000	page B-93
0x05:0x20E	0x2C1C	T_WLR1	0x0000	page B-95
0x05:0x20F	0x2C1E	T_WHR1	0x0000	page B-95
0x05:0x210	0x2C20	T_GSR2	0x0000	page B-90
0x05:0x211	0x2C22	T_CFGR2	0x0000	page B-90

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x05:0x212	0x2C24	T_CNTL2	0x0000	page B-92
0x05:0x213	0x2C26	T_CNTH2	0x0000	page B-92
0x05:0x214	0x2C28	T_PRDL2	0x0000	page B-93
0x05:0x215	0x2C2A	T_PRDH2	0x0000	page B-93
0x05:0x216	0x2C2C	T_WLR2	0x0000	page B-95
0x05:0x217	0x2C2E	T_WHR2	0x0000	page B-95
"Programmable Flag Registers" on page B-96				
0x06:0x000	0x3000	DIR	0x0000	page B-97
0x06:0x002	0x3004	FLAGC	Input	page B-97
0x06:0x003	0x3006	FLAGS	Input	page B-97
0x06:0x004	0x3008	MASKAC	0x0000	page B-97
0x06:0x005	0x300A	MASKAS	0x0000	page B-97
0x06:0x006	0x300C	MASKBC	0x0000	page B-97
0x06:0x007	0x300E	MASKBS	0x0000	page B-97
0x06:0x008	0x3010	FSPRC	0x0000	page B-98
0x06:0x009	0x3012	FSPRS	0x0000	page B-98
0x06:0x00A	0x3014	FSSR	0x0000	page B-99
0x06:0x00C	0x3018	FSBERC	0x0000	page B-99
0x06:0x00D	0x301A	FSBERS	0x0000	page B-99
"External Memory Interface Registers" on page B-99				
0x00:0x080	0x0100	E_STAT	0x0300	page B-100
0x06:0x201	0x3402	EMICTL	0x0070	page B-100

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x06:0x202	0x3404	BMSCTL	0x0DFF	page B-102
0x06:0x203	0x3406	MS0CTL	0x0DFF	page B-104
0x06:0x204	0x3408	MS1CTL	0x0DFF	page B-104
0x06:0x205	0x340A	MS2CTL	0x0DFF	page B-104
0x06:0x206	0x340C	MS3CTL	0x0000	page B-104
0x06:0x207	0x340E	IOMSCTL	0x0000	page B-105
0x06:0x208	0x3410	EMISTAT	0x0000	page B-105
0x06:0x209	0x3412	MSPG10	0x0000	page B-105
0x06:0x20A	0x3414	MSPG32	0x0000	page B-105
"Host Port Registers" on page B-107				
0x07:0x001	0x3802	HPCR	0x0000	page B-109
0x07:0x002	0x3804	HPPR	0x0000	page B-109
0x07:0x003	0x3806	HPDER	0x0000	page B-109
0x07:0x0FC	0x39F8	HPSMPHA	0x0000	page B-112
0x07:0x0FD	0x39FA	HPSMPHB	0x0000	page B-112
0x07:0x100	0x3A00	HOSTD_PTR	0x0000	page B-112
0x07:0x101	0x3A02	HOSTD_CFG	0x0000	page B-112
0x07:0x102	0x3A04	HOSTD_SRP	0x0000	page B-113
0x07:0x103	0x3A06	HOSTD_SRA	0x0000	page B-113
0x07:0x104	0x3A08	HOSTD_CNT	0x0000	page B-113
0x07:0x105	0x3A0A	HOSTD_CP	0x0000	page B-113

Table B-1. I/O Processor Registers Memory Map (Cont'd)

DSP I/O Address (IOPG:Address)	Host I/O Address[1] (on HA16–HAD1)	Register Name	Initialization After Reset	Page Cross Reference
0x07:0x106	0x3A0C	HOSTD_CPR	0x0000	page B-115
0x07:0x107	0x3A0E	HOSTD_IRQ	0x0000	page B-115

1 HAD0 usage depends on Host port configuration.

Clock and System Control Registers

Clock and system control group of I/P registers include:

- "PLL Control (PLLCTL) Register" on page B-17
- "PLL Lock Counter (LOCKCNT) Register" on page B-19
- "Software Reset (SWRST) Register" on page B-19
- "Next System Configuration (NXTSCR) Register" on page B-19
- "System Configuration (SYSCR) Register" on page B-20

PLL Control (PLLCTL) Register

The PLL Control (`PLLCTL`) register lets systems select and change the DSP's core clock (CCLK) frequency and select power-down modes. The PLL multiplies the clock frequency of the input clock with a programmable ratio. The PLL Control register address is `0x00:0x200`.

At reset, the PLL starts in BYPASS mode, running the CCLK clock directly from CLKIN. The reset must be active at least four clock cycle to allow full initialization of the synchronizer chain. After the PLL is locked, software can switch to a clock multiplier mode.

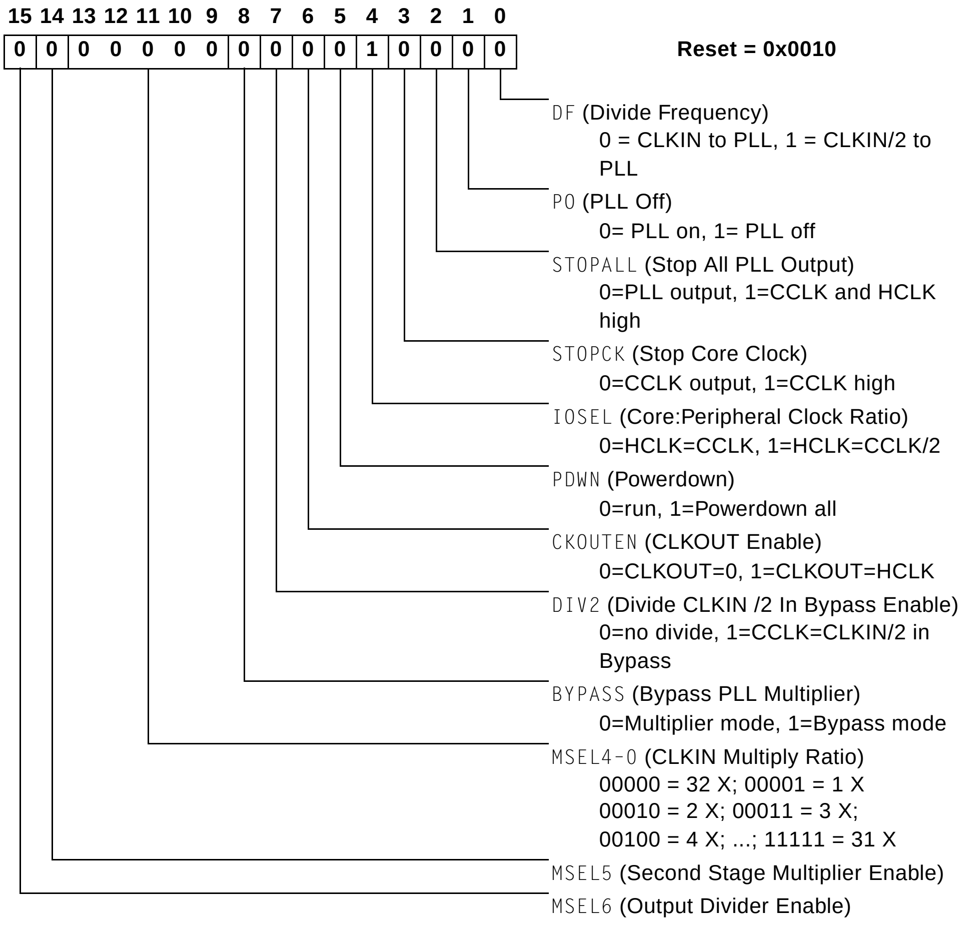

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
Reset = 0x0010
DF (Divide Frequency)
0 = CLKIN to PLL, 1 = CLKIN/2 to PLL
PO (PLL Off)
0= PLL on, 1= PLL off
STOPALL (Stop All PLL Output)
0=PLL output, 1=CCLK and HCLK high
STOPCK (Stop Core Clock)
0=CCLK output, 1=CCLK high
IOSEL (Core:Peripheral Clock Ratio)
0=HCLK=CCLK, 1=HCLK=CCLK/2
PDWN (Powerdown)
0=run, 1=Powerdown all
CKOUTEN (CLKOUT Enable)
0=CLKOUT=0, 1=CLKOUT=HCLK
DIV2 (Divide CLKIN /2 In Bypass Enable)
0=no divide, 1=CCLK=CLKIN/2 in Bypass
BYPASS (Bypass PLL Multiplier)
0=Multiplier mode, 1=Bypass mode
MSEL4-0 (CLKIN Multiply Ratio)
00000 = 32 X; 00001 = 1 X
00010 = 2 X; 00011 = 3 X;
00100 = 4 X; ...; 11111 = 31 X
MSEL5 (Second Stage Multiplier Enable)
MSEL6 (Output Divider Enable)

Figure B-1. PLL Control (PLLCTL) Register Bits

Figure B-1 on page B-18 provides bit descriptions for the register.

Refer to "Using the PLL Control (PLLCTL) Register" on page 14-32 for more information.

PLL Lock Counter (LOCKCNT) Register

The Lock Counter (`LOCKCNT`) is a 10-bit register. The register address is `0x00:0x201`.

The process of changing the multiplication factor of the PLL takes a certain number of cycles, and therefore a lock counter is required to calculate when the PLL is locked to the new ratio. The value of the `LOCKCNT` depends on the frequency (the higher the capacitor must be charged, the longer is the time required to lock). At power-up, the Lock Counter has to be initialized. Therefore, during reset, the lock signal is forced and set active indicating that the PLL is locked even though this may not be true. The reset pulse must be long enough to guarantee that the PLL is effectively locked at the end of the reset sequence or the software must wait before switching the clock source to the PLL output.

Software Reset (SWRST) Register

The Software Reset (`SWRST`) register is write-only. Its address is `0x00:0x202`. The DSP core software reset is initiated by the DSP core by writing `0x07` into the software reset (`SWR`) bits 2–0 in the Software Rest (`SWRST`) register. Thus, value "7" triggers Software reset, values 0–6 specify no software reset. Bits 3 through 15 are set to 0.

If bits 2–0 are set, the reset affects only the state of the core and most of the peripherals. It does not make use of the hardware reset timer and logic and does not reset the PLL and PLL control register.

A software reset of the peripheral will cause loss of state and immediate termination of DMA processing.

Next System Configuration (NXTSCR) Register

This register address is `0x00:0x203`.

During normal chip operation, reset parameters may be written by the DSP core into the I/O-mapped Next System Configuration (NXTSCR) register. The state is latched/registered into this register and held there until a software reset. A subsequent software reset updates the state of the System Configuration register with the contents of NXTSCR , and will then be allowed to propagate through to the register output drivers and distributed to DSP core and peripherals. For bit descriptions, see Figure B-2 on page B-21.

ⓘ The reset state is initialized during hardware reset from boot mode pins and may also be altered by the boot kernel. These bits are read-write during normal chip operation.

System Configuration (SYSCR) Register

The System Configuration (SYSCR) register is a read-write register. Its address is 0x00:0x204.

The SYSCR register has the same bit layout as the Next System Configuration (NXTSCR) register shown in Figure B-2. A software reset will update the state of the SYSCR register with the contents of the NXTSCR register, which will then be allowed to propagate through to the register output drivers and be distributed to the DSP core and peripherals. When writing directly to the SYSCR register, the RMODE pin can be used to control the base address of the interrupt vector table whether 0x00 0000 or 0x01 0000.

The OPMODE pin is a dedicated mode control pin, it is typically used to select between one serial port or two SPI ports. During boot, the OPMODE pin serves as the BMODE2 pin.

The BMODE1-0 pins are the dedicated mode control pins. The pins and the corresponding bits in the System Configuration register configure the boot mode that is employed following hardware reset or software reset.

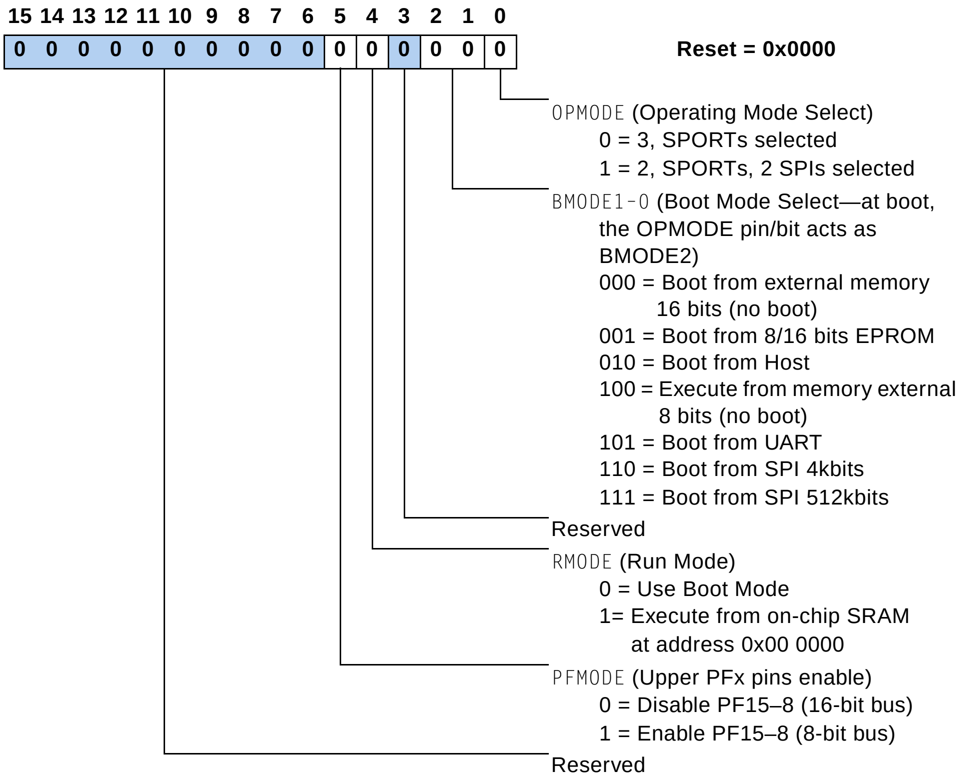

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
OPMODE (Operating Mode Select)
0 = 3, SPORTs selected
1 = 2, SPORTs, 2 SPIs selected
BMODE1-0 (Boot Mode Select—at boot, the OPMODE pin/bit acts as BMODE2)
000 = Boot from external memory 16 bits (no boot)
001 = Boot from 8/16 bits EPROM
010 = Boot from Host
100 = Execute from memory external 8 bits (no boot)
101 = Boot from UART
110 = Boot from SPI 4kbits
111 = Boot from SPI 512kbits
Reserved
RMODE (Run Mode)
0 = Use Boot Mode
1= Execute from on-chip SRAM at address 0x00 0000
PFMODE (Upper PFx pins enable)
0 = Disable PF15–8 (16-bit bus)
1 = Enable PF15–8 (8-bit bus)
Reserved

Figure B-2. Next System Configuration (NXTSCR) Register Bits

System Interrupt Controller Registers

The Interrupt Controller module (IRQ) combines and prioritizes interrupt sources from the various peripherals. The peripheral interrupt controller module is a generic module used to combine and prioritize 16 interrupt sources into 12 interrupt requests. The module includes four configuration registers that individually define the priority of interrupt sources. The module also includes twelve interrupt read registers, each register being associated with one of the interrupt request. The Interrupt Controller registers are:

- "Interrupt Priority (IPRx) Registers" on page B-22
- "Interrupt Source (INTRDx) Registers" on page B-25

Interrupt Priority (IPRx) Registers

There are four interrupt priority registers (which are part of the peripheral interrupt controller module). The registers have the following addresses:

```
IPR0 0x01:0x200
IPR1 0x01:0x201
IPR2 0x01:0x202
IPR3 0x01:0x203
```

As shown in Figure B-3, Figure B-4, Figure B-5, and Figure B-6, the `IPRx` registers individually define the priority of interrupt sources. Each `IPRx` register is a 16-bit peripheral memory mapped register which is divided into four-bit priority fields, each associated to one interrupt source. The priority level is defined from 0 to 11, 0 being the highest priority and 11 being the lowest. The interrupt request of priority 0 is connected to interrupt 4 of the DSP core and the interrupt request of priority 11 is connected to interrupt 15 of the DSP core.

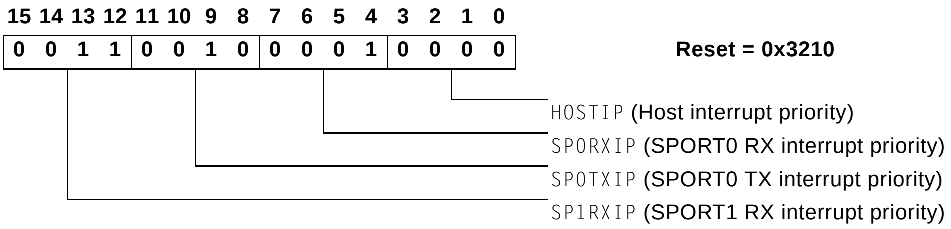

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0
Reset = 0x3210
HOSTIP (Host interrupt priority)
SP0RXIP (SPORT0 RX interrupt priority)
SP0TXIP (SPORT0 TX interrupt priority)
SP1RXIP (SPORT1 RX interrupt priority)

Figure B-3. Interrupt Priority Register 0 (IPR0) Bits

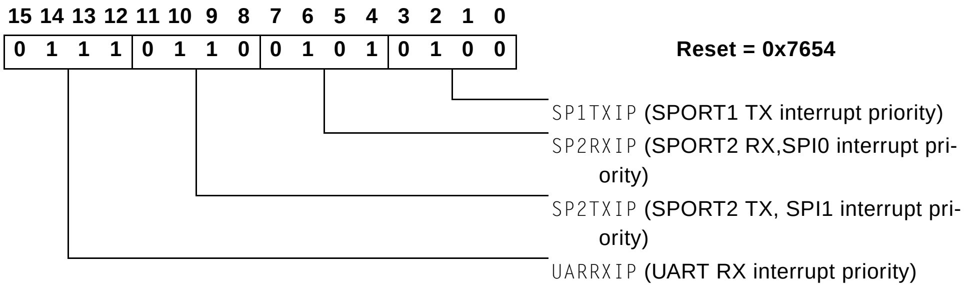

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 0
Reset = 0x7654
SP1TXIP (SPORT1 TX interrupt priority)
SP2RXIP (SPORT2 RX,SPI0 interrupt priority)
SP2TXIP (SPORT2 TX, SPI1 interrupt priority)
UARRXIP (UART RX interrupt priority)

Figure B-4. Interrupt Priority Register 1 (IPR1) Bits

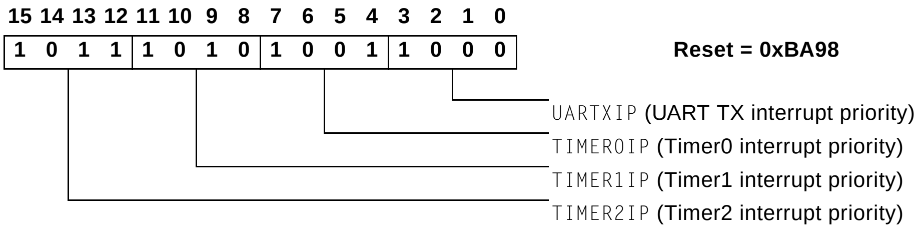

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 1 0 1 0 1 0 0 1 1 0 0 0
Reset = 0xBA98
UARTXIP (UART TX interrupt priority)
TIMER0IP (Timer0 interrupt priority)
TIMER1IP (Timer1 interrupt priority)
TIMER2IP (Timer2 interrupt priority)

Figure B-5. Interrupt Priority Register 2 (IPR2) Bits

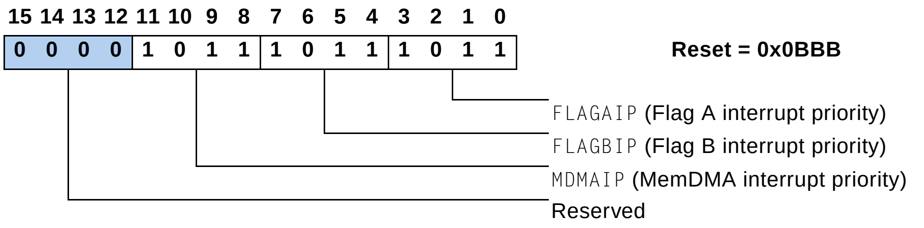

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 1 1 0 1 1 1 0 1 1
Reset = 0x0BBB
FLAGAIP (Flag A interrupt priority)
FLAGBIP (Flag B interrupt priority)
MDMAIP (MemDMA interrupt priority)
Reserved

Figure B-6. Interrupt Priority Register 3 (IPR3) Bits

An interrupt source is configured and applied to one of the interrupt requests line as described in Table B-2 on page B-24.

Table B-2. IPRx Register Bits

Value in Priority Field (PERIxP)	Interrupt Request	DSP interrupt (IMASK/IRPTL)
0	DSPIRQ0	IRPTL4
1	DSPIRQ1	IRPTL5
2	DSPIRQ2	IRPTL6
3	DSPIRQ3	IRPTL7
4	DSPIRQ4	IRPTL8
5	DSPIRQ5	IRPTL9
6	DSPIRQ6	IRPTL10
7	DSPIRQ7	IRPTL11
8	DSPIRQ8	IRPTL12
9	DSPIRQ9	IRPTL13
10	DSPIRQ10	IRPTL14
11	DSPIRQ11	IRPTL15
15–12	Reserved	Reserved

If the value of priority level field is 12 (0xC) or higher, then the interrupt source is masked and the interrupt does not propagate to any of the interrupt requests.

Some boot scenarios may alter the default values of these registers. It is good programming style to set all unused priority level fields to `0x0B` (priority 12).

According to the values of the priority field in the `IPR3-0` registers at reset, the initial configuration of the interrupt sources after reset are described in Table B-4 on page B-26.

Interrupt Source (INTRDx) Registers

There are twelve interrupt read registers; each register is associated with one of the interrupt request. The register addresses are as follows:

```
INTRD0  0x01:0x204       INTRD1  0x01:0x205
INTRD2  0x01:0x206       INTRD3  0x01:0x207
INTRD4  0x01:0x208       INTRD5  0x01:0x209
INTRD6  0x01:0x20A       INTRD7  0x01:0x20B
INTRD8  0x01:0x20C       INTRD9  0x01:0x20D
INTRD10 0x01:0x20E       INTRD11 0x01:0x20F
```

Each `INTRDx` register is a 16-bit peripheral memory mapped register. (which is part of the peripheral interrupt controller module). The value of each register is associated with one of the interrupt request (for example, `INTRD0` is associated with `DSPIRQ0`).

Each bit indicates the status of the 16 interrupt sources for the given request (0 if masked or inactive, 1 if unmasked and active). Since several interrupt sources can be combined on one interrupt request, reading the interrupt source register allows the DSP to determine the active interrupt source(s).

Table B-3. INTDRx Register Bit Descriptions

Bit	Interrupt Source
0	Slave DMA / Host
1	SPORT 0 RX
2	SPORT 0 TX
3	SPORT 1 RX
4	SPORT 1 TX
5	SPORT 2 RX / SPI 0
6	SPORT 2 TX / SPI 1

Table B-3. INTDRx Register Bit Descriptions (Cont'd)

Bit	Interrupt Source
7	UART RX
8	UART TX
9	Timer 0
10	Timer 1
11	Timer 2
12	Flag A
13	Flag B
14	MemDMA
15	reserved

At reset, interrupt sources have been assigned to a given priority level (interrupt request). For more information, see "Interrupt Priority (IPRx) Registers" on page B-22.

Table B-4 on page B-26 shows the ID and priority at reset of each of the peripheral interrupts at reset. To assign the peripheral interrupts a different priority, applications write the new priority to their corresponding control bits (determined by their ID) in the Interrupt Priority (IPRx) register. For more information, see "Interrupt Priority (IPRx) Registers" on page B-22.

Table B-4. Peripheral Interrupts and Priority at Reset

Interrupt	ID	Reset Priority
Slave DMA/Host Port Interface	0	0
SPORT0 Receive	1	1
SPORT0 Transmit	2	2
SPORT1 Receive	3	3

Table B-4. Peripheral Interrupts and Priority at Reset (Cont'd)

Interrupt	ID	Reset Priority
SPORT1 Transmit	4	4
SPORT2 Receive/SPI0	5	5
SPORT2 Transmit/SPI1	6	6
UART Receive	7	7
UART Transmit	8	8
Timer0	9	9
Timer1	10	10
Timer2	11	11
Programmable Flag 0 (any PFx)	12	11
Programmable Flag 1 (any PFx)	13	11
Memory DMA port	14	11

DMA Controller Registers

The Memory DMA peripheral (MemDMA) is responsible for moving data and instructions between internal and off-chip memory. This is performed over the DMA bus.

The MemDMA is made up of two DMA channels: a dedicated "read" channel and a dedicated "write" channel. Data is first read and stored in an internal 4-word FIFO buffer. Once full, the FIFO's contents are written to their destination. This process is repeated for the desired number of the transfers. Upon completion an interrupt is generated to the processor. It should be noted that this scheme is free from overrun errors because of the interlocking nature of a read followed by a write.

The section includes the following topics:

- "MemDMA Channel Read Count (DMACR_CNT) Register" on page B-34
- "MemDMA Channel Read Chain Pointer (DMACR_CP) Register" on page B-34
- "MemDMA Channel Read Chain Pointer Ready (DMACR_CPR) Register" on page B-35
- "MemDMA Channel Read Interrupt (DMACR_IRQ) Register" on page B-35

MemDMA Channel Write Pointer (DMACW_PTR) Register

The register address is `0x02:0x100`. This read-only register holds the pointer to the current descriptor block for the DMA write operation. The reset value is `0x0000`.

MemDMA Channel Write Configuration (DMACW_CFG) Register

The register address is `0x02:0x101`. The `DMACW_CFG` register should only be written when starting DMA operation. Figure B-7 describes this register bits. Additional information on bits (not covered in the Figure B-7) include:

- **Direction:** Bit 1 (`TRAN`) must be set (=1) for the write operation
- **DMA Buffer Clear:** Bit 7 (`FLSH`) should be set (=1) only if a DMA transfer has completed unsuccessfully.
- **Descriptor Ownership:** Bit 15 (`DOWN`) is checked before a full descriptor block download is begun to determine if the descriptor block is configured and ready for use.

For more information on using DMA processes, see "I/O Processor" on page 6-1.

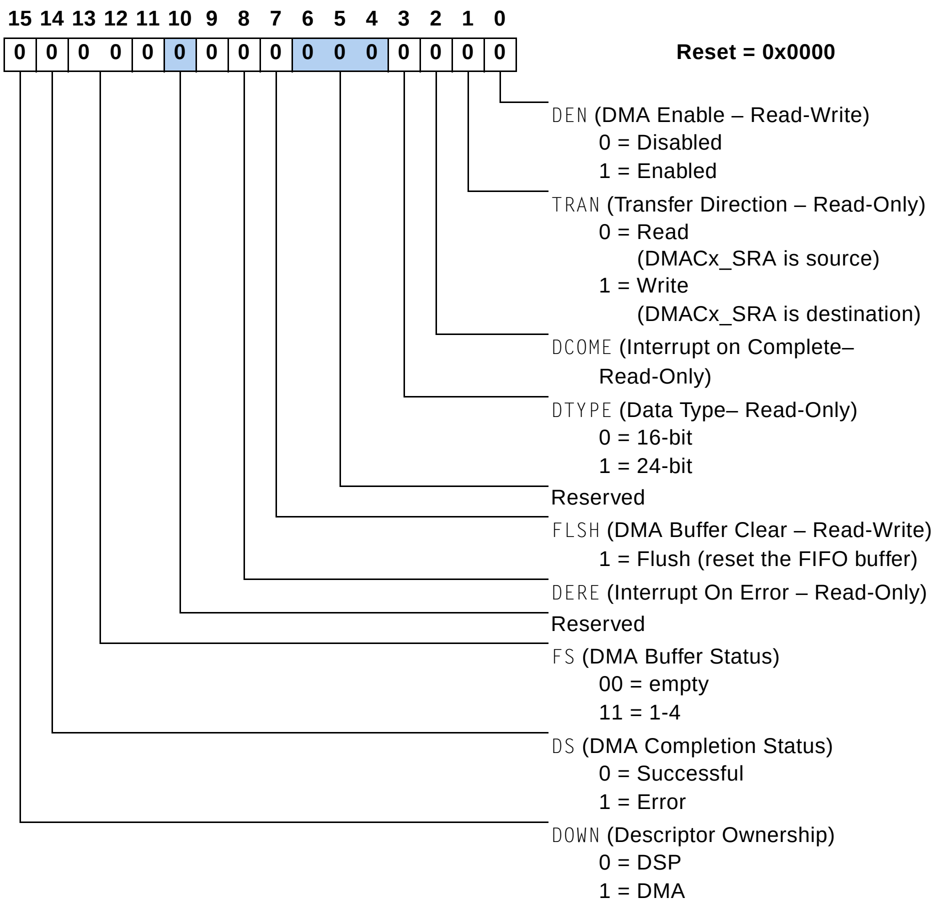

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DEN (DMA Enable – Read-Write)
0 = Disabled
1 = Enabled
TRAN (Transfer Direction – Read-Only)
0 = Read
(DMACx_SRA is source)
1 = Write
(DMACx_SRA is destination)
DCOME (Interrupt on Complete–
Read-Only)
DTYPE (Data Type– Read-Only)
0 = 16-bit
1 = 24-bit
Reserved
FLSH (DMA Buffer Clear – Read-Write)
1 = Flush (reset the FIFO buffer)
DERE (Interrupt On Error – Read-Only)
Reserved
FS (DMA Buffer Status)
00 = empty
11 = 1-4
DS (DMA Completion Status)
0 = Successful
1 = Error
DOWN (Descriptor Ownership)
0 = DSP
1 = DMA

Figure B-7. DMA, MemDMA Channel Write Configuration (DMACW_CFG) Register Bits

MemDMA Channel Write Start Page (DMACW_SRP) Register

The register address is 0x02:0x102. The 16-bit DMA Write Start Page (DMACW_SPR) register holds a running pointer to the DMA address that is being accessed and the memory space being used for a Write transfer. The reset value is 0x0000.

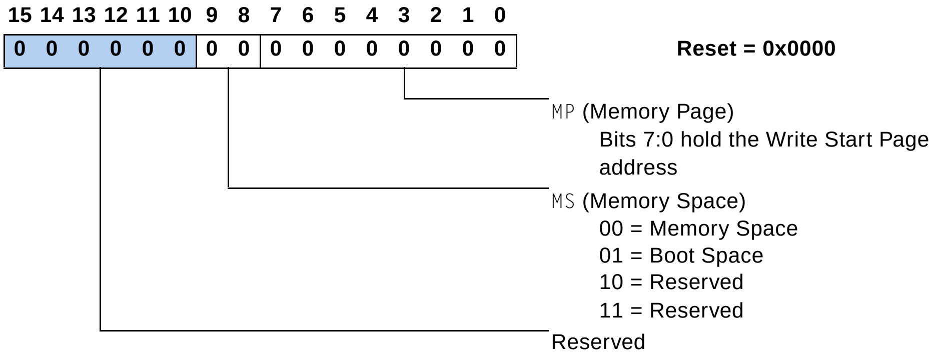


Figure B-8. DMA, MemDMA Channel Write Start Page (DMACW_SRP) Register Bits

MemDMA Channel Write Start Address (DMACW_SRA) Register

The register address is 0x02:0x103. This 16-bit read-only register holds the write transfer start address. The reset value is 0x0000.

MemDMA Channel Write Count (DMACW_CNT) Register

The register address is 0x02:0x104. The 16-bit Write Count (DMACW_CNT) read-only register holds the number of words in the transfer. The reset value is 0x0000.

MemDMA Channel Write Chain Pointer (DMACW_CP) Register

The register address is `0x02:0x105`. The 16-bit `DMACW_CP` register holds the pointer to address of next descriptor for a write transfer. The reset value is `0x0000`.

MemDMA Channel Write Chain Pointer Ready (DMACW_CPR) Register

The register address is `0x02:0x106`. Bit 0 in the 16-bit read-write register sets the status of the descriptor write operation. If bit = 1, the status is descriptor ready; 0 = wait. Bits 15–1 are not used.

This register should be set in the software after each descriptor is written to the internal memory. This lets the DMA know that a new descriptor block has been written in case the state machine has stalled because the descriptor block was not ready. This bit is cleared by the hardware upon beginning the data transfers of the described work block or after a reading a descriptor block with the ownership bit not set. Failure of the software to set this bit could potentially cause the DMA engine to permanently stall waiting for this bit.

MemDMA Channel Write Interrupt (DMACW_IRQ) Register

The register address is `0x02:0x107`. The DMA, MemDMA Channel can generate an interrupt upon a completion of a transfer. The interrupt occurs after the last write of the transfer is executed. Writing a one to bit 0 of the `DMACW_IRQ` register clears the DMA interrupt. Bits 15–1 are not used. The reset value is `0x0000`. Because this bit is sticky, it needs to be cleared in the interrupt service routine to prevent the interrupt from occurring repeatedly.

MemDMA Channel Read Pointer (DMACR_PTR) Register

The register address is `0x02:0x180`. This is a read-only register that holds the pointer to the current descriptor block for the DMA read operation. The reset value is `0x0000`.

MemDMA Channel Read Configuration (DMACR_CFG) Register

The register address is `0x02:0x181`. The `DMACR_CFG` register should only be written when starting DMA operation. The reset value is `0x0000`. The first descriptor's address should be written to the `DMACR_CP` Chain Pointer register followed by writing a "1" to the configuration register setting bit 0, the DMA enable (`DEN`) bit. This enables the DMA process and the first descriptor block will be fetched from internal memory. The dynamic allocation of descriptors is controlled by the "ownership" bit (bit 15) of each descriptor block.

Bit 1 (Direction) is set to 0 for the read operation.

The DMA, MemDMA Channel generates an interrupt if the "Interrupt on Error" bit 8 is set and the corresponding DMA channel is disabled during operation. For bit descriptions for this register (which are the same as the `DMACW_CFG` register), see Figure B-7 on page B-30.

MemDMA Channel Read Start Page (DMACR_SRP) Register

The register address is `0x02:0x182`. The 16-bit DMA Read Start Page (`DMACR_SRP`) register holds a running pointer to the DMA address that is being accessed and the memory space being used for a read operation.

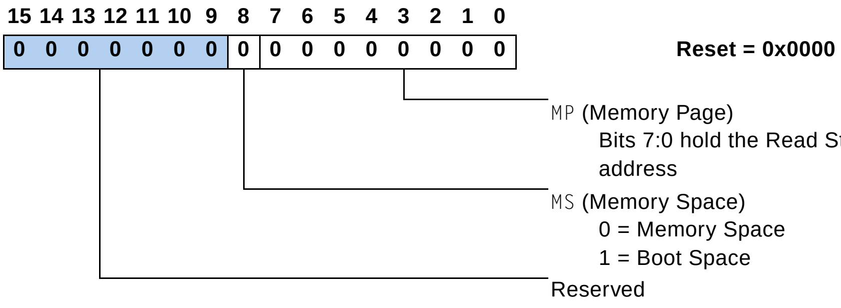


Figure B-9. DMA, MemDMA Channel Read Start Page (DMACR_SRP) Register Bits

MemDMA Channel Read Start Address (DMACR_SRA) Register

The register address is `0x02:0x183`. This 16-bit read-only register holds the read transfer start address. The reset value is `0x0000`.

MemDMA Channel Read Count (DMACR_CNT) Register

The register address is `0x02:0x184`. The 16-bit Read Count (`DMACR_CNT`) read-only register holds the number of words in the transfer. The reset value is `0x0000`.

MemDMA Channel Read Chain Pointer (DMACR_CP) Register

The register address is `0x02:0x185`. The 16-bit `DMACR_CP` register holds the pointer to the address of the next descriptor for a read transfer. The reset value is `0x0000`.

MemDMA Channel Read Chain Pointer Ready (DMACR_CPR) Register

The register address is `0x02:0x186`. `Bit 0` in the 16-bit read-write register sets the status of the descriptor write operation. If bit = 1, the status is descriptor ready; 0 = wait. Bits 15–1 are not used.

This register should be set in the software after each descriptor is written to the internal memory. This lets the DMA know that a new descriptor block has been written in case the state machine has stalled because the descriptor block was not ready. This bit is cleared by the hardware upon beginning the data transfers of the described work block or after a reading a descriptor block with the ownership bit not set. Failure of the software to set this bit could potentially cause the DMA engine to permanently stall waiting for this bit. The reset value is `0x0000`.

MemDMA Channel Read Interrupt (DMACR_IRQ) Register

The register address is `0x02:0x187`. The DMA, MemDMA Channel can generate an interrupt upon a completion of a transfer. The interrupt occurs after the last write of the transfer is executed. Writing a one to bit 0 of the `DMACR_IRQ` register clears the DMA interrupt. Bits 15–1 are not used.

SPORT Registers

The general-purpose programmable Serial port (SPORT) controller is designed to be used as an on-chip peripheral of a DSP. It supports a variety of serial data communications protocols and can provide a direct interconnection between processors in a multiprocessor system.

The SPORT can be viewed as two functional sections. The configuration section is a block of control registers (mapped to IO space memory) that the program must initialize before using the SPORTs. The data section is a register file used to transmit and receive values through the SPORT.

The section includes the following topics:

- "SPORT Transmit Configuration (SPx_TCR) Registers" on page B-38
- "SPORT Receive Configuration (SPx_RCR) Registers" on page B-38
- "SPORT Transmit Data (SPx_TX) Registers" on page B-41
- "SPORT Receive Data (SPx_RX) Registers" on page B-41
- "SPORT Transmit Serial Clock Divisor (SPx_TSCKDIV) Registers and SPORT Receive Serial Clock Divisor (SPx_RSCKDIV) Registers" on page B-42
- "SPORT Transmit Frame Sync Divisor (SPx_TFSDIV) Registers and SPORT Receive Frame Sync Divisor (SPx_RFSDIV) Registers" on page B-43
- "SPORT Status (SPx_STATR) Registers" on page B-43
- "SPORT Multichannel Transmit Channel Select (SPx_MTCSx) Registers" on page B-44
- "SPORT Multichannel Receive Channel Select (SPx_MRCSx) Registers" on page B-46
- "SPORT Multichannel Mode Configuration (SPx_MCMCx) Registers" on page B-47
- "SPORT DMA Receive Pointer (SPxDR_PTR) Registers" on page B-48

- "SPORT Receive DMA Configuration (SPxDR_CFG) Registers" on page B-48
- "SPORT Receive DMA Start Page (SPxDR_SRP) Registers" on page B-52
- "SPORT Receive DMA Start Address (SPxDR_SRA) Registers" on page B-53
- "SPORT Receive DMA Count (SPxDR_CNT) Registers" on page B-53
- "SPORT Receive DMA Chain Pointer (SPxDR_CP) Register" on page B-53
- "SPORT Receive DMA Chain Pointer Ready (SPxDR_CPR) Registers" on page B-54
- "SPORT Receive DMA Interrupt (SPxDR_IRQ) Registers" on page B-54
- "SPORT Transmit DMA Pointer (SPxDT_PTR) Registers" on page B-55
- "SPORT Transmit DMA Configuration (SPxDT_CFG) Registers" on page B-56
- "SPORT Transmit DMA Start Address (SPxDT_SRA) Registers" on page B-56
- "SPORT Transmit DMA Start Page (SPxDT_SRP) Registers" on page B-57
- "SPORT Transmit DMA Count (SPxDT_CNT) Registers" on page B-57
- "SPORT Transmit DMA Chain Pointer (SPxDT_CP) Registers" on page B-58

- "SPORT Transmit DMA Chain Pointer Ready (SPxDT_CPR) Registers" on page B-58
- "SPORT Transmit DMA Interrupt (SPxDT_IRQ) Registers" on page B-59

SPORT Transmit Configuration (SPx_TCR) Registers

SPORTs are enabled through bits in the Transmit and Receive Configuration (SPx_TCR) registers. The transmit registers' I/O addresses are:

```
SP0_TCR 0x02:0x200
SP1_TCR 0x03:0x000
SP2_TCR 0x03:0x200
```

Refer to Figure B-10 on page B-39 for bit descriptions.

Bit 0 (TSPEN) enables a SPORT for transmit if it is set to 1. When this bit is set, it locks further changes to the SPORT from occurring—for more information, see the discussion on on page 9-13. This bit is cleared at reset, disabling all SPORT channels. The reset value is 0x0000.

SPORT Receive Configuration (SPx_RCR) Registers

SPORTs are enabled through bits in the Receive (and Transmit) Configuration registers. The Receive registers' I/O addresses are:

```
SP0_RCR 0x02:0x201
SP1_RCR 0x03:0x001
SP2_RCR 0x03:0x201
```

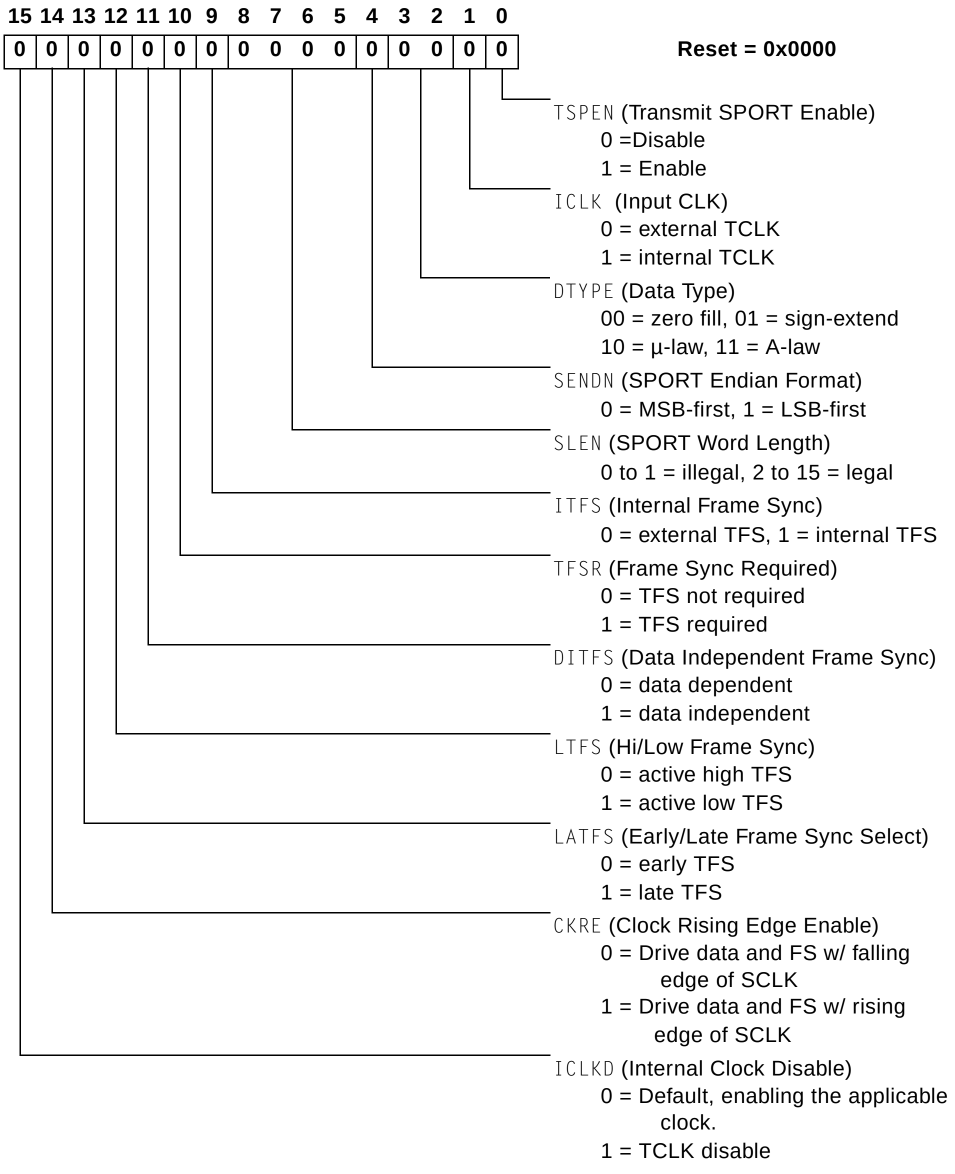

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TSPEN (Transmit SPORT Enable)
0 =Disable
1 = Enable
ICLK (Input CLK)
0 = external TCLK
1 = internal TCLK
DTYPE (Data Type)
00 = zero fill, 01 = sign-extend
10 = µ-law, 11 = A-law
SENDN (SPORT Endian Format)
0 = MSB-first, 1 = LSB-first
SLEN (SPORT Word Length)
0 to 1 = illegal, 2 to 15 = legal
ITFS (Internal Frame Sync)
0 = external TFS, 1 = internal TFS
TFSR (Frame Sync Required)
0 = TFS not required
1 = TFS required
DITFS (Data Independent Frame Sync)
0 = data dependent
1 = data independent
LTFS (Hi/Low Frame Sync)
0 = active high TFS
1 = active low TFS
LATFS (Early/Late Frame Sync Select)
0 = early TFS
1 = late TFS
CKRE (Clock Rising Edge Enable)
0 = Drive data and FS w/ falling edge of SCLK
1 = Drive data and FS w/ rising edge of SCLK
ICLKD (Internal Clock Disable)
0 = Default, enabling the applicable clock.
1 = TCLK disable

Figure B-10. SPORT Transmit Configuration (SPx_TCR) Registers' Bits

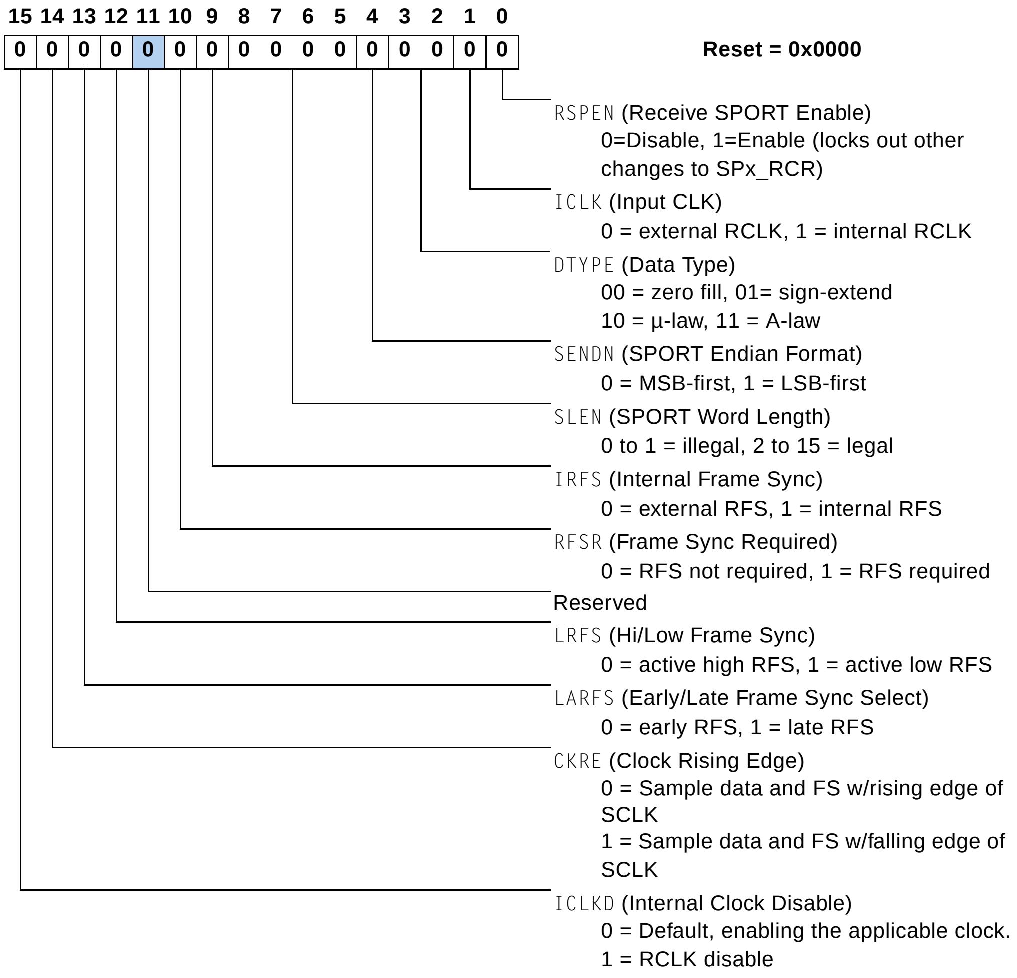

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RSPEN (Receive SPORT Enable)
0=Disable, 1=Enable (locks out other changes to SPx_RCR)
ICLK (Input CLK)
0 = external RCLK, 1 = internal RCLK
DTYPE (Data Type)
00 = zero fill, 01= sign-extend
10 = µ-law, 11 = A-law
SENDN (SPORT Endian Format)
0 = MSB-first, 1 = LSB-first
SLEN (SPORT Word Length)
0 to 1 = illegal, 2 to 15 = legal
IRFS (Internal Frame Sync)
0 = external RFS, 1 = internal RFS
RFSR (Frame Sync Required)
0 = RFS not required, 1 = RFS required
Reserved
LRFS (Hi/Low Frame Sync)
0 = active high RFS, 1 = active low RFS
LARFS (Early/Late Frame Sync Select)
0 = early RFS, 1 = late RFS
CKRE (Clock Rising Edge)
0 = Sample data and FS w/rising edge of SCLK
1 = Sample data and FS w/falling edge of SCLK
ICLKD (Internal Clock Disable)
0 = Default, enabling the applicable clock.
1 = RCLK disable

Figure B-11. SPORT Receive Configuration (SPx_RCR) Registers' Bits

SPORT Transmit Data (SPx_TX) Registers

These registers' addresses are:

```
SP0_TX 0x02:0x202
SP1_TX 0x03:0x002
SP2_TX 0x03:0x202
```

These registers can be accessed at any time during program execution using an IO Space access with immediate address. For example, the following instruction would ready SPORT to transmit a serial value, assuming SPORT is configured and enabled:

```
IOPG = 0x02;        /* selects I/O memory page 0x02 */
IO(0x202) = AX0;    /* loads TX from AX0, transmitting data */
```

The `TX` registers act like a two-location FIFO buffers because they have a data register plus an output shift register; two 16-bit words may be stored in the `TX` buffers at any one time. When the `TX` buffer is loaded and any previous word has been transmitted, the buffer contents are automatically loaded into the output shifter. An interrupt is generated when the output shifter has been loaded, signifying that the `TX` buffer is ready to accept the next word (i.e. the `TX` buffer is "not full"). This interrupt will not occur if serial port DMA is enabled. The reset value is `0x0000`.

SPORT Receive Data (SPx_RX) Registers

These registers' addresses are:

```
SP0_RX 0x02:0x203
SP1_RX 0x03:0x003
SP2_RX 0x03:0x203
```

These registers can be accessed at any time during program execution using an I/O space access with immediate address.

For example, the following instruction would access a serial value received on SPORT:

```
IOPG = 0x02;         /* selects I/O memory page 0x02 */
AY0 = IO(0x203);     /* loads AY0 from RX, received data */
```

The `RX` registers act like a two-location FIFO buffer because they have a data register plus an input shift register. They are read-only and their reset values are undefined.

Two 16-bit words can be stored in `RX` at any one time. The third word will overwrite the second if the first word has not been read out (by the Master core or the DMA controller). When this happens, the receive overflow status bit (`ROVF`) will be set in SPORT Status register. The overflow status is generated on the last bit of the second word. The `ROVF` status bit is "sticky" and is only cleared by disabling the serial port.

An interrupt is generated when the `RX` buffer has been loaded with a received word (that is, the `RX` buffer is "not empty"). This interrupt will be masked out if serial port DMA is enabled.

SPORT Transmit Serial Clock Divisor (SPx_TSCKDIV) Registers and SPORT Receive Serial Clock Divisor (SPx_RSCKDIV) Registers

The frequency of an internally generated clock is a function of the processor clock frequency (as seen at the `HCLK` pin) and the value of the 16-bit serial clock divide modulus registers: `TSCKDIV` and `RSCKDIV`. The reset value is `0x0000`.

The transmit `TSCKDIV` registers' addresses are:

```
SP0_TSCKDIV 0x02:0x204
SP1_TSCKDIV 0x03:0x004
SP2_TSCKDIV 0x03:0x204
```

The receive TSCKDIV registers' addresses are:

```
SP0_RSCKDIV 0x02:0x205
SP1_RSCKDIV 0x03:0x005
SP2_RSCKDIV 0x03:0x205
```

SPORT Transmit Frame Sync Divisor (SPx_TFSDIV) Registers and SPORT Receive Frame Sync Divisor (SPx_RFSDIV) Registers

These 16-bit registers specify how many transmit or receive clock cycles are counted before generating a TFS or RFS pulse (when the frame synch 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 reset value is 0x0000.

The transmit TFSDIV registers' addresses are:

```
SP0_TFSDIV 0x02:0x206
SP1_TFSDIV 0x03:0x006
SP2_TFSDIV 0x03:0x206
```

The receive RFSDIV registers' addresses are:

```
SP0_RFSDIV 0x02:0x207
SP1_RFSDIV 0x03:0x007
SP2_RFSDIV 0x03:0x207
```

SPORT Status (SPx_STATR) Registers

These registers' addresses are:

```
SP0_STATR 0x02:0x208
SP1_STATR 0x03:0x008
SP2_STATR 0x03:0x208
```

Figure B-12 on page B-45 provides bit descriptions.

The RXS and TXS status bits in the SPORT Status (SPx_STATR) registers are updated upon reads and writes from the core processor even when the serial port is disabled. The SPORT Status register is used to determine if the access to a SPORT RX or TX buffer can be made via determining their full or empty status. It is a read-only register; its reset value is undefined.

The transmit underflow status bit (TUVF) is set in the SPORT Status register when a transmit frame synch occurs and no new data has been loaded into the SPORT TX register. The TUVF status bit is "sticky" and is only cleared by disabling the serial port.

When the SPORT RX buffer is full, the receive overflow status bit (ROVF) is set in SPORT Status register. The overflow status is generated on the last bit of the second word. The ROVF status bit is "sticky" and is only cleared by disabling the serial port.

The 7-bit CHNL field is the read-only status indicator that shows which channel is currently selected during multi-channel operation. CHNL6-0 increments by one as each channel is serviced. In channel select offset mode, the CHNL value is reset to 0 after the offset has been completed. For example, with offset equals to 21 and a window of 8, in the regular mode the counter will display a value between 0 and 28, while in channel select offset mode, the counter will reset to 0 after counting up to 21, and then the frame will complete when the CHNL reaches 8th channel (value of 7).

SPORT Multichannel Transmit Channel Select (SPx_MTCSx) Registers

The multi-channel selection registers are used to enable and disable individual channels. The MTCSx register specifies the active transmit channels. Each register has 16 bits, corresponding to the 16 channels. Setting a bit

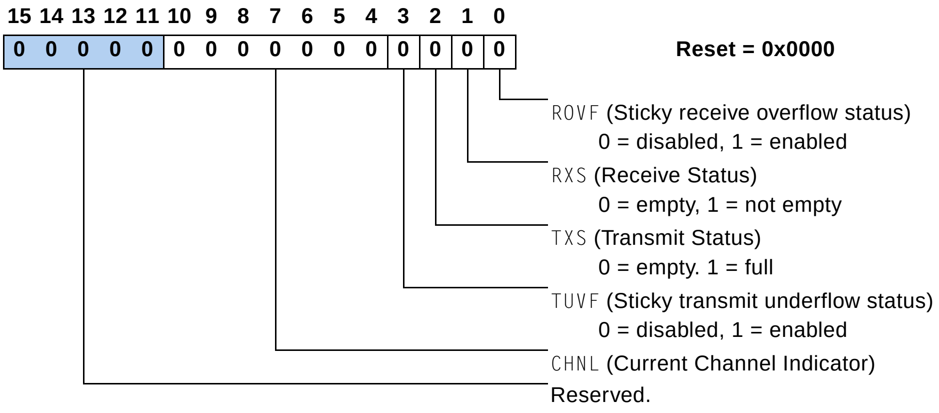


Figure B-12. SPORT Status (SPx_STATR) Registers' Bits

enables that channel so that the serial port will select its word from the multiple-word block of data (for either receive or transmit). For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit to 1 in a `MTCSx` register causes the serial port to transmit the word in that channel's position of the data stream. Clearing the bit to 0 in the `MTCSx` register causes the serial port's `DT` (data transmit) pin to three-state during the time slot of that channel. The reset value is `0x0000`.

Register addresses are listed in Table B-4.

Table B-5. SPx_MTCSx Register Addresses

SPORT 0		SPORT1		SPORT2	
Register	**Address**	**Register**	**Address**	**Register**	**Address**
SP0_MTCS0	0x02:0x209	SP1_MTCS0	0x03:0x009	SP2_MTCS0	0x03:0x209
SP0_MTCS1	0x02:0x20A	SP1_MTCS1	0x03:0x00A	SP2_MTCS1	0x03:0x20A
SP0_MTCS2	0x02:0x20B	SP1_MTCS2	0x03:0x00B	SP2_MTCS2	0x03:0x20B
SP0_MTCS3	0x02:0x20C	SP1_MTCS3	0x03:0x00C	SP2_MTCS3	0x03:0x20C

Table B-5. SPx_MTCSx Register Addresses (Cont'd)

SPORT 0		SPORT1		SPORT2	
Register	Address	Register	Address	Register	Address
SP0_MTCS4	0x02:0x20D	SP1_MTCS4	0x03:0x00D	SP2_MTCS4	0x03:0x20D
SP0_MTCS5	0x02:0x20E	SP1_MTCS5	0x03:0x00E	SP2_MTCS5	0x03:0x20E
SP0_MTCS6	0x02:0x20F	SP1_MTCS6	0x03:0x00F	SP2_MTCS6	0x03:0x20F
SP0_MTCS7	0x02:0x210	SP1_MTCS7	0x03:0x010	SP2_MTCS7	0x03:0x210

SPORT Multichannel Receive Channel Select (SPx_MRCSx) Registers

The multichannel receive channel selection registers are used to enable and disable individual channels. The MRCSx register specifies the active receive channels. Each register has 16 bits, corresponding to the 16 channels. Setting a bit enables that channel so that the serial port will select its word from the multiple-word block of data (for either receive or transmit). For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit to 1 in the MRCSx register causes the serial port to receive the word in that channel's position of the data stream; the received word is loaded into the RX buffer. Clearing the bit to 0 in the MRCSx register causes the serial port to ignore the data. The reset value is 0x0000.

Register addresses are listed in Table B-6.

Table B-6. SPx_MRCSx Register Addresses

SPORT0		SPORT1		SPORT2	
Register	Address	Register	Address	Register	Address
SP0_MRCS0	0x02:0x211	SP1_MRCS0	0x03:0x011	SP2_MRCS0	0x03:0x211
SP0_MRCS1	0x02:0x212	SP1_MRCS1	0x03:0x012	SP2_MRCS1	0x03:0x212

Table B-6. SPx_MRCSx Register Addresses (Cont'd)

SPORT0		SPORT1		SPORT2	
Register	Address	Register	Address	Register	Address
SP0_MRCS2	0x02:0x213	SP1_MRCS2	0x03:0x013	SP2_MRCS2	0x03:0x213
SP0_MRCS3	0x02:0x214	SP1_MRCS3	0x03:0x014	SP2_MRCS3	0x03:0x214
SP0_MRCS4	0x02:0x215	SP1_MRCS4	0x03:0x015	SP2_MRCS4	0x03:0x215
SP0_MRCS5	0x02:0x216	SP1_MRCS5	0x03:0x016	SP2_MRCS5	0x03:0x216
SP0_MRCS6	0x02:0x217	SP1_MRCS6	0x03:0x017	SP2_MRCS6	0x03:0x217
SP0_MRCS7	0x02:0x218	SP1_MRCS7	0x03:0x018	SP2_MRCS7	0x03:0x218

SPORT Multichannel Mode Configuration (SPx_MCMCx) Registers

There are two SPx_MCMCx registers for each SPORT. Their addresses are in Table B-6.

Table B-7. SPx_MCMCx Registers Addresses

SPORT0		SPORT1		SPORT2	
Register	Address	Register	Address	Register	Address
SP0_MCMC1	0x02:0x219	SP1_MCMC1	0x03:0x019	SP2_MCMC1	0x03:0x219
SP0_MCMC2	0x02:0x21A	SP1_MCMC2	0x03:0x01A	SP2_MCMC2	0x03:0x21A

Refer to Figure B-13 on page B-49 and Figure B-14 on page B-50 for SPx_MCMCx registers' bit descriptions.

The SPx_MCMCx registers are used to enable multi-channel mode. Setting the MCM bit enables multi-channel operation for both receive and transmit sides of the SPORT. A transmitting SPORT must therefore be in multi-channel mode if the receiving SPORT is in multi-channel mode.

The value of MFD is the number of serial clock cycles of the delay. Multi-channel frame delay allows the processor to work with different types of T1 interface devices.

A value of zero for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15.

The reset value for both `SPx_MCMCx` registers is `0x0000`.

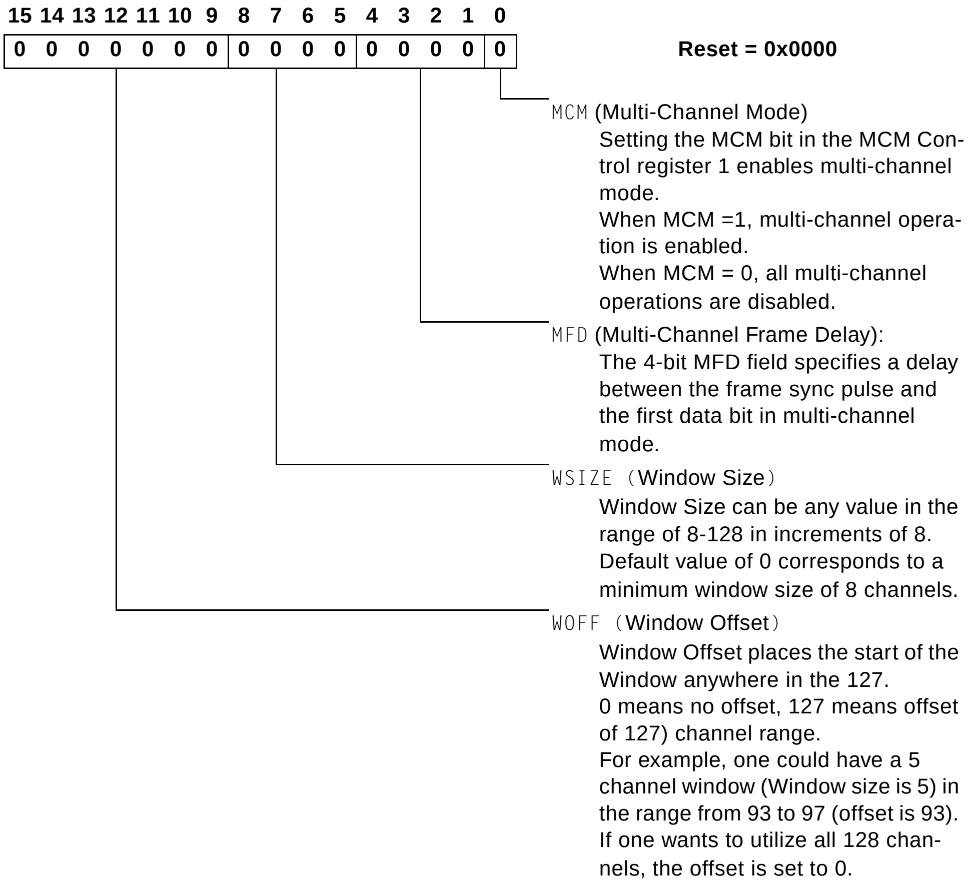

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MCM (Multi-Channel Mode)
Setting the MCM bit in the MCM Control register 1 enables multi-channel mode.
When MCM =1, multi-channel operation is enabled.
When MCM = 0, all multi-channel operations are disabled.
MFD (Multi-Channel Frame Delay):
The 4-bit MFD field specifies a delay between the frame sync pulse and the first data bit in multi-channel mode.
WSIZE (Window Size)
Window Size can be any value in the range of 8-128 in increments of 8. Default value of 0 corresponds to a minimum window size of 8 channels.
WOFF (Window Offset)
Window Offset places the start of the Window anywhere in the 127.
0 means no offset, 127 means offset of 127) channel range.
For example, one could have a 5 channel window (Window size is 5) in the range from 93 to 97 (offset is 93).
If one wants to utilize all 128 channels, the offset is set to 0.

Figure B-13. SPORT Multi-Channel Configuration (SPx_MCMC1) Register Bits

A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

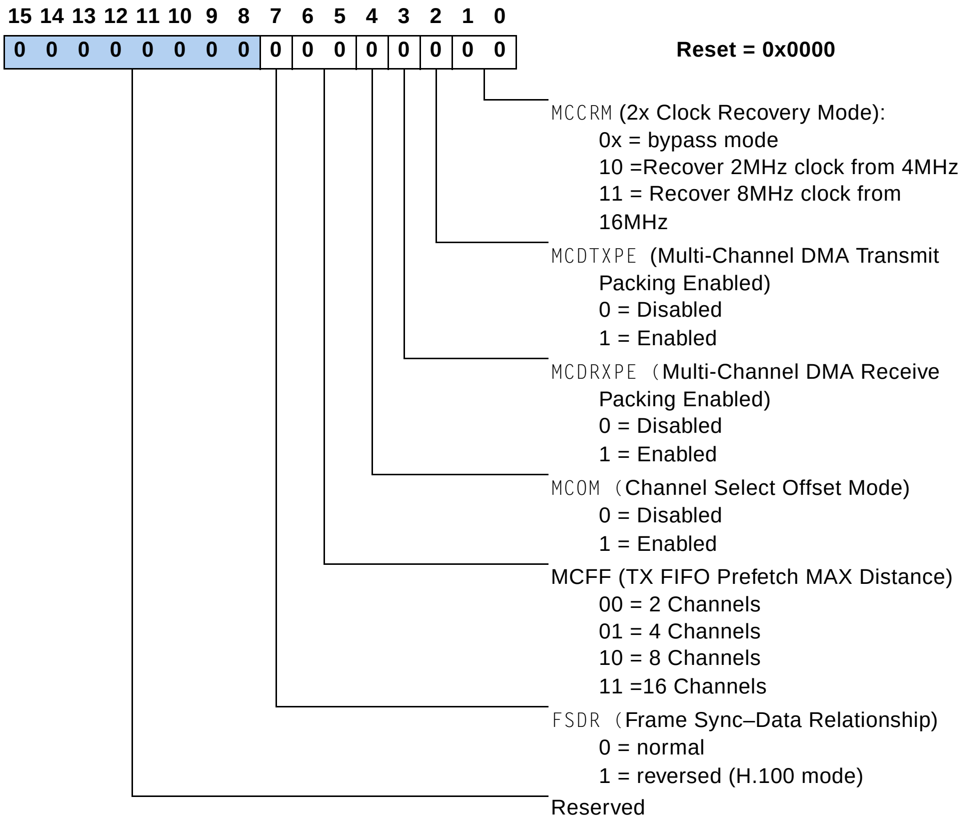

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MCCRM (2x Clock Recovery Mode):
0x = bypass mode
10 =Recover 2MHz clock from 4MHz
11 = Recover 8MHz clock from 16MHz
MCDTXPE (Multi-Channel DMA Transmit Packing Enabled)
0 = Disabled
1 = Enabled
MCDRXPE (Multi-Channel DMA Receive Packing Enabled)
0 = Disabled
1 = Enabled
MCOM (Channel Select Offset Mode)
0 = Disabled
1 = Enabled
MCFF (TX FIFO Prefetch MAX Distance)
00 = 2 Channels
01 = 4 Channels
10 = 8 Channels
11 =16 Channels
FSDR (Frame Sync–Data Relationship)
0 = normal
1 = reversed (H.100 mode)
Reserved

Figure B-14. SPORT Multi-Channel Configuration (SPx_MCMC2) Register Bits

SPORT DMA Receive Pointer (SPxDR_PTR) Registers

These registers' addresses are:

```
SP0DR_PTR 0x02:0x300
SP1DR_PTR 0x03:0x100
SP2DR_PTR 0x03:0x300
```

These 16-bit read-only registers hold the pointer to the current descriptor block for the SPORT DMA operation. The reset value is `0x0000`.

SPORT Receive DMA Configuration (SPxDR_CFG) Registers

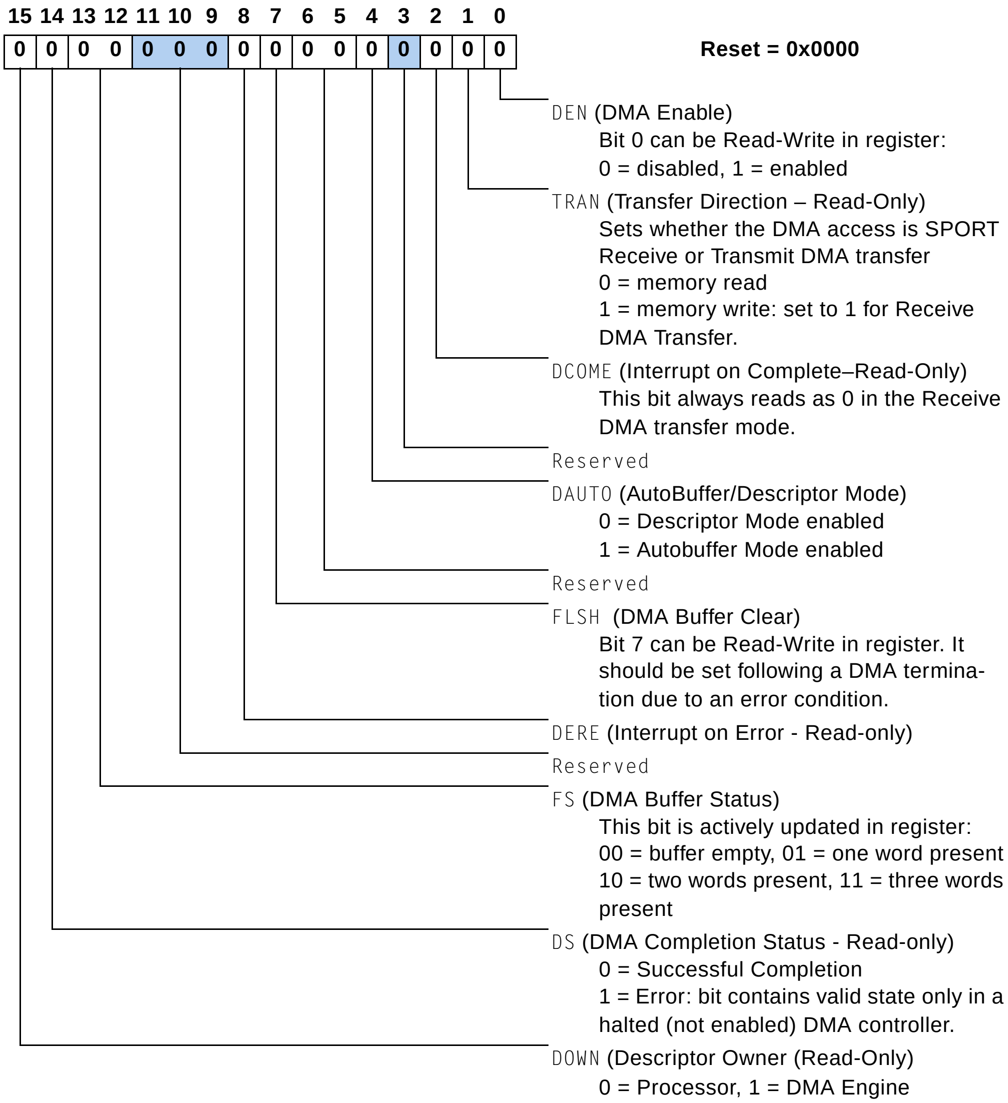
These register's addresses are:

```
SP0DR_CFG 0x02:0x301
SP1DR_CFG 0x03:0x101
SP2DR_CFG 0x03:0x301
```

During SPORT initialization, the program can write the head address of the first DMA descriptor block to the Receive DMA Descriptor Pointer (`SPxDR_PTR`) register and then set the DMA enable bit in the Receive DMA Configuration (`SPxDR_CFG`) registers. The DMA Configuration register maintains real-time DMA buffer status.

Each SPORT DMA channel has an enable bit (DMA Enable) in these registers for each of the three serial ports. When DMA is not enabled for a particular channel, the SPORT generates an interrupt every time it has received a data word. The reset value is `0x0000`. Refer to Figure B-15 on page B-51 for bit descriptions.

The `DCOME` bit will result in an interrupt of the core DSP once the last word of the DMA transfer has completed transmission (for a SPORT transmit), or has been written to memory (for a SPORT receive).



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DEN (DMA Enable)
Bit 0 can be Read-Write in register:
0 = disabled, 1 = enabled
TRAN (Transfer Direction – Read-Only)
Sets whether the DMA access is SPORT Receive or Transmit DMA transfer
0 = memory read
1 = memory write: set to 1 for Receive DMA Transfer.
DCOME (Interrupt on Complete–Read-Only)
This bit always reads as 0 in the Receive DMA transfer mode.
Reserved
DAUTO (AutoBuffer/Descriptor Mode)
0 = Descriptor Mode enabled
1 = Autobuffer Mode enabled
Reserved
FLSH (DMA Buffer Clear)
Bit 7 can be Read-Write in register. It should be set following a DMA termination due to an error condition.
DERE (Interrupt on Error - Read-only)
Reserved
FS (DMA Buffer Status)
This bit is actively updated in register:
00 = buffer empty, 01 = one word present
10 = two words present, 11 = three words present
DS (DMA Completion Status - Read-only)
0 = Successful Completion
1 = Error: bit contains valid state only in a halted (not enabled) DMA controller.
DOWN (Descriptor Owner (Read-Only)
0 = Processor, 1 = DMA Engine

Figure B-15. SPORT Receive DMA Configuration (SPxDR_CFG) Registers' Bits

The DMA buffer clear bit (`FLSH`) has write-one-to-clear characteristics. It may also be used by a descriptor block load to initialize a DMA FIFO to a cleared condition prior starting a DMA transfer. Not only is the DMA extended buffer cleared, but the SPORT transmit double buffer and receive triple buffers are also cleared.

SPORT Receive DMA Start Page (SPxDR_SRP) Registers

These registers' addresses are:

```
SP0DR_SRP 0x02:0x302
SP1DR_SRP 0x03:0x102
SP2DR_SRP 0x03:0x302
```

These registers hold a running pointer to the DMA address that is being accessed and the type of memory space being used. It is a read-only register (can be written in the Autobuffer Mode).

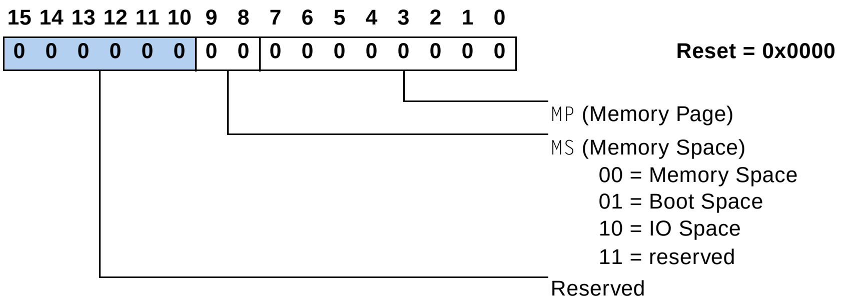


Figure B-16. SPORT Receive DMA Start (SPxDR_SRP) Registers' Bits

SPORT Receive DMA Start Address (SPxDR_SRA) Registers

These registers' addresses are:

```
SP0DR_SRA 0x02:0x303
SP1DR_SRA 0x03:0x103
SP2DR_SRA 0x03:0x303
```

The DMA Start Address (`SPxDR_SRA`) registers maintain a running pointer to the DMA address that is being accessed. They are read-only (can be written in the autobuffer mode). The reset value is `0x0000`.

SPORT Receive DMA Count (SPxDR_CNT) Registers

These registers' addresses are:

```
SP0DR_CNT 0x02:0x304
SP1DR_CNT 0x03:0x104
SP2DR_CNT 0x03:0x304
```

Bits 12:0 in the SPORT DMA Word Count (`SPxDR_CNT`) registers hold the number of remaining words in the transfer. These are read-only registers (can be written in the autobuffer mode). The reset value is `0x0000`.

SPORT Receive DMA Chain Pointer (SPxDR_CP) Register

These registers' addresses are:

```
SP0DR_CP 0x02:0x305
SP1DR_CP 0x03:0x105
SP2DR_CP 0x03:0x305
```

The 16-bit DMA Chain (Next Descriptor) Pointer (`SPxDR_CP`) register maintains the head address of the next DMA descriptor block. During SPORT initialization, the programmer will write the head address of the first DMA descriptor block to the Receive (or Transmit) DMA Chain Pointer register and then set the DMA enable bit in the Transmit or Receive DMA Configuration registers.

Once a DMA process has started, no further control of the SPORT controller or the DMA process should be performed by write accesses to the SPORT DMA control registers. Performing IO Space writes to these registers during operation will have no effect on DMA transfers since these registers are read-only. The reset value is `0x0000`.

SPORT Receive DMA Chain Pointer Ready (SPxDR_CPR) Registers

These registers' addresses are:

```
SP0DR_CPR 0x02:0x306
SP1DR_CPR 0x03:0x106
SP2DR_CPR 0x03:0x306
```

These registers are used to show the descriptor's status. A DMA Chain Pointer Ready (`SPxDR_CPR`) register is needed for the descriptor ownership setup. They are write-only registers (always read as zero). The reset value is `0x0000`.

SPORT Receive DMA Interrupt (SPxDR_IRQ) Registers

These registers' addresses are:

```
SP0DR_IRQ 0x02:0x307
SP1DR_IRQ 0x03:0x107
SP2DR_IRQ 0x03:0x307
```

Each SPORT DMA unit generates an interrupt upon a completion of a data transfer. Writing a one to bit 0 clears the DMA interrupt. Writing a one to bit 1 clears the error interrupt condition.

Refer to Figure B-17 on page B-55 for bit descriptions.

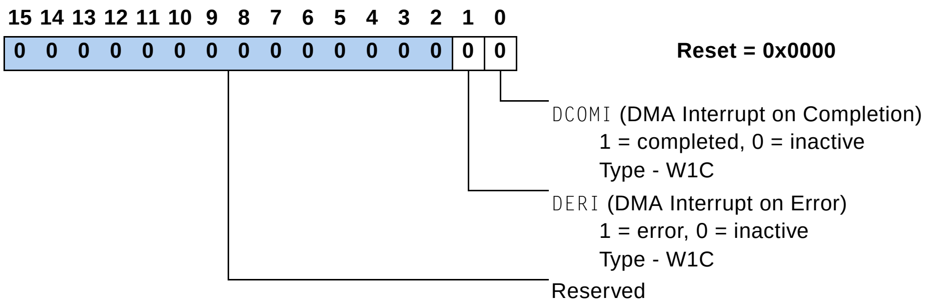


Figure B-17. SPORT Receive DMA Interrupt (SPxDR_IRQ) Registers' Bits

SPORT Transmit DMA Pointer (SPxDT_PTR) Registers

These registers' addresses are:

```
SP0DT_PTR 0x02:0x380
SP1DT_PTR 0x03:0x180
SP2DT_PTR 0x03:0x380
```

These registers hold the address for the current transmit control block (descriptor) in a chained DMA operation. The reset value is `0x0000`.

SPORT Transmit DMA Configuration (SPxDT_CFG) Registers

Register addresses are:

```
SP0DT_CFG 0x02:0x381
SP1DT_CFG 0x03:0x181
SP2DT_CFG 0x03:0x381
```

During SPORT initialization, the program can write the head address of the first DMA descriptor block to the Transmit DMA Descriptor Pointer register and then set the DMA enable bit in the Transmit DMA Configuration registers. The DMA Configuration register maintains real-time DMA buffer status. The reset value is `0x0000`.

Each SPORT DMA channel has an enable bit (DMA enable) in these registers for each of the three serial ports. When DMA is not enabled for a particular channel, the SPORT generates an interrupt every time it has started to transmit a data word.

`For information on the bits in this register (which are the same as the SPxDR_CFG` register), see Figure B-15 on page B-51.

SPORT Transmit DMA Start Address (SPxDT_SRA) Registers

Register addresses are:

```
SP0DT_SRA 0x02:0x383
SP1DT_SRA 0x03:0x183
SP2DT_SRA 0x03:0x383
```

The DMA Start Address (`SPxDT_SRA`) register holds a running pointer to the DMA address that is being accessed. These are read-only registers (can be written in the autobuffer mode). The reset value is `0x0000`.

SPORT Transmit DMA Start Page (SPxDT_SRP) Registers

Register addresses are:

```
SP0DT_SRP 0x02:0x382
SP1DT_SRP 0x03:0x182
SP2DT_SRP 0x03:0x382
```

The SPORT DMA Start Page (`SPxDT_SRP`) register (as well as the SPORT DMA Start Address and DMA Word Count registers) maintain a running pointer to the DMA address that is being accessed and the number of remaining words in the transfer. These are read-only registers (can be written in the autobuffer mode).

Refer to Figure B-19 on page B-59 for bit descriptions.

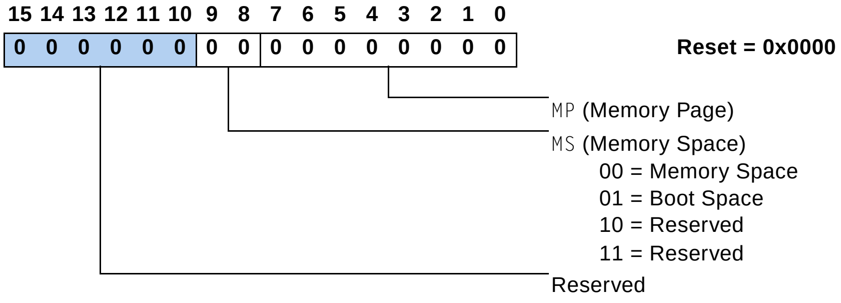


Figure B-18. SPORT Transmit DMA Start Page (SPxDT_SRP) Registers' Bits

SPORT Transmit DMA Count (SPxDT_CNT) Registers

These register addresses are:

```
SP0DT_CNT 0x02:0x384
SP1DT_CNT 0x03:0x184
SP2DT_CNT 0x03:0x384
```

The DMA Word Count register holds the DMA block word count (the number of remaining words in the transfer). These are read-only registers (can be written in the autobuffer mode). The reset value is `0x0000`.

SPORT Transmit DMA Chain Pointer (SPxDT_CP) Registers

These register addresses are:

```
SP0DT_CP 0x02:0x385
SP1DT_CP 0x03:0x185
SP2DT_CP 0x03:0x385
```

The SPORT Transmit DMA Chain Pointer (`SPxDT_CP`) register holds the head address of the next DMA descriptor block. During SPORT initialization, the programmer will write the head address of the first DMA descriptor block to the Transmit (or Receive) DMA Chain Pointer register and then set the DMA enable bit in the Transmit or Receive DMA Configuration registers. Once a DMA process has started, no further control of the SPORT controller or the DMA process should be performed by write accesses to the SPORT DMA control registers.

Performing I/O space writes to these registers during operation will have no effect on DMA transfers since these registers are read-only. The reset value is `0x0000`.

SPORT Transmit DMA Chain Pointer Ready (SPxDT_CPR) Registers

Registers' addresses are:

```
SP0DT_CPR 0x02:0x386
SP1DT_CPR 0x03:0x186
SP2DT_CPR 0x03:0x386
```

These registers are used to show the descriptor's status. A DMA Chain Pointer Ready register is needed for the descriptor ownership setup. They are write-only registers (always read as zero). The reset value is `0x0000`.

SPORT Transmit DMA Interrupt (SPxDT_IRQ) Registers

These register addresses are:

```
SP0DT_IRQ 0x02:0x387
SP1DT_IRQ 0x03:0x187
SP2DT_IRQ 0x03:0x387
```

Each SPORT DMA unit generates an interrupt upon a completion of a data transfer. Writing a one to bit 0 clears the DMA interrupt. Writing a one to bit 1 clears the error interrupt condition.

Refer to Figure B-18 on page B-57 for bit descriptions.

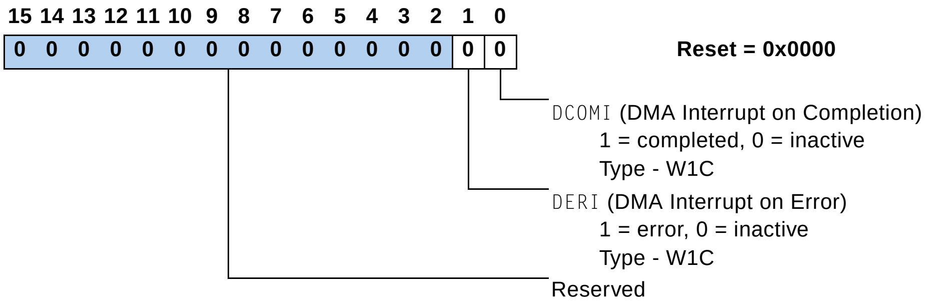


Figure B-19. SPORT Transmit DMA Interrupt (SPxDT_IRQ) Registers' Bits

Serial Peripheral Interface Registers

The Serial Peripheral Interface module (SPI) provides functionality for a generic configurable serial port interface based on the SPI standard.

The Serial Peripheral Interface is essentially a shift register that serially transmits and receives data bits to/from other SPI-compatible devices. During an SPI transfer, data is simultaneously transmitted (shifted out serially) and received (shifted in serially). A serial clock line synchronizes shifting and sampling of the information on the two serial data lines.

The section includes the following topics:

- “SPI DMA Next Chain Pointer (SPIxD_CP) Registers” on page B-71
- “SPI DMA Chain Pointer Ready (SPIxD_CPR) Registers” on page B-71
- “SPI DMA Interrupt (SPIxD_IRQ) Registers” on page B-71

SPI Control (SPICTLx) Registers

Registers’ addresses are: `SPICTL0 0x04:0x000` and `SPICTL1 0x04:0x200`.

The SPI control register (`SPICTLx`) is used to configure the SPI system. The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length bit (`SIZE`) in `SPICTL`. There are two special bits which can also be modified by the hardware: `SPE` and `MSTR`.

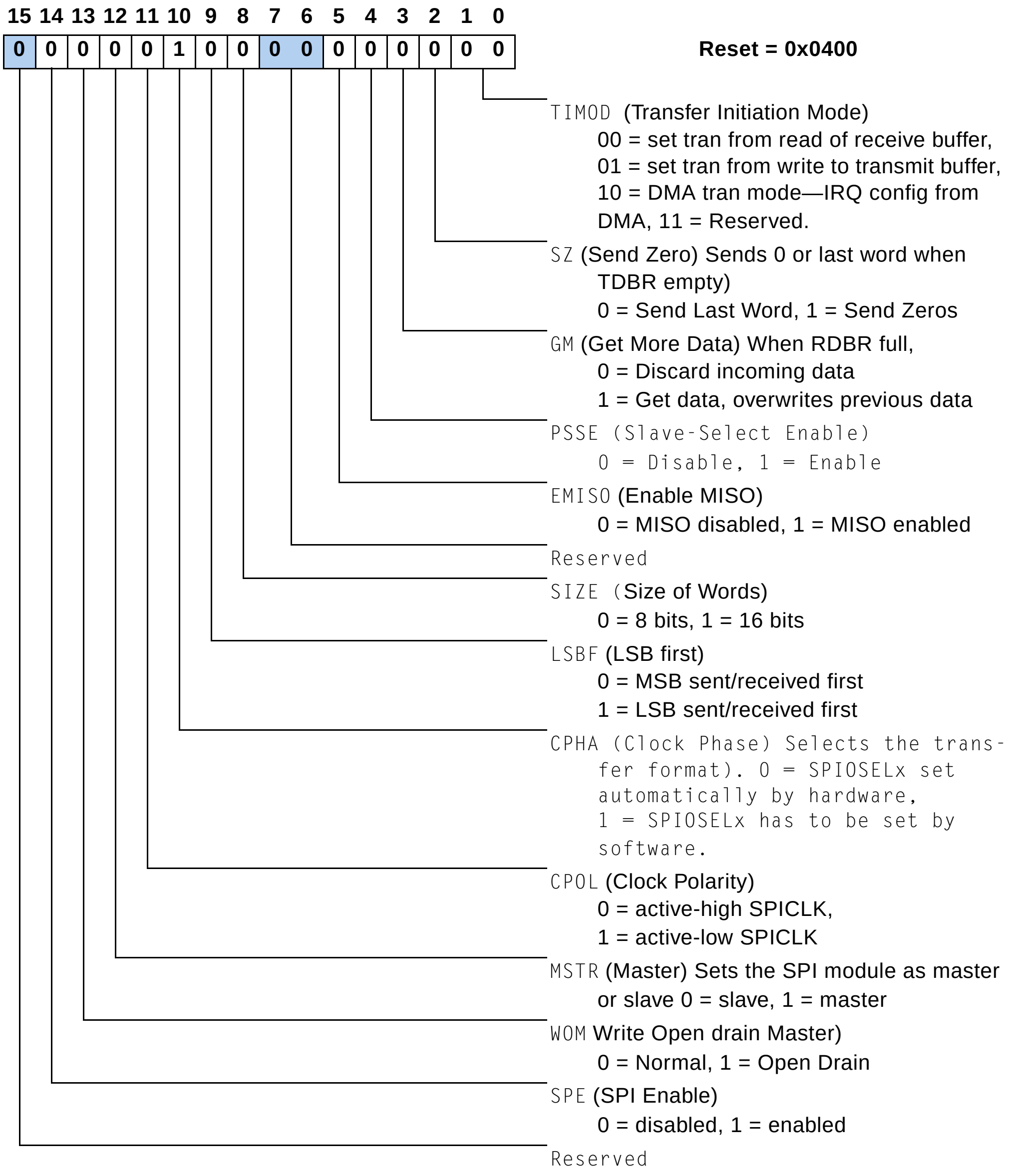
The SPI control register bit descriptions are as shown in Figure B-20 on page B-62.

Note: Bit default is 0 unless marked otherwise.

Bits 1–0 are used to initiate transfers to/from the receive/transmit buffers. When set to 00, the Interrupt is active when the receive buffer is full. When set to 01, the Interrupt is active when the transmit buffer is empty.

Bit 4 is used to enable the SPISS input for master. When not used, SPISS can be disabled, freeing up a chip pin as general purpose I/O.

Bit 5 allows to enable the `MISO` pin as an output. This is needed in an environment where 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 set.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0400
TIMOD (Transfer Initiation Mode)
00 = set tran from read of receive buffer,
01 = set tran from write to transmit buffer,
10 = DMA tran mode—IRQ config from DMA, 11 = Reserved.
SZ (Send Zero) Sends 0 or last word when TDBR empty)
0 = Send Last Word, 1 = Send Zeros
GM (Get More Data) When RDBR full,
0 = Discard incoming data
1 = Get data, overwrites previous data
PSSE (Slave-Select Enable)
0 = Disable, 1 = Enable
EMISO (Enable MISO)
0 = MISO disabled, 1 = MISO enabled
Reserved
SIZE (Size of Words)
0 = 8 bits, 1 = 16 bits
LSBF (LSB first)
0 = MSB sent/received first
1 = LSB sent/received first
CPHA (Clock Phase) Selects the transfer format). 0 = SPIOSELx set automatically by hardware, 1 = SPIOSELx has to be set by software.
CPOL (Clock Polarity)
0 = active-high SPICLK,
1 = active-low SPICLK
MSTR (Master) Sets the SPI module as master or slave 0 = slave, 1 = master
WOM Write Open drain Master)
0 = Normal, 1 = Open Drain
SPE (SPI Enable)
0 = disabled, 1 = enabled
Reserved

Figure B-20. SPI Control (SPICTLx) Registers' Bits

Bit 10 (CPHA) 5 allows to enable the MISO pin as an output. This is needed in an environment where 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 set.

SPI Flag (SPIFLGx) Registers

Register addresses are: SPIFLG0 0x04:0x001 and SPIFLG1 0x04:0x200.

The SPI Flag (SPIFLGx) register is a read/write register that is used to enable individual SPI slave-select lines when the SPI is enabled as a master. There are 7 bits which select the outputs to be driven as slave-select lines (FLS) and 7 bits which can activate the selected slave-selects (FLG).

The following table provides the bit mappings for the SPIFLG0 register.

Table B-8. SPIFLG0 Register Bits

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	SPI0SEL1 Enable	PF2	0
2	FLS2	SPI0SEL2 Enable	PF4	0
3	FLS3	SPI0SEL3 Enable	PF6	0
4	FLS4	SPI0SEL4 Enable	PF8	0
5	FLS5	SPI0SEL5 Enable	PF10	0
6	FLS6	SPI0SEL6 Enable	PF12	0
7	FLS7	SPI0SEL7 Enable	PF14	0
8		Reserved		1
9	FLG1	SPI0SEL1 Value	PF2	1
10	FLG2	SPI0SEL2 Value	PF4	1
11	FLG3	SPI0SEL3 Value	PF6	1

Table B-8. SPIFLG0 Register Bits (Cont'd)

Bit	Name	Function	PFx Pin	Default
12	FLG4	SPI0SEL4 Value	PF8	1
13	FLG5	SPI0SEL5 Value	PF10	1
14	FLG6	SPI0SEL6 Value	PF12	1
15	FLG7	SPI0SEL7 Value	PF14	1

The following table provides the bit mappings for the `SPIFLG1` register.

Table B-9. SPIFLG1 Register Bits

Bit	Name	Function	PFx Pin	Default
0		Reserved		0
1	FLS1	SPI1SEL1 Enable	PF3	0
2	FLS2	SPI1SEL2 Enable	PF5	0
3	FLS3	SPI1SEL3 Enable	PF7	0
4	FLS4	SPI1SEL4 Enable	PF9	0
5	FLS5	SPI1SEL5 Enable	PF11	0
6	FLS6	SPI1SEL6 Enable	PF13	0
7	FLS7	SPI1SEL7 Enable	PF15	0
8		Reserved		1
9	FLG1	SPI1SEL1 Value	PF3	1
10	FLG2	SPI1SEL2 Value	PF5	1
11	FLG3	SPI1SEL3 Value	PF7	1
12	FLG4	SPI1SEL4 Value	PF9	1
13	FLG5	SPI1SEL5 Value	PF11	1
14	FLG6	SPI1SEL6 Value	PF13	1
15	FLG7	SPI1SEL7 Value	PF15	1

If the SPI is enabled and configured as a master, up to seven of the chip's general-purpose flag I/O pins may be used as slave-select outputs.

SPI Status (SPISTx) Registers

These registers' addresses are: `SPIST0 0x04:0x002` and `SPIST1 0x04:0x202`.

Note: Bit default is 0 unless marked otherwise.

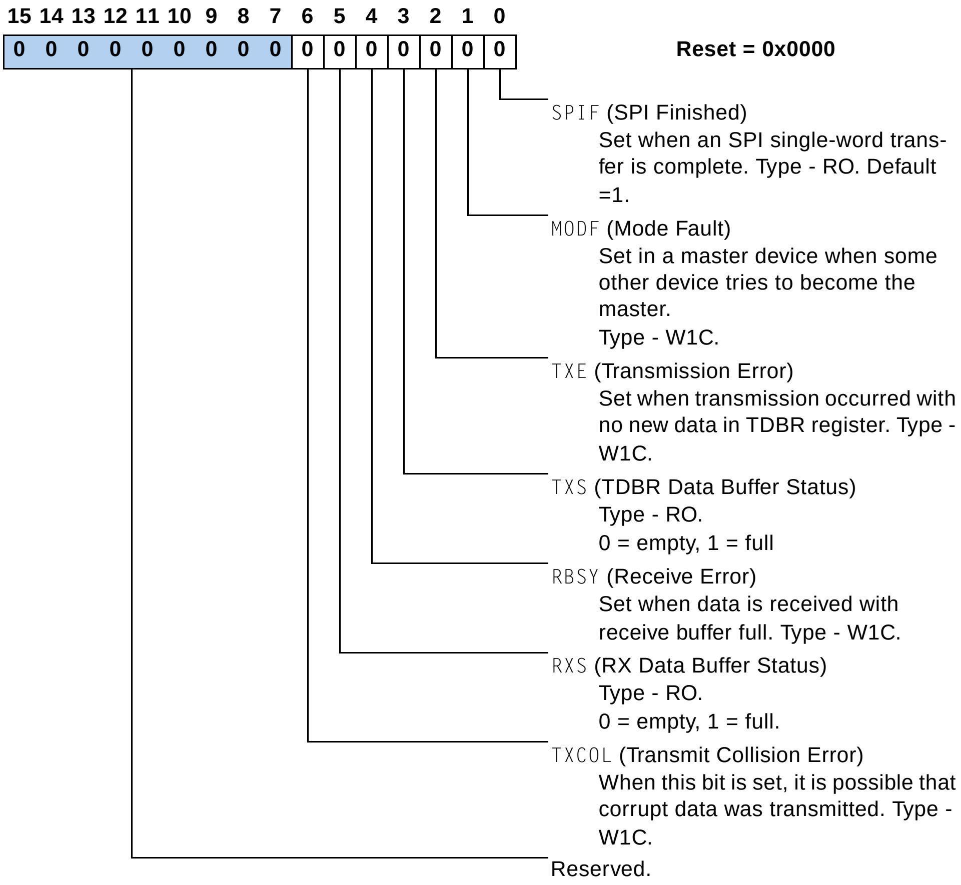
The SPI Status registers can be read at any time. Some of the register's bits are read-only (RO), and others are cleared by a write-1 (W1C) operation. Bits which merely provide information about the SPI are read-only; these bits are set and cleared by the hardware. W1C bits are set when an error condition occurs; these bits are set by hardware, and must be cleared by software. To clear a W1C bit, write a 1 to the desired bit position of the `SPIST` 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.

SPI Transmit Buffer (TDBRx) Registers

These registers' addresses are: `TDBR0 0x04:0x003` and `TDBR1 0x04:0x203`.

These are 16-bit read-write (RW) registers. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in `TDBR` is loaded into the shift register `(SFDR)`. A normal core read of `TDBR` can be done at any time and does not interfere with, or initiate, SPI transfers. The reset value is `0x0000`.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SPIF (SPI Finished)
Set when an SPI single-word transfer is complete. Type - RO. Default =1.
MODF (Mode Fault)
Set in a master device when some other device tries to become the master.
Type - W1C.
TXE (Transmission Error)
Set when transmission occurred with no new data in TDBR register. Type - W1C.
TXS (TDBR Data Buffer Status)
Type - RO.
0 = empty, 1 = full
RBSY (Receive Error)
Set when data is received with receive buffer full. Type - W1C.
RXS (RX Data Buffer Status)
Type - RO.
0 = empty, 1 = full.
TXCOL (Transmit Collision Error)
When this bit is set, it is possible that corrupt data was transmitted. Type - W1C.
Reserved.

Figure B-21. SPI Status (SPISTx) Registers' Bits

When the DMA is enabled for transmit operation (described later in this document), the DMA automatically loads TDBR with the data to be transmitted. Just prior to the beginning of a data transfer, the data in TDBR is loaded into the shift register. A normal core write to TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, whatever is in TDBR will repeatedly be transmitted. A normal core write to TDBR is permitted, and this data will be transmitted.

If the "send zeros" control bit (SZ) is set, TDBR may be reset to 0 under certain circumstances. If multiple writes to TDBR occur while a transfer is already in progress, only the last data which was written will be transmitted; all intermediate values written to TDBR will not be transmitted. Multiple writes to TDBR are possible, but not recommended.

Receive Data Buffer (RDBRx) Registers

Register addresses are: RDBR0 0x04:0x004 and RDBR1 0x04:0x204. The reset value is 0x0000.

These are 16-bit read-only (RO) registers. At the end of a data transfer, the data in the shift register is loaded into RDBR. During a DMA receive operation, the data in RDBR is automatically read by the DMA. When RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD=00).

Receive Data Buffer Shadow, SPI (RDBRSx) Registers

Registers' addresses are: RDBRS0 0x04:0x006 and RDBRS1 0x04:0x206. The reset value is 0x0000.

This is a 16-bit read-only shadow register (for the Receive Data Buffer register) provided for use with debugging software. The RDBRSx register is at a different address from RDBR, but its contents are identical to that of RDBR. When RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD=00). No such hardware action occurs when the shadow register is read.

SPI Baud Rate (SPIBAUDx) Registers

Register addresses are: `SPIBAUD0 0x04:0x005` and `SPIBAUD1 0x04:0x205`.

The SPI baud rate register (`SPIBAUD`) 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 the following formula:

SCK Frequency = (Peripheral clock frequency)/(2*`SPIBAUD`)

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the peripheral clock rate (`HCLK`). The reset value is `0x0000`.

SPI DMA Current Pointer (SPIxD_PTR) Registers

Registers' addresses are: `SPI0D_PTR 0x04:0x100` and `SPI1D_PTR 0x04:0x300`.

A Current Chain Pointer register holds the address for the current transfer control block in a chained DMA operation. The reset value is `0x0000`.

SPI DMA Configuration (SPIxD_CFG) Registers

Register addresses are: `SPI0D_CFG 0x04:0x101` and `SPI1D_CFG 0x04:0x301`.

There are five registers which make up the descriptor block for a DMA transfer. The SPI DMA Configuration (`SPIxD_CFG`) register is one of these registers. They are accessible through the DMA bus.

Note: Bit default is 0 unless marked otherwise.

Figure B-22 on page B-69 provides bit descriptions.

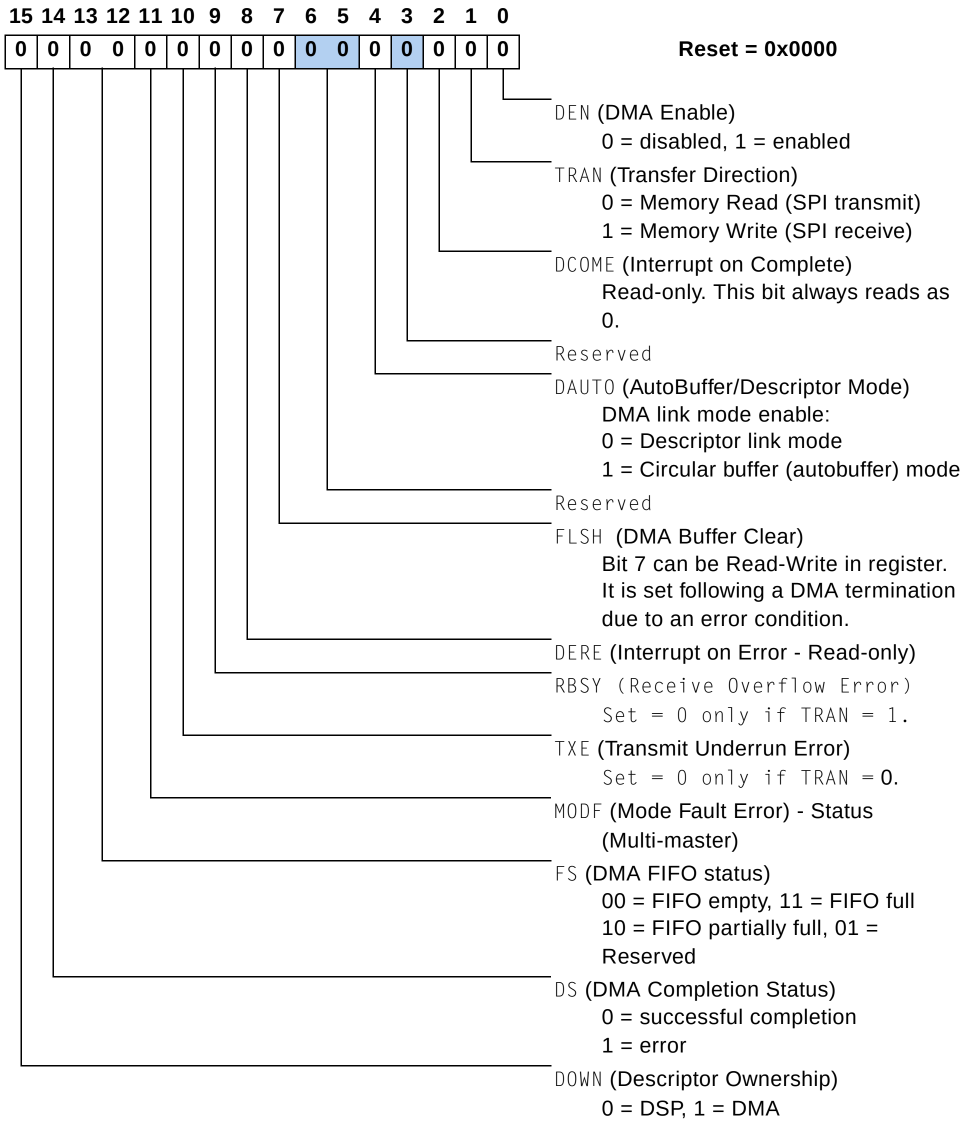

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DEN (DMA Enable)
0 = disabled, 1 = enabled
TRAN (Transfer Direction)
0 = Memory Read (SPI transmit)
1 = Memory Write (SPI receive)
DCOME (Interrupt on Complete)
Read-only. This bit always reads as 0.
Reserved
DAUTO (AutoBuffer/Descriptor Mode)
DMA link mode enable:
0 = Descriptor link mode
1 = Circular buffer (autobuffer) mode
Reserved
FLSH (DMA Buffer Clear)
Bit 7 can be Read-Write in register. It is set following a DMA termination due to an error condition.
DERE (Interrupt on Error - Read-only)
RBSY (Receive Overflow Error)
Set = 0 only if TRAN = 1.
TXE (Transmit Underrun Error)
Set = 0 only if TRAN = 0.
MODF (Mode Fault Error) - Status (Multi-master)
FS (DMA FIFO status)
00 = FIFO empty, 11 = FIFO full
10 = FIFO partially full, 01 = Reserved
DS (DMA Completion Status)
0 = successful completion
1 = error
DOWN (Descriptor Ownership)
0 = DSP, 1 = DMA

Figure B-22. SPI DMA Configuration (SPIxD_CFG) Registers' Bits

SPI DMA Start Page (SPIxD_SRP) Registers

Registers' addresses are: `SPI0D_SRP 0x04:0x102` and `SPI1D_SRP 0x04:0x302`. The 16-bit SPI DMA Start Page (`SPIxD_SPR`) register holds a running pointer to the DMA address that is being accessed and the type of memory space being used.

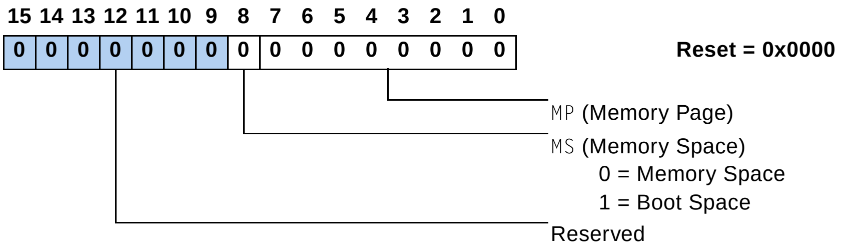


Figure B-23. SPI DMA Start Page (SPIxD_SRP) Registers' Bits

SPI DMA Start Address (SPIxD_SRA) Registers

Registers' addresses are: `SPI0D_SRA 0x04:0x103` and `SPI1D_SRA 0x04:0x303`. The 16-bit SPI DMA Start Address (`SPIxD_SRA`) register holds a running pointer to the DMA address that is being accessed. The reset value is `0x0000`.

SPI DMA Word Count (SPIxD_CNT) Registers

Registers' addresses are: `SPI0D_CNT 0x04:0x104` and `SPI1D_CNT 0x04:0x304`. The 16-bit SPI DMA Word Count (`SPIxD_CNT`) register holds the block word count (the number of remaining words in the transfer). The reset value is `0x0000`.

SPI DMA Next Chain Pointer (SPIxD_CP) Registers

Registers' addresses are: `SPI0D_CP 0x04:0x105` and `SPI1D_CP 0x04:0x305`. The SPI DMA Next Chain Pointer (`SPIxD_CP`) register is used to write the head of descriptor list. A `CPx` register holds the address for the next transfer control block in a chained DMA operation. The reset value is `0x0000`.

SPI DMA Chain Pointer Ready (SPIxD_CPR) Registers

These registers' addresses are: `SPI0D_CPR 0x04:0x106` and `SPI1D_CPR 0x04:0x306`. These 1-bit registers are used to show the descriptor's status. If bit 0 is set to 0, the descriptor block is ready (set). The reset value is `0x0000`.

SPI DMA Interrupt (SPIxD_IRQ) Registers

These registers' addresses are: `SPI0D_IRQ 0x04:0x107` and `SPI1D_IRQ 0x04:0x307`.

These registers are used to indicate the SPI DMA interrupt status.

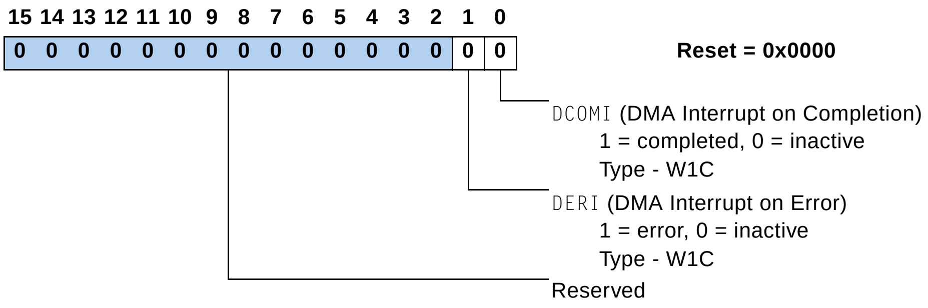


Figure B-24. SPI DMA Interrupt (SPIxD_IRQ) Registers' Bits

UART Registers

The UART peripheral is a full-duplex Universal Asynchronous Receiver / Transmitter that is compatible with the industry standard 16450. The UART is responsible for converting data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, and parity generation options. This UART also contains control interrupt handling hardware. Interrupts may be generated from twelve unique events. The UART registers can be divided into three groups:

- "UART Control Registers" on page B-72
- "UART RX DMA Registers" on page B-80
- "UART TX DMA Registers" on page B-85

All registers are mapped into the I/O page 5. To access them, the I/O Page (IOPG) register must be set to 5.

UART Control Registers

The UART control registers are typically used in I/O mode. To meet the 16450 standard they are all eight bits wide. Also, the two divisor latch registers share their access addresses with others.

Table B-10 on page B-73 summarizes the UART control registers.

Except for the Scratch (SCR) register and the four Modem Status signals (DCD, RI, DSR, and CTS), all bits are predefined after reset. NINT, TEMT, and THRE are set; DLL resets to 0x01; and all remaining bits are cleared.

This section includes the following topics:

- "Transmit Hold (THR) Register" on page B-74
- "Receive Buffer (RBR) Register" on page B-74

Table B-10. UART Control Registers Summary

Register	Addr	DLAB	Access	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
THR	0	0	W	Transmit Holding Register							
RBR	0	0	R	Receive Buffer Register							
DLL	0	1	R/W	Divisor Latch Low-Byte							
IER	1	0	R/W	0	0	0	0	EDSS I	ELSI	ETB EI	ERB FI
DLH	1	1	R/W	Divisor Latch High-Byte							
IIR	2	x	R	0	0	0	0	0	STATUS		NIN T
LCR	3	x	R/W	DLA B	BRK	SP	EPS	PEN	STB	WLS	
MCR	4	x	R/W	0	0	0	LOO P	OUT 2	OU T1	RTS	DTR
LSR	5	x	R	0	TEM T	THR E	BI	FE	PE	OE	DR
MSR	6	x	R	DC D	RI	DSR	CTS	DDC D	TER I	DDS R	DCT S
SCR	7	x	R/W	Scratch Register							

- "Interrupt Enable (IER) Register" on page B-75
- "UART Divisor Latch Registers (DLL and DLH)" on page B-76
- "Interrupt Identification (IIR) Register" on page B-77
- "Line Control (LCR) Register" on page B-77
- "Modem Control (MCR) Register" on page B-77
- "Line Status (LSR) Register" on page B-78

- "Modem Status (MSR) Register" on page B-78
- "Scratch (SCR) Register" on page B-80

Transmit Hold (THR) Register

The `THR` register address is `0x05:0x000`. The register is write-only. To access `THR`, the divisor latch access bit (`DLAB`) in the Line Control (`LCR`) register must be cleared.

The transmit operation is initiated by writing to the Transmit Hold register (`THR`). After a synchronization delay, the data is moved to the Transmit Shift (`TSR`) register where it will shifted out at a baud (bit) rate equal to `HCLK / (16 * DIVISOR)` with start, stop, and parity bits appended as required. All data words begin with a low start bit. The transfer of the `THR` to the `TSR` causes the transmit hold register empty (`THRE`) status flag to be set. Data is transmitted and received least significant bit first (that is, `TSR` bit 0). The reset value is `0x00`.

Receive Buffer (RBR) Register

The `RSR` register address is `0x05:0x000`. The register is read-only. To access `RBR`, the divisor latch access bit (`DLAB`) in the Line Control (`LCR`) register must be cleared.

The receive operation uses the same data format as the transmit configuration, except for the number of stop bits which is always one. After detection of the start bit, the received word is shifted in the Receive Shift register `(RSR)`. After the appropriate number of bits (including stop bits) are received the data and any status is updated and the `RSR` is transferred to the Receive Buffer `(RBR)` register. The Receive Buffer register's full (`DR`) status flag is updated upon the transfer of the received word to this buffer and the appropriate synchronization delay. The reset value is `0x00`.

Interrupt Enable (IER) Register

The IER register address is 0x05:0x001. To access IER the divisor latch access bit (DLAB) in the Line Control (LCR) register must be cleared. The IER register applies to the I/O mode only. There, four different Interrupt sources ar ORed to share a single IRQ channel. In I/O mode, UART Interrupts can be generated when either:

- Data is ready in the RBR register
- Transmit data is moved from the THR to the TSR register
- Received data is misaligned (parity and so on)
- The modem status has changed

These four interrupt sources can be individually enabled or masked by the Interrupt Enable (IER) register. Interrupts are also masked by the IMASK register and the global interrupt enable bit. An Interrupt Service Routine should read the IIR register to determine the interrupt source.

Because the four modem status signals are tied low on the DSP, the modem status interrupt might not be generated as long as the LOOP bit in Modem Control (MC) register is not set.

The IER register resets to 0x00.

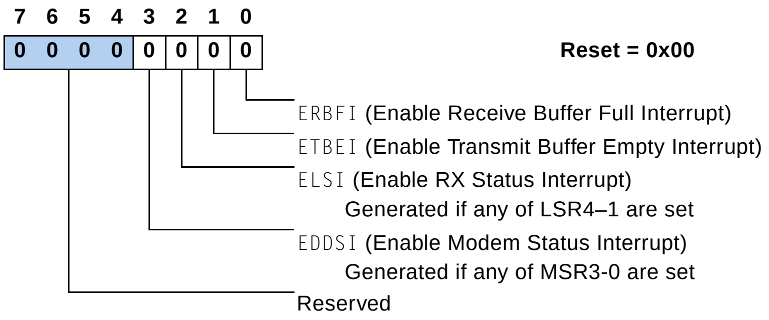

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
Reset = 0x00
ERBFI (Enable Receive Buffer Full Interrupt)
ETBEI (Enable Transmit Buffer Empty Interrupt)
ELSI (Enable RX Status Interrupt)
Generated if any of LSR4–1 are set
EDDSI (Enable Modem Status Interrupt)
Generated if any of MSR3-0 are set
Reserved

Figure B-25. UART Interrupt Enable (IER) Register Bits

UART Divisor Latch Registers (DLL and DLH)

The register addresses are: `0x05:0x000` for the Divisor Latch (Low-Byte) register (`DLL`) and `0x05:0x001` for the Divisor Latch (High-Byte) register (`DLH`).

To access these registers, the divisor latch access bit (`DLAB`) of the Line Control (`LCR`) register must be set. `DLL` resets to `0x01`, and `DLH` resets to `0x00`.

The `DLL` and `DLH` registers form a 16-bit divisor. These read/write registers are accessed when `DLAB = 1`. Keep in mind that the Baud Clock is divided by an additional `16` as per a protocol such as:

```
BAUD RATE = HCLK / (16 * DL)
```

where:

`DL` = 65536 when `DLL = DLH = 0`

Table B-11. Example with HCLK =80 MHz

Baud rate	DL	Actual	% Error
600	8333	600.02	-0.004
1200	4167	1199.90	0.008
2400	2083	2400.38	-0.016
4800	1042	4798.46	0.032
9600	521	9596.93	0.032
19200	260	19230.77	-0.160
38400	130	38461.54	-0.160
57600	87	57471.26	0.223
115200	43	116279.07	-0.937

Interrupt Identification (IIR) Register

This register address is `0x05:0x002`. The reset value is `0x01`. In I/O mode, a UART interrupt service routine should read the `IIR` register to determine the exact interrupt source, whenever more than one source is enabled by the Interrupt Enable (`IER`) register.

When cleared, bit 0 (`NINT`) signals that an interrupt is pending. Then, bits 2 and 1 (`STATUS`) indicate the highest priority pending interrupt as follows:

Table B-12. UART Interrupt Identification STATUS Field

Status	Priority	Source	Cleared When...
0	4	Modem status	Read MSR
1	3	THR Empty	Write THR or read IIR when priority = 3
2	2	RX Data Ready	Read RBR
3	1	RX Status	Read LSR

Line Control (LCR) Register

The `LCR` register address is `0x05:0x003`. The reset value is 0 for all bits. The `LCR` register formats the asynchronous serial bit stream as specified below:

Modem Control (MCR) Register

The `MCR` register address is `0x05:0x004`. The reset value is 0 for all bits.

The four modem control bits (`DTR`, `RTS`, `OUT1`, and `OUT2`) do not affect the DSP as long as the `LOOP` bit is not set. The `MCR` register is a read-write register that guarantees a certain level of software legacy support.

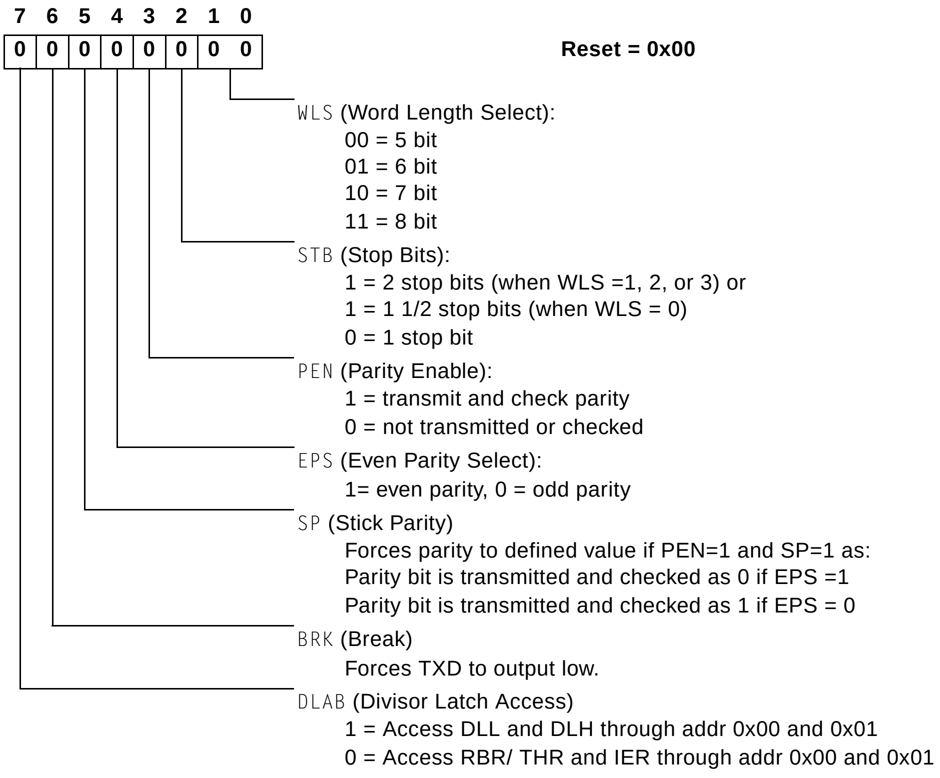

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
Reset = 0x00
WLS (Word Length Select):
00 = 5 bit
01 = 6 bit
10 = 7 bit
11 = 8 bit
STB (Stop Bits):
1 = 2 stop bits (when WLS =1, 2, or 3) or
1 = 1 1/2 stop bits (when WLS = 0)
0 = 1 stop bit
PEN (Parity Enable):
1 = transmit and check parity
0 = not transmitted or checked
EPS (Even Parity Select):
1= even parity, 0 = odd parity
SP (Stick Parity)
Forces parity to defined value if PEN=1 and SP=1 as:
Parity bit is transmitted and checked as 0 if EPS =1
Parity bit is transmitted and checked as 1 if EPS = 0
BRK (Break)
Forces TXD to output low.
DLAB (Divisor Latch Access)
1 = Access DLL and DLH through addr 0x00 and 0x01
0 = Access RBR/ THR and IER through addr 0x00 and 0x01

Figure B-26. UART Line Control (LCR) Register Bits

Line Status (LSR) Register

The `LSR` register address is `0x05:0x005`. The reset value is 0x60.

Modem Status (MSR) Register

The `MSR` register address is `0x05:0x006`. The reset value is 0 for all bits, except bits 7–4 which mirror input signals.

ⓘ Note that on the DSP the Modem Status Signals (`CTS`, `DSR`, `RI`, and `DCD`) are all tied low as long as the `LOOP` bit in the Modem Control (`MCR`) register is not set.

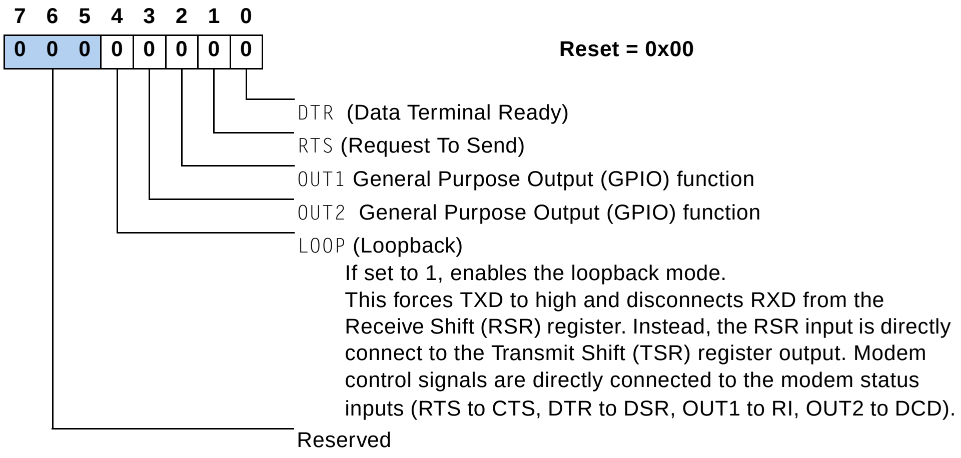

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
Reset = 0x00
DTR (Data Terminal Ready)
RTS (Request To Send)
OUT1 General Purpose Output (GPIO) function
OUT2 General Purpose Output (GPIO) function
LOOP (Loopback)
If set to 1, enables the loopback mode.
This forces TXD to high and disconnects RXD from the Receive Shift (RSR) register. Instead, the RSR input is directly connect to the Transmit Shift (TSR) register output. Modem control signals are directly connected to the modem status inputs (RTS to CTS, DTR to DSR, OUT1 to RI, OUT2 to DCD).
Reserved

Figure B-27. UART Modem Control (MCR) Register Bits

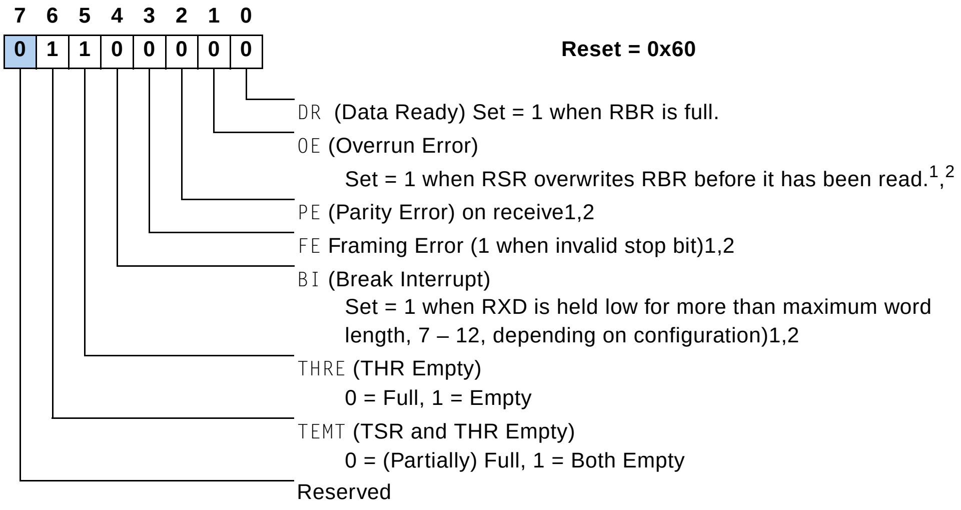

7 6 5 4 3 2 1 0
0 1 1 0 0 0 0 0
Reset = 0x60
DR (Data Ready) Set = 1 when RBR is full.
OE (Overrun Error)
Set = 1 when RSR overwrites RBR before it has been read.1,2
PE (Parity Error) on receive1,2
FE Framing Error (1 when invalid stop bit)1,2
BI (Break Interrupt)
Set = 1 when RXD is held low for more than maximum word length, 7 – 12, depending on configuration)1,2
THRE (THR Empty)
0 = Full, 1 = Empty
TEMT (TSR and THR Empty)
0 = (Partially) Full, 1 = Both Empty
Reserved

Figure B-28. UART Line Status (LSR) Register

1 Bit is cleared when program reads LSR.

2 Error condition request the RX Status Interrupt when enabled.

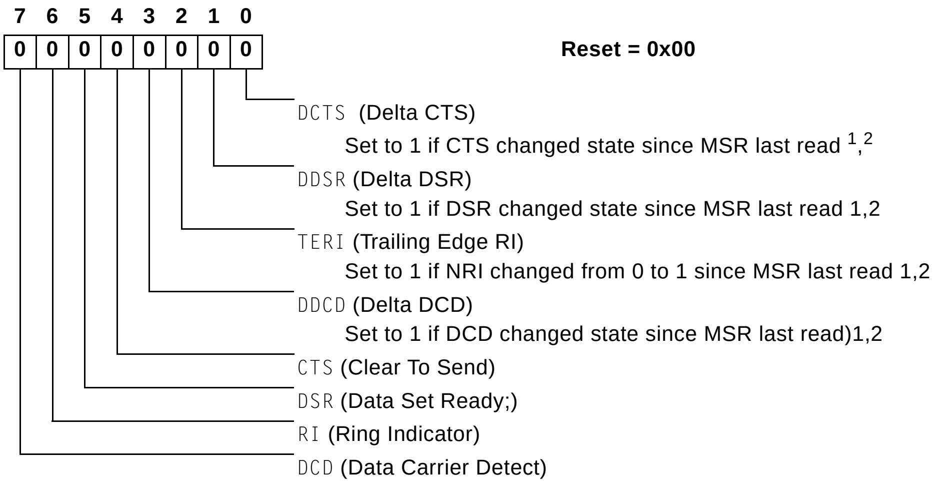

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
Reset = 0x00
DCTS (Delta CTS)
Set to 1 if CTS changed state since MSR last read 1,2
DDSR (Delta DSR)
Set to 1 if DSR changed state since MSR last read 1,2
TERI (Trailing Edge RI)
Set to 1 if NRI changed from 0 to 1 since MSR last read 1,2
DDCD (Delta DCD)
Set to 1 if DCD changed state since MSR last read)1,2
CTS (Clear To Send)
DSR (Data Set Ready;)
RI (Ring Indicator)
DCD (Data Carrier Detect)

Figure B-29. UART Modem Status (MSR) Register

1 Bit is cleared when program reads MSR.
2 Modem status change requests the Modem Status Interrupt when enabled.

Scratch (SCR) Register

This register address is `0x05:0x007`. It is not changed by reset. The 8-bit scratch register is used for general-purpose data storage only. It does not control the UART hardware in any way.

UART RX DMA Registers

This section includes the following topics:

- "UART DMA Receive Pointer (UARDR_PTR) Register" on page B-81
- "UART Receive DMA Configuration (UARDR_CFG) Register" on page B-81
- "UART Receive DMA Start Page (UARDR_SRP) Register" on page B-83

- "UART Receive DMA Start Address (UARDR_SRA) Register" on page B-83
- "UART Receive DMA Count (UARDR_CNT) Register" on page B-84
- "UART Receive DMA Chain Pointer (UARDR_CP) Register" on page B-84
- "UART Receive DMA Chain Pointer Ready (UARDR_CPR) Register" on page B-84
- "UART Transmit DMA Interrupt (UARDT_IRQ) Register" on page B-88

UART DMA Receive Pointer (UARDR_PTR) Register

This register address is `0x05:0x100`. This 16-bit read-only register holds the pointer to the current descriptor block for UART receive DMA operation. The reset value is 0 for all bits

UART Receive DMA Configuration (UARDR_CFG) Register

This register address is `0x05:0x101`. This register, which is normally read-only, becomes writable when the autobuffer mode is enabled for the selected bits.

During UART initialization, the head address of the first DMA descriptor block must be written into the Receive DMA Chain Pointer register. Then, the DMA enable bit is set in the Receive DMA Configuration registers.

The DMA Configuration register maintains real-time DMA buffer status. When DMA is not enabled, the UART generates a receive interrupt every time the Receive Buffer (`RBR`) register holds new data and the date ready bit (`DR`) is set.

Refer to Figure on page B-82 for bit descriptions.

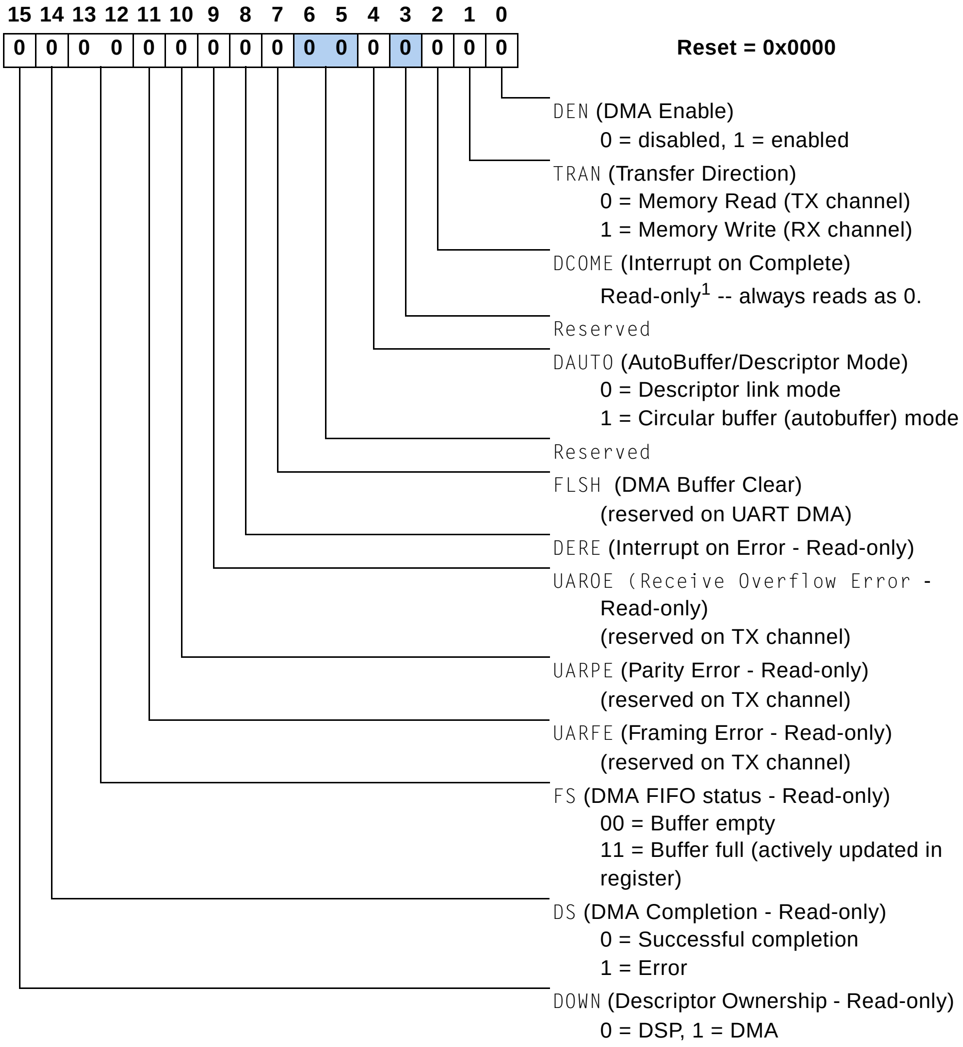

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DEN (DMA Enable)
0 = disabled, 1 = enabled
TRAN (Transfer Direction)
0 = Memory Read (TX channel)
1 = Memory Write (RX channel)
DCOME (Interrupt on Complete)
Read-only[1] -- always reads as 0.
Reserved
DAUTO (AutoBuffer/Descriptor Mode)
0 = Descriptor link mode
1 = Circular buffer (autobuffer) mode
Reserved
FLSH (DMA Buffer Clear)
(reserved on UART DMA)
DERE (Interrupt on Error - Read-only)
UAROE (Receive Overflow Error - Read-only)
(reserved on TX channel)
UARPE (Parity Error - Read-only)
(reserved on TX channel)
UARFE (Framing Error - Read-only)
(reserved on TX channel)
FS (DMA FIFO status - Read-only)
00 = Buffer empty
11 = Buffer full (actively updated in register)
DS (DMA Completion - Read-only)
0 = Successful completion
1 = Error
DOWN (Descriptor Ownership - Read-only)
0 = DSP, 1 = DMA

Figure B-30. UART Receive DMA Configuration (UARDR_CFG) Register Bits

1 Writable when the AUTOBUFFER mode is enabled.

UART Receive DMA Start Page (UARDR_SRP) Register

The register address is 0x05:0x102. This register, which is normally read-only, becomes writable when the autobuffer mode is enabled, i.e. DAUTO = 1.

The 9-bit UART Receive DMA Start Page (UARDR_SRP) register holds a pointer to the DMA address that is being accessed.

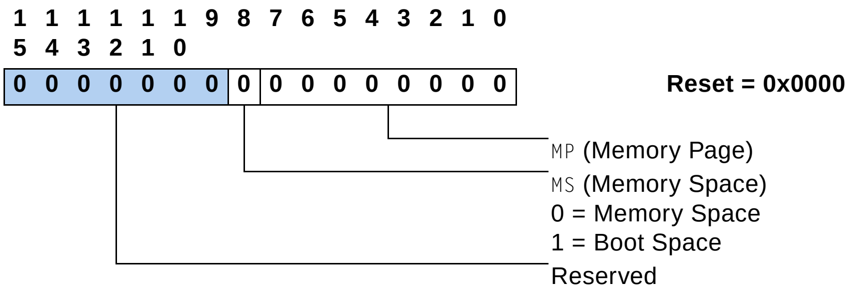


Table B-13. UART Receive DMA Start Page (UARDR_SRP) Register Bits

UART Receive DMA Start Address (UARDR_SRA) Register

This register address is 0x05:0x103. The 16-bit UART Receive DMA Start Address (UARDR_SRA) register holds a pointer to the DMA address that is being accessed. It is a read-only register (can be written in the autobuffer mode; that is, DAUTO = 1). The reset value is 0x0000.

UART Receive DMA Count (UARDR_CNT) Register

This register address is 0x05:0x104. The 16-bit UART Receive DMA Count (UARDR_CNT) register holds the word count for a DMA transfer. It is a read-only register (can be written if autobuffer mode is enabled; that is, DAUTO = 1). The reset value is 0x0000.

UART Receive DMA Chain Pointer (UARDR_CP) Register

This register address is `0x05:0x105`. During initialization, the head address of the next DMA descriptor block must be written into the 16-bit DMA Chain (Next Descriptor) Pointer register. Then, the DMA enable bit is set in the Receive DMA Configuration register. The descriptor ready bit should be set in software after each descriptor is written to internal memory to start the DMA. The reset value is `0x0000`.

UART Receive DMA Chain Pointer Ready (UARDR_CPR) Register

This register address is `0x05:0x106`. This 1-bit register is used to show the descriptor's status. If Bit 0 is set to 0, the descriptor block is ready (set). The reset value is `0x0000`.

UART Receive DMA Interrupt Register (UARDR_IRQ) Register

This register address is `0x05:0x107`. This read-write register is used to indicate the UART Receive DMA interrupt status. 1s are to be written to clear this register. Interrupt service routines write 0x01 to this register to clear the pending receive interrupt request.

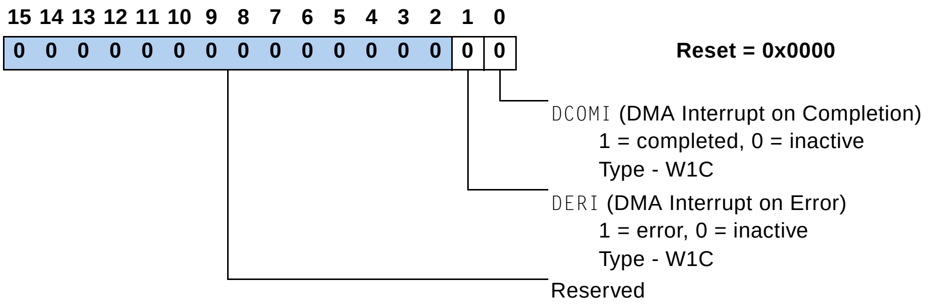


Figure B-31. UART Receive DMA Interrupt (UARDR_IRQ) Register Bits

UART TX DMA Registers

This section includes the following topics:

UART Transmit DMA Pointer (UARDT_PTR) Register

This register address is `0x05:0x180`. This 16-bit read-only register holds the pointer to the current descriptor block for UART transmit DMA operation. The reset value is `0x0000`.

UART Transmit DMA Configuration (UARDT_CFG) Register

This register address is 0x05:0x181. This register, which is normally read-only, becomes writable when the autobuffer mode is enabled for the selected bits. The reset value is 0x0000.

During UART initialization, the head address of the first DMA descriptor block must be written into the Transmit DMA Descriptor Pointer register. Then, the DMA enable bit is set in the Transmit DMA Configuration registers. The DMA Configuration register maintains real-time DMA buffer status. When DMA is not enabled, the UART generates a transmit interrupt every time the content of the THR register is moved to the TSR registers and the THR empty bit (THRE) is set.

For information on the bits in this register (which are the same as the UARDR_CFG register), see Figure B-30 on page B-82.

UART Transmit DMA Start Page (UARDT_SRP) Register

This register address is 0x05:0x182. The register which is normally read only becomes writable when the autobuffer mode is enabled.

The 9-bit DMA Start Page register holds a pointer to the DMA page that is being accessed. Bits 0–7 determine the receive DMA start page address. Bit 8 specifies the Space type: 0 = memory space, 1 = boot space. See Figure B-13 on page B-83 for bit illustration. The reset value is 0x0000.

UART Transmit DMA Start Address (UARDT_SRA) Register

This register address is 0x05:0x183. The 16-bit Receive DMA Start Address register holds a pointer to the DMA address that is being accessed. It is a read-only register (can be written in the autobuffer mode; that is, DAUTO = 1). The reset value is 0x0000.

UART Transmit DMA Count (UARDT_CNT) Register

This register address is `0x05:0x184`. The 16-bit DMA Count register holds the word count for a DMA transfer. It is a read-only register (can be written in autobuffer mode; that is, `DAUTO = 1`). The reset value is `0x0000`.

UART Transmit DMA Chain Pointer (UARDT_CP) Register

This register address is `0x05:0x185`. During initialization, the head address of the next DMA descriptor block must be written into the 16-bit `UARDT_CP` (Next Descriptor) register. Then, the DMA enable bit is set in the Receive DMA Configuration register. The reset value is `0x0000`.

UART Transmit DMA Chain Pointer Ready (UARDT_CPR) Register

This register address is `0x05:0x186`. This 1-bit register is used to show the descriptor's status. If bit 0 is set to 0, the descriptor block is ready (set). The descriptor ready bit should be set in software after each descriptor is written to internal memory to start the DMA. The reset value is `0x0000`.

UART Transmit DMA Interrupt (UARDT_IRQ) Register

This register address is `0x05:0x187`. This read-write register is used to indicate the UART transmit DMA interrupt status. 1s are to be written to clear this register. Interrupt service routines write 0x01 to this register to clear the pending transmit interrupt request.

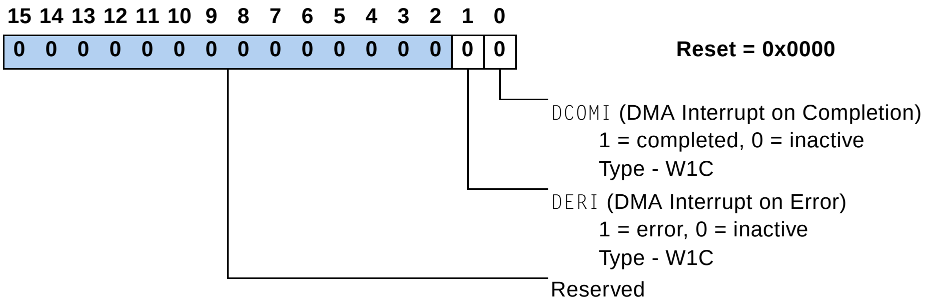


Figure B-32. UART Transmit DMA Interrupt (UARDT_IRQ) Register Bits

Timer Registers

This section includes the following topics:

Overview

The ADSP-2191 timer peripheral module provides general-purpose timer functionality. It consists of three identical timer units.

To provide the required functionality, each timer has seven 16-bit memory-mapped registers. Six of these registers are paired to achieve 32-bit precision and appropriate range. Entity pair values are not accessed concurrently over the 16-bit peripheral bus, requiring a mechanism to insure coherency of the register pair values. For example, the user must disable the timer to ensure high-low register pair coherency for the timer counter.

Each timer provides four registers:

- Config 15–0 – Configuration register
- Width 31–0 – Pulse Width register
- Period 31–0 – Pulse Period register
- Counter 31–0 – Timer Counter

A common status register, global status 15–0 is also provided, requiring only a single read to determine the status. Status bits are "sticky" and require a "write-one" to clear operation.

Timer Global Status and Control (T_GSRx) Registers

The three global status registers' addresses are:

```
T_GSR0 0x05:0x200
T_GSR1 0x05:0x208
T_GSR2 0x05:0x210
```

Each timer has a common status register, Status 15–0, requiring only a single read to determine the status. Status bits are "sticky" and require a "write-one" to clear operation. During a status register read access, all reserved or unused bits return a zero. The reset state is `0x0000`.

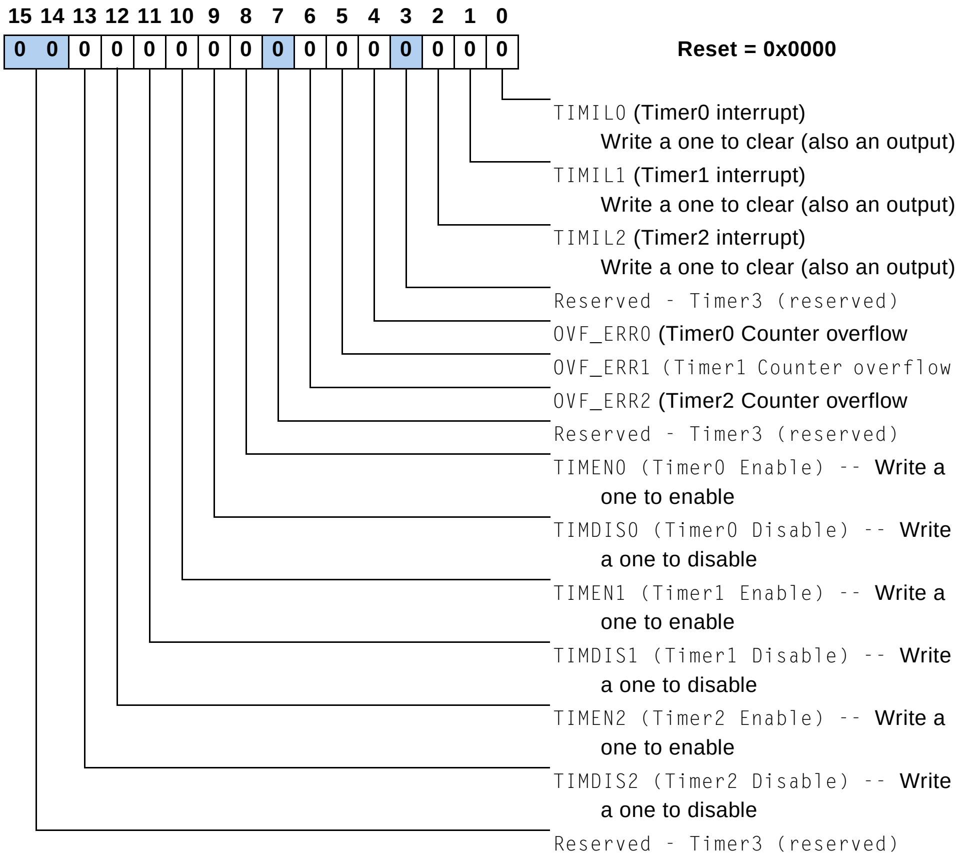

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TIMIL0 (Timer0 interrupt) Write a one to clear (also an output)
TIMIL1 (Timer1 interrupt) Write a one to clear (also an output)
TIMIL2 (Timer2 interrupt) Write a one to clear (also an output)
Reserved - Timer3 (reserved)
OVF_ERR0 (Timer0 Counter overflow
OVF_ERR1 (Timer1 Counter overflow
OVF_ERR2 (Timer2 Counter overflow
Reserved - Timer3 (reserved)
TIMEN0 (Timer0 Enable) -- Write a one to enable
TIMDIS0 (Timer0 Disable) -- Write a one to disable
TIMEN1 (Timer1 Enable) -- Write a one to enable
TIMDIS1 (Timer1 Disable) -- Write a one to disable
TIMEN2 (Timer2 Enable) -- Write a one to enable
TIMDIS2 (Timer2 Disable) -- Write a one to disable
Reserved - Timer3 (reserved)

Figure B-33. Timer Global Status and Sticky (T_GSRx) Registers' Bits

Each timer generates a unique DSP interrupt request signal, TMR_IRQ. A common status register latches these interrupts so you can determine the interrupt source without reference to the unique interrupt signal. Interrupt bits are “sticky” and must be cleared to assure that the interrupt is not re-issued.

Each timer is provided with its own “sticky” Status register TIMENx bit. To enable or disable an individual timer, set or clear the TIMEN bit. For example, writing a one to bit 8 sets the TIMEN0 bit; writing a one to bit 9 clears it. Writing a one to both bit 8 and bit 9 clears TIMEN0. Reading the status register returns the TIMEN0 state on both bit 8 and bit 9. The remaining TIMENx bits operate similarly using bit 10 and bit 11 for Timer1, and bit 12 and bit 13 for Timer2.

Timer Configuration (T_CFGRx) Registers

The three T_CFGR registers' addresses are:

```
T_CFGR0 0x05:0x201
T_CFGR1 0x05:0x209
T_CFGR2 0x05:0x211
```

All Timer clocks are gated "OFF" when the specific timer's configuration register is set to zero at system reset or subsequently reset by the user. Figure B-34 on page B-92 provides bit descriptions.

Timer Counter Low Word (T_CNTLx) and Timer Counter High Word (T_CNTHx) Registers

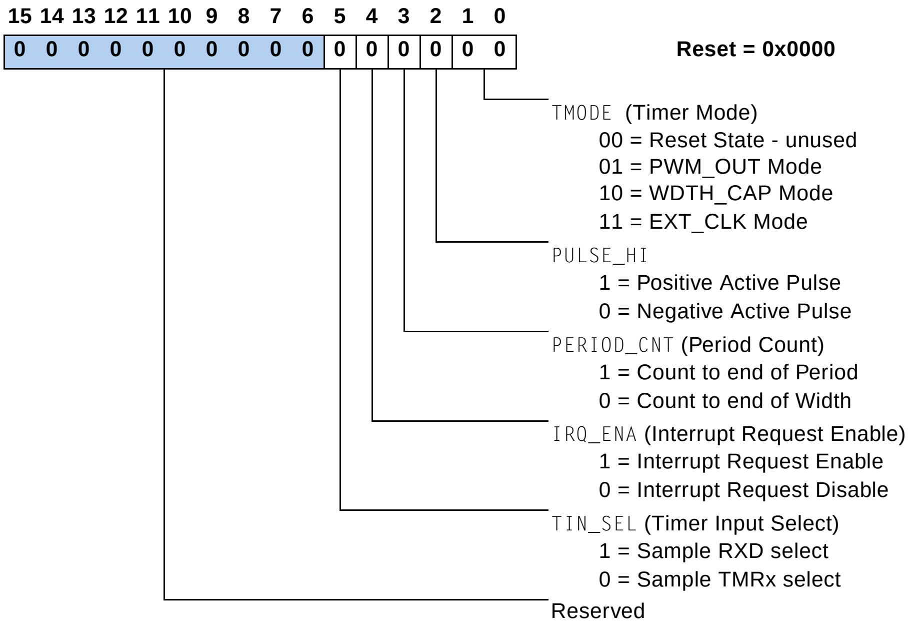
The T_CNTLx registers' addresses are:

```
T_CNTL0 0x05:0x202
T_CNTL1 0x05:0x20A
T_CNTL2 0x05:0x212
```

The T_CNTHx registers' addresses are:

```
T_CNTH0 0x05:0x203
T_CNTH1 0x05:0x20B
T_CNTH2 0x05:0x213
```

These 16-bit memory-mapped registers are paired (15:0 as low and 31:16 as high) to achieve 32-bit precision and appropriate range.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TMODE (Timer Mode)
00 = Reset State - unused
01 = PWM_OUT Mode
10 = WDTH_CAP Mode
11 = EXT_CLK Mode
PULSE_HI
1 = Positive Active Pulse
0 = Negative Active Pulse
PERIOD_CNT (Period Count)
1 = Count to end of Period
0 = Count to end of Width
IRQ_ENA (Interrupt Request Enable)
1 = Interrupt Request Enable
0 = Interrupt Request Disable
TIN_SEL (Timer Input Select)
1 = Sample RXD select
0 = Sample TMRx select
Reserved

Figure B-34. Timer Configuration (T_CFGRx) Register Bits

When disabled, the timer counter retains its state. When enabled again, the timer counter is re-initialized from the period/width registers based on configuration and mode.

The timer counter value cannot be set directly by the software. It can be set indirectly by initializing the period or width values in the appropriate mode. The counter should only be read when the respective timer is disabled. This prevents erroneous data from being returned.

In the EXT_CLK Mode, the TMRx (or RXD in Auto-baud mode) pin is used to clock the timer counter. The counter is initialized with the period value and counts until the period expires. In the EXT_CLK mode, the timer counter can operate at a maximum frequency of 25 MHz. This limitation results from a synchronization/latency trade off in the counter control logic.

If the 32-bit counter were clocked by a 10 MHz external clock, it is possible to achieve a maximum timer counter period of $(2^{32}-1)$ * 100ns.

Timer Period Low Word (T_PRDLx) and Timer Period High Word (T_PRDHx) Registers

The `T_PRDLx` registers' addresses are:

```
T_PRDL0 0x05:0x204
T_PRDL1 0x05:0x20C
T_PRDL2 0x05:0x214
```

The `T_PRDHx` registers' addresses are:

```
T_PRDH0 0x05:0x205
T_PRDH1 0x05:0x20D
T_PRDH2 0x05:0x215
```

These 16-bit memory-mapped registers are paired (`15:0` as low and `31:16` as high) to achieve 32-bit precision and appropriate range.

Once a timer is enabled and running, when the DSP writes new values to the timer period and timer pulse width registers, 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).

- During the *Pulse Width Modulation* (`PWM_OUT`), the period value is written into the timer period registers. Both period and width register values must be updated "on the fly" since the period and width (duty cycle) change simultaneously. To ensure the period and width value concurrency, a 32-bit period buffer and a 32-bit width buffer are used.
- The high-low period values are updated first if necessary. Once the period value has been updated, it is necessary to update the high-word width value followed by the low-word width value.

Updating the low-word width value is what actually transfers the period and width values to their respective Buffers. This permits low-only width value updates for low-resolution situations while maintaining high-low value coherency.

- If the period value is updated, the low-word width value must be updated as well. This mechanism permits width-only updates while maintaining period and width value coherency. When the low-word width value is updated, the Timer simultaneously updates the period and width buffers on the next clock cycle.
- During the *Pulse Width and Period Capture* (`WDTH_CAP`) mode, the period values are captured at the appropriate time. Since both the period and width registers are read-only in this mode, the existing 32-bit period and width buffers are used.
- During the `EXT_CLK` mode, the period register is write-only. Therefore, the period buffer is used in this mode to insure high/low period value coherency.

Timer Width Low Word (T_WLRx) Register and TImer Width High Word (T_WHRx) Register

The `T_WLRx` registers' addresses are:

```
T_WLR0 0x05:0x206
T_WLR1 0x05:0x20E
T_WLR2 0x05:0x216
```

The `T_WHRx` registers' addresses are:

```
T_WHR0 0x05:0x207
T_WHR1 0x05:0x20F
T_WHR2 0x05:0x217
```

These 16-bit memory-mapped registers are paired (15:0 as low and 31:16 as high) to achieve 32-bit precision and appropriate range.

- During the *Pulse Width Modulation* (`PWM_OUT`), the width value is written into the timer width registers. Both width and period register values must be updated "on the fly" since the period and width (duty cycle) change simultaneously. To ensure period and width value concurrency, a 32-bit period buffer and a 32-bit width buffer are used.

  The high-low period values are updated first if necessary. Once the period value has been updated, it is necessary to update the high-word width value followed by the low-word width value. Updating the low-word width value is what actually transfers the period and width values to their respective buffers. This permits low-only width value updates for low-resolution situations while maintaining high-low value coherency.

  If the period value is updated, the low-word width value must be updated as well. This mechanism permits width-only updates while maintaining period and width value coherency. When the low-word width value is updated, the timer simultaneously updates the period and width buffers on the next clock cycle.

- During the *Pulse Width and Period Capture* (`WDTH_CAP`) mode, both the period and width values are captured at the appropriate time. Since both the width and period registers are read-only in this mode, the existing 32-bit period and width buffers are used.

- During the `EXT_CLK` mode, the width register is unused.

Programmable Flag Registers

The ADSP-2191 DSP supports sixteen bi-directional programmable flag (or general-purpose) I/O pins, PF15:0. Each pin, configurable via the Flag Direction (DIR) register, is an input or an output. During chip hardware reset, this pin group is used to define clock control mode parameters. These parameters are configured by strapping each pin either high or low via a weak pull-up or pull-down resistor or by driving the pin with the clock configuration while the RESET pin is asserted.

This section includes the following topics:

- "Direction for Flags (DIR) Register" on page B-97
- "Flag (PFx) Interrupt Registers: Flag Clear (FLAGC) and Flag Set (FLAGS)" on page B-97
- "Flag (PFx) Interrupt Mask Registers" on page B-97
- "Flag Source Polarity (FSPR) Register" on page B-98
- "Flag Source Sensitivity (FSSR) Register" on page B-99
- "Flag Sensitivity Both Edges (FSBER) Register" on page B-99

Direction for Flags (DIR) Register

This register's address is: 0x06:0x000. This is a 16-bit read-write register. Each bit position corresponds to a PFx pin. A logic one configures a PFx pin to be an output, driving the state contained in the Peripheral Flag Direction register. A logic zero configures a PFx pin to be an input.

The DIR register can be used to set or clear the output state associated with each output PFx and to set or clear the latched interrupt state captured from each input PFx. This register is initialized with "zeros".

Flag (PFx) Interrupt Registers: Flag Clear (FLAGC) and Flag Set (FLAGS)

The Flag Clear (`FLAGC`) register's addresses is `0x06:0x002` and the Flag Set (`FLAGS`) register's addresses is `0x06:0x003`. Both registers can initialized with input signals.

The Flag Interrupt register is a write-one-to-clear register. Since a level sensitive interrupt is generated to the core, an interrupting flag must have its latch bit cleared prior to returning from the ISR or prior to unmasking to prevent the core from continually responding to the same interrupt.

Flag (PFx) Interrupt Mask Registers

The Flag Interrupt Mask registers (`MASKAC`, `MASKAS`, `MASKBC`, and `MASKBS`) are implemented as complementary pairs of write-one-to-clear and write-one-to-set registers. This provides the ability to enable or disable a `PFx` to act as a DSP interrupt without requiring read-modify-write accesses. These registers are:

- PF Interrupt Flag Mask A Clear (`MASKAC`) register -- `0x06:0x004`
- PF Interrupt Flag Mask A Set (`MASKAS`) register -- `0x06:0x005`
- PF Interrupt Flag Mask B Clear (`MASKBC`) register -- `0x06:0x006`
- PF Interrupt Flag Mask B Set (`MASKBS`) register -- `0x06:0x007`

All sixteen of the `PFx` pins, when configured as inputs, can be individually configured to provide user interrupts. "Interrupt on Input" bits (15–0) in the Flag Interrupt Mask register enable this feature for each pin. All registers are initialized with "zeros".

Two sets of mask registers exist. Setting a bit in the Flag Interrupt Mask A register enables the corresponding bit in the PF Direction (`DIR`) register to interrupt the DSP core when configured as either an input (hardware interrupt) or an output (software interrupt) and when set. The interrupt A line is driven with a logical OR of all masked bits.

Similarly, the Flag Interrupt Mask B register contents are logically ANDed with the contents of the Flag Direction (`DIR`) register. The logic OR of the result is driven onto the interrupt B line.

Flag Source Polarity (FSPR) Register

This register's address is: `0x06:0x008`. This register is initialized with "zeros".

Writing a "0" to a bit of the `FSPR` register configures the corresponding flag pin as an active high input signal. Writing a "1" configures the corresponding flag pin as an active low input signal. The 16 bits of the `FSPR` register correspond to the 16 available flag pins of the ADSP-2191.

When configured as an input, the input signal could be programmed to set the `FLAG` in either level-sensitive or edge-sensitive interrupt mode. Input signal sensitivity is defined in the `FSSR` register.

Flag Source Sensitivity (FSSR) Register

This register's address is `0x06:0x00A`. This register is initialized with "zeros".

Writing a "0" to a bit of the `FSSR` register configures the corresponding flag pin as a level sensitive input. Writing a "1" configures the corresponding flag pin as an edge sensitive input. The 16 bits of the `FSSR` register correspond to the 16 available flag pins of the ADSP-2191.

Flag Sensitivity Both Edges (FSBER) Register

This register's address is `0x06:0x00C`. This register is initialized with "zeros".

Writing a "0" to a bit of the `FSBER` register configures the corresponding flag pin for rising-edge or falling-edge sensitivity (as determined by the value of the corresponding bit of the `FSPR` register). Writing a "1" configures the corresponding flag pin for both-edges sensitivity. The 16 bits of the `FSBER` register correspond to the 16 available flag pins of the ADSP-2191.

External Memory Interface Registers

The External Memory Interface (EMI) peripheral provides an asynchronous parallel data interface to the outside world for ADSP-2191 core based devices. The EMI supports instruction and data transfers from the core to external memory space and boot space. The EMI function is to move 8, 16, or 24 bit data between the core and its peripherals and off-chip memory devices.

This section includes the following topics:

- "External Memory Interface Control/Status (E_STAT) Register" on page B-100
- "External Memory Interface Control (EMICTL) Register" on page B-100
- "Boot Memory Select Control (BMSCTL) Register" on page B-102
- "Memory Select Control (MSxCTL) Registers" on page B-104
- "I/O Memory Select Control (IOMSCTL) Registers" on page B-105

- "External Port Status (EMISTAT) Register" on page B-105
- "Memory Page (MEMPGx) Registers" on page B-105

External Memory Interface Control/Status (E_STAT) Register

This register address is `0x00:0x080`.

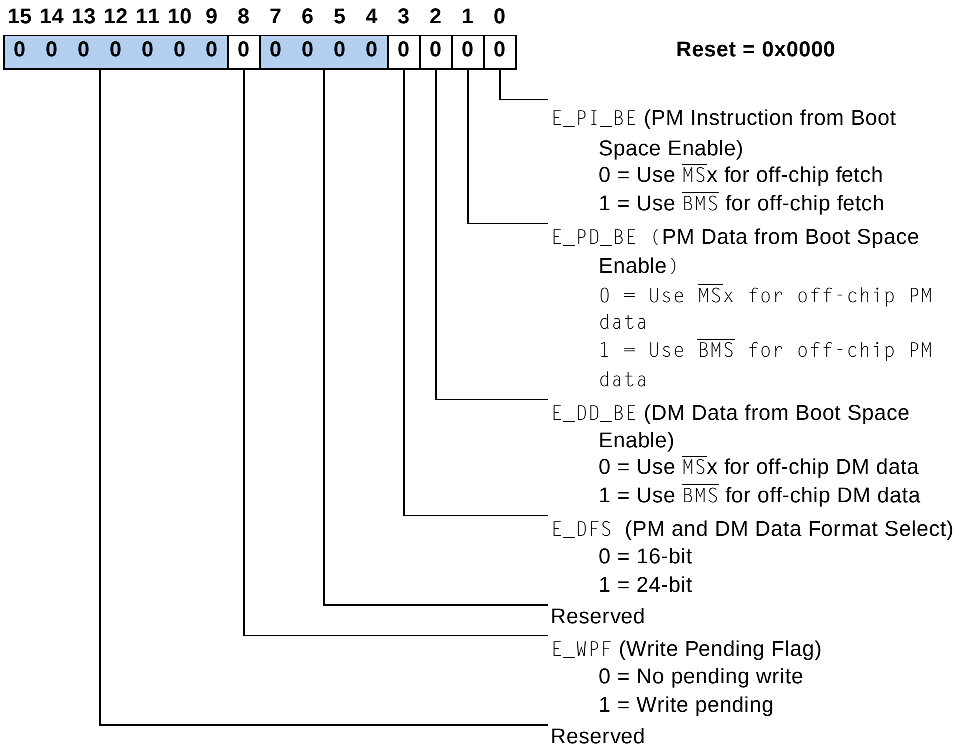
The EMI Control/Status (`E_STAT`) register configures access to external or boot memory space, selects the external data format, and indicates pending status for memory writes.

Figure B-35 on page B-101 provides bit descriptions.

External Memory Interface Control (EMICTL) Register

This register address is `0x06:0x201`. The EMI Interface Control (`EMICTL`) register is a 7-bit register. It can be used to configure the interface for an 8- or 16-bit external data bus. The register provides a lock bit to disable write accesses to the EMI Memory Access Control registers. Setting the lock bit in the EMI Control register causes the arbitration unit to provide grants only to direct access or peripheral register access requests.

Separate register bits are also provided to set the read and write strobe sense for positive logic (bit=0) or negative logic (bit=1). The sense bits are common to all memory spaces. Figure B-36 on page B-102 provides bit descriptions.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
E_PI_BE (PM Instruction from Boot Space Enable)
0 = Use $\overline{\text{MS}}$x for off-chip fetch
1 = Use $\overline{\text{BMS}}$ for off-chip fetch
E_PD_BE (PM Data from Boot Space Enable)
0 = Use $\overline{\text{MS}}$x for off-chip PM data
1 = Use $\overline{\text{BMS}}$ for off-chip PM data
E_DD_BE (DM Data from Boot Space Enable)
0 = Use $\overline{\text{MS}}$x for off-chip DM data
1 = Use $\overline{\text{BMS}}$ for off-chip DM data
E_DFS (PM and DM Data Format Select)
0 = 16-bit
1 = 24-bit
Reserved
E_WPF (Write Pending Flag)
0 = No pending write
1 = Write pending
Reserved

Figure B-35. EMI Control/Status (E_STAT) Register Bits

Boot Memory Select Control (BMSCTL) Register

This register address is 0x06:0x202. The Boot Memory Select Control (BMSCTL) register stores configuration data for the Boot memory space. The following are six parameters that can be programmed to customize accesses for the selected memory space. Figure B-37 on page B-103 provides bit descriptions.

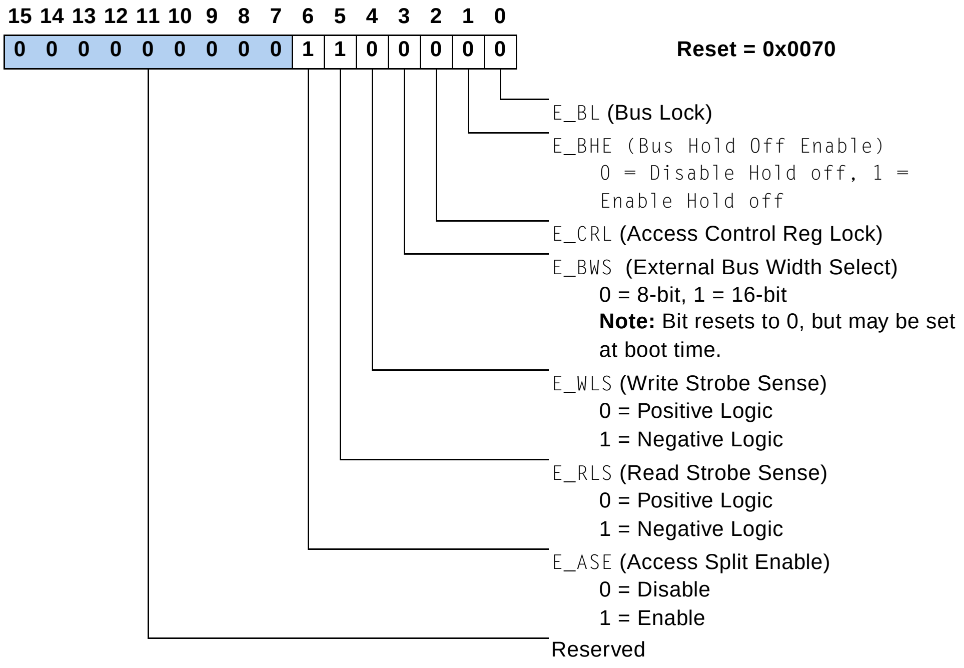

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
Reset = 0x0070
E_BL (Bus Lock)
E_BHE (Bus Hold Off Enable)
0 = Disable Hold off, 1 = Enable Hold off
E_CRL (Access Control Reg Lock)
E_BWS (External Bus Width Select)
0 = 8-bit, 1 = 16-bit
Note: Bit resets to 0, but may be set at boot time.
E_WLS (Write Strobe Sense)
0 = Positive Logic
1 = Negative Logic
E_RLS (Read Strobe Sense)
0 = Positive Logic
1 = Negative Logic
E_ASE (Access Split Enable)
0 = Disable
1 = Enable
Reserved

Figure B-36. EMI Control (EMICTL) Register Bits

The read and write waitstate counts indicate the number of I/O clock cycles that the EMI will wait before completing execution of an external transfer. The wait mode indicates how the waitstate counter and memory ACK line are used to determine the end of a transaction.These are the actual counts and are not encoded.

The base clock divider sets the I/O clock rate to be a sub-multiple of the peripheral clock rate. The write hold mode bit is set to 1 to extend the write data by one cycle following de-asserting of the strobe in order to provide more data hold time for slow devices.

Setting the CMS output enable bit to 1 enables the CMS signal to be asserted when the selected memory space is accessed. This bit has no effect on the ADSP-2191, because the ADSP-2191 does not have a CMS pin.

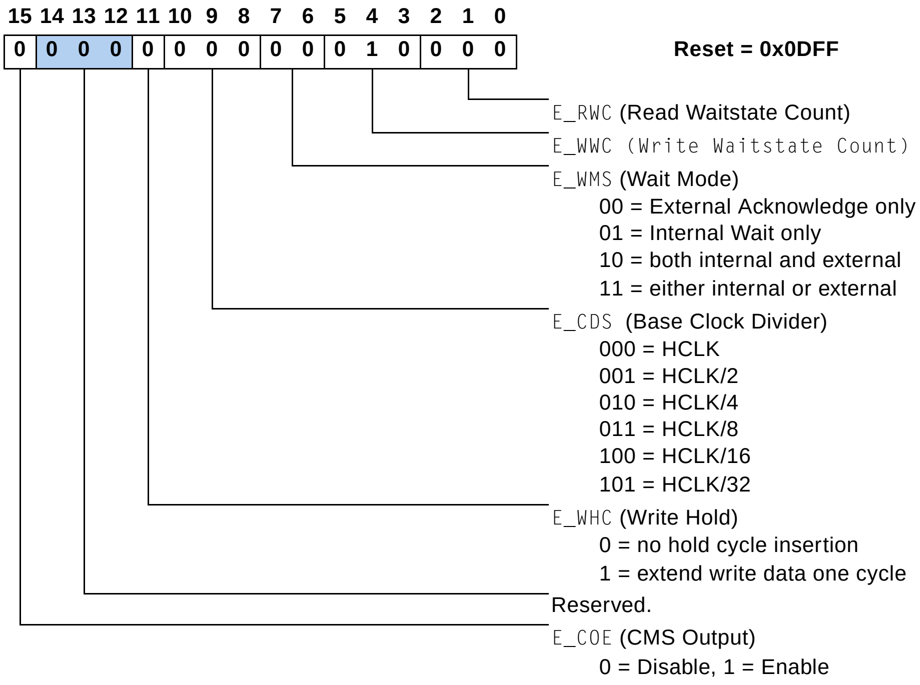


Figure B-37. Boot Memory Select Control (BMSCTL) Register Bits

Memory Select Control (MSxCTL) Registers

The ADSP-2191 supports selection of up to four memory banks (`MS3-0`). Each of these banks can also be configured to support 8-bit-wide or 16-bit-wide memories on a bank basis.

These memory bank registers' addresses are:

```
(Bank 0)   MS0CTRL 0x06:0x203
(Bank 1)   MS1CTRL 0x06:0x204
(Bank 2)   MS2CTRL 0x06:0x205
(Bank 3)   MS3CTRL 0x06:0x206
```

Each Memory Select Control (`MSxCTL`) register stores configuration data for the memory space. The following six parameters can be programmed to customize accesses for the selected memory space.

The read and write waitstate counts indicate the number of EMICLK clock cycles that the EMI will wait before completing execution of an external transfer. The wait mode indicates how the waitstate counter and memory ACK line are used to determine the end of a transaction. These are the actual counts and are not encoded.

The base clock divider sets the EMICLK clock rate to be a sub-multiple of the peripheral clock rate. The write hold mode bit is set to 1 to extend the write data by one cycle following de-asserting of the strobe in order to provide more data hold time for slow devices. Setting CMS output enable is set to 1 enables the `CMS` signal to be asserted when the selected memory space is accessed.

For information on the bits in this register (which are the same as the `BMSCTL` register), see Figure B-37 on page B-103.

I/O Memory Select Control (IOMSCTL) Registers

This register address is `0x06:0x207`. The I/O Memory Select Control (`IOMSCTL`) register stores configuration data for the I/O memory space. The following are six parameters that can be programmed to customize accesses for the selected memory space.

The read and write waitstate counts indicate the number of I/O clock cycles that the EMI will wait before completing execution of an external transfer. The wait mode indicates how the waitstate counter and memory ACK line are used to determine the end of a transaction. These are the actual counts and are not encoded.

The base clock divider sets the I/O clock rate to be a sub-multiple of the peripheral clock rate. The write hold mode bit is set to 1 to extend the write data by one cycle following de-asserting of the strobe in order to provide more data hold time for slow devices. Setting CMS output enable to 1 enables the CMS signal to be asserted when the selected memory space is accessed.

For information on the bits in this register (which are the same as the BMSCTL register), see Figure B-37 on page B-103.

External Port Status (EMISTAT) Register

The External Port Status (EMISTAT) register address is 0x06:0x208. The reset value is undefined. This read-only register can be polled to return three types of status shown below.

Figure B-38 on page B-106 provides bit descriptions.

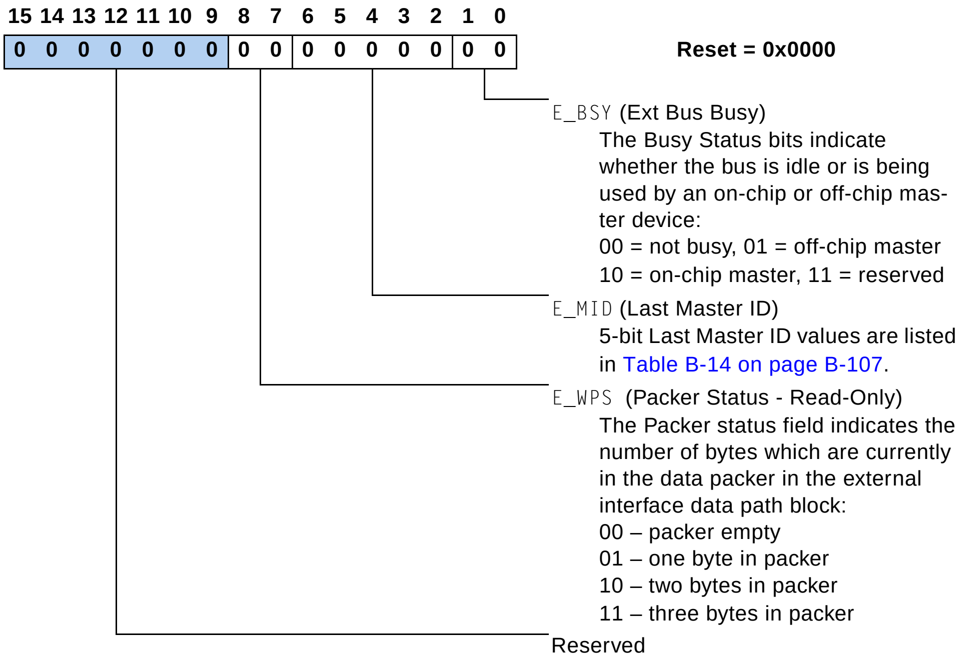

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
E_BSY (Ext Bus Busy)
The Busy Status bits indicate whether the bus is idle or is being used by an on-chip or off-chip master device:
00 = not busy, 01 = off-chip master
10 = on-chip master, 11 = reserved
E_MID (Last Master ID)
5-bit Last Master ID values are listed in Table B-14 on page B-107.
E_WPS (Packer Status - Read-Only)
The Packer status field indicates the number of bytes which are currently in the data packer in the external interface data path block:
00 – packer empty
01 – one byte in packer
10 – two bytes in packer
11 – three bytes in packer
Reserved

Figure B-38. External Port Status (EMISTAT) Register Bits

Table B-14. Last Master ID Parameters in EMI Status Register

Bit(s)	Name	Definition
6–2	E_MID	Last Master ID. The Last Master ID will return a 5-bit value which identifies the current or last device to use the interface: BitsDMA Masters NonDMA Masters 5432(bit 6=0)(bit 6=1) 0000SPORT0 RX DMADSP core I/O mem. 0001SPORT1 RX DMAHost port I/O mem. 0010SPORT2 RX DMAreserved 0011SPORT0 TX DMAreserved 0100SPORT1 TX DMAreserved 0101SPORT2 TX DMAreserved 0110SPI0 RX/TX DMAreserved 0111SPI1 RX/TX DMAreserved 1000UART RX DMAreserved 1001UART TX DMAreserved 1010Host RX/TX DMAreserved 1011MemDMA RX DMAreserved 1100MemDMA TX DMAreserved 1101reservedreserved 1110reservedreserved 1111reservedDSP core ext. mem.

Memory Page (MEMPGx) Registers

The EMI contains two registers which are used to program the lower page boundary addresses for the MS0, MS1, MS2, and MS3 memory spaces.

The MEMPGx registers are not intended to provide contiguous addressing across different MSx strobes, they are used to decrease the address space sticking to the individual MSx strobes.

The Memory Page registers' addresses are:

```
(Page 1/0)   MEMPG10 0x06:0x209
(Page 3/2)   MEMPG32 0x06:0x20A
```

The lower eight bits of Memory Page register 1/0 contain the upper eight bits of the lowest address in Bank 0 (MS0). The upper eight bits of Memory Page register 1/0 contain the upper eight bits of the lowest address in Bank 1 (MS1).

The lower eight bits of Memory Page register 3/2 contain the upper eight bits of the lowest address in Bank 2 (MS2). The upper eight bits of Memory Page register 3/2 contain the upper eight bits of the lowest address in Bank 3 (MS3). Memory bank address ranges are defined to include the lowest address in the bank and one less than the lowest address in the next highest bank.

Host Port Registers

The Host Port Bus Interface (HPI) provides a Host Port asynchronous parallel pin interface. The primary use of this interface is to provide a parallel slave port to an off-chip host agent allowing direct access to ADSP-2191 memory space, boot space, and IO space. This interface includes a built-in DMA controller that eases the transfer of block of data between the ADSP-2191 memory/boot space and the external Host processor. The ADSP-2191 supports boot loading under control of the HPI DMA Controller function.

This section includes the following topics:

- "Host Port Configuration (HPCR) Register" on page B-109
- "Host Port Direct Page (HPPR) Register" on page B-109
- "Host Port DMA Error (HPDER) Register" on page B-109
- "Host Port Semaphore (HPSMPHx) Registers" on page B-112
- "Host Port DMA Pointer (HOSTD_PTR) Register" on page B-112

- "Host Port DMA Configuration (HOSTD_CFG) Register" on page B-112
- "Host Port DMA Start Page (HOSTD_SRP) Register" on page B-113
- "Host Port DMA Start Address (HOSTD_SRA) Register" on page B-113
- "Host Port DMA Word Count (HOSTD_CNT) Register" on page B-113
- "Host Port DMA Chain Pointer (HOSTD_CP) Register" on page B-113
- "Host Port DMA Chain Pointer Ready (HOSTD_CPR) Register" on page B-115
- "Host Port DMA Interrupt (HOSTD_IRQ) Register" on page B-115

Host Port Configuration (HPCR) Register

The Host Port Configuration (`HPCR`) register address is `0x07:0x001`. The data path of the interface is set to 8-bits width after hardware reset. The host can change the data path width to 16-bits by writing the proper value to the I/O-mapped Host Port Configuration register.

In addition to the two select lines, transactions on the external host bus are controlled by four signals: `HALE`, `HRDB`,`HWRB`, and `HACK`. The values, sensed during the hardware reset sequence, are stored into the Host Port Configuration register as read-only bits. Figure B-39 on page B-110 provides Host Port Configuration register bit descriptions.

Two mode bits, `HPCR7-6`, define the functionality of the `HACK` line as shown in Table B-15 on page B-111.

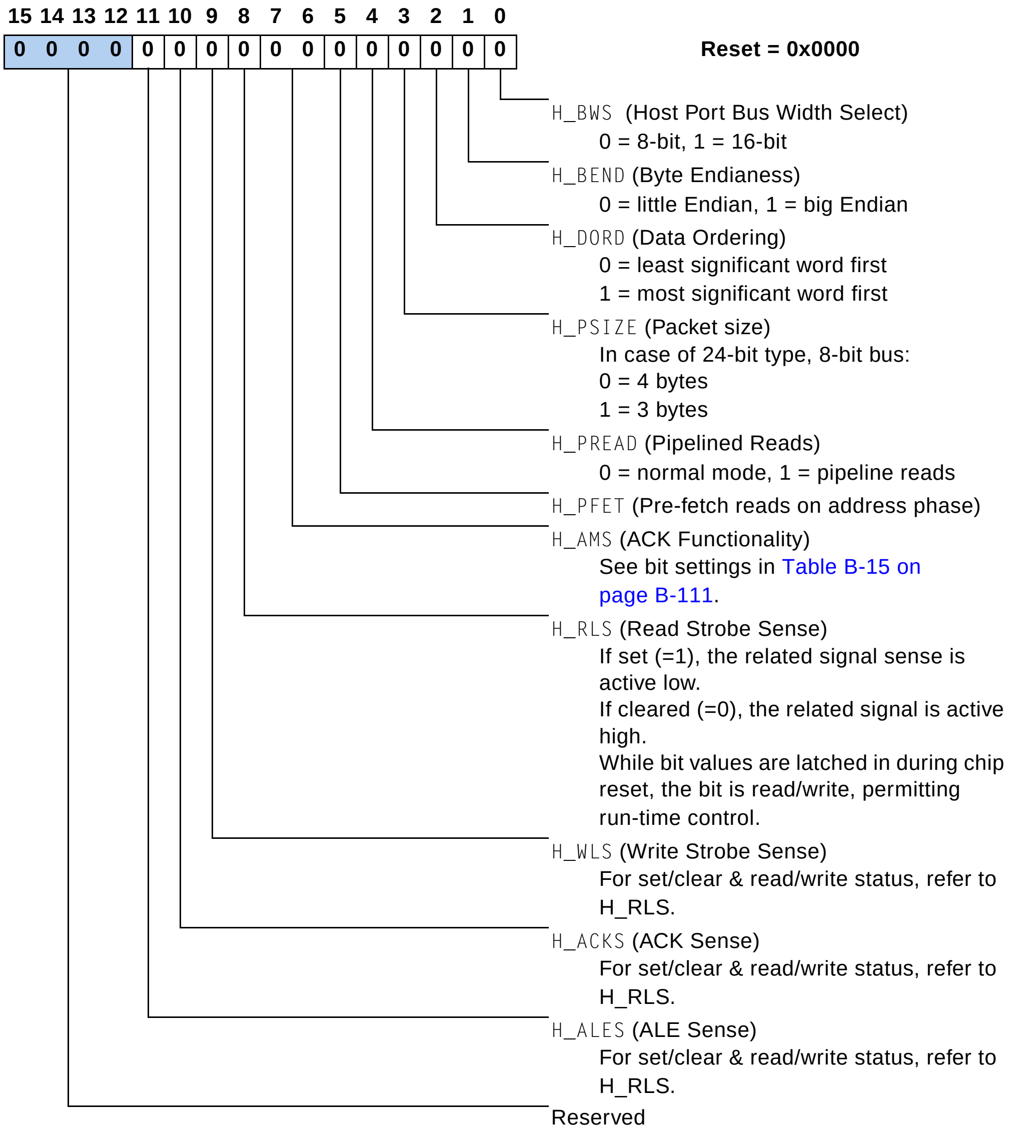

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
H_BWS (Host Port Bus Width Select)
0 = 8-bit, 1 = 16-bit
H_BEND (Byte Endianess)
0 = little Endian, 1 = big Endian
H_DORD (Data Ordering)
0 = least significant word first
1 = most significant word first
H_PSIZE (Packet size)
In case of 24-bit type, 8-bit bus:
0 = 4 bytes
1 = 3 bytes
H_PREAD (Pipelined Reads)
0 = normal mode, 1 = pipeline reads
H_PFET (Pre-fetch reads on address phase)
H_AMS (ACK Functionality)
See bit settings in Table B-15 on page B-111.
H_RLS (Read Strobe Sense)
If set (=1), the related signal sense is active low.
If cleared (=0), the related signal is active high.
While bit values are latched in during chip reset, the bit is read/write, permitting run-time control.
H_WLS (Write Strobe Sense)
For set/clear & read/write status, refer to H_RLS.
H_ACKS (ACK Sense)
For set/clear & read/write status, refer to H_RLS.
H_ALES (ALE Sense)
For set/clear & read/write status, refer to H_RLS.
Reserved

Figure B-39. Host Port Configuration (HPCR) Register Bits

Table B-15. HACK {7:6} Bit Descriptions

Bit(s)	Name	Definition
7–6	H_AMS	ACK Functionality Three functional modes selected by HPCR 7–6 are as follow (assuming active HIGH signal): • 00 ACK Mode: Acknowledge is active on strobes; HACK goes High from the leading edge of the strobe to indicate when the access can complete. After the Host samples the HACK active, it can complete the access by removing the strobe. The HPI then removes the HACK. • 01 READY Mode: Ready active on strobes, goes Low to insert waitstate during the access. If the HPI can not complete the access, it drives the HACK/READY line inactive. In this case, the Host has to extend the access by keeping the strobe active. When the Host samples the HACK active, it can then proceed and complete the access by removing the strobe. • 10 and 11 Reserved

Host Port Direct Page (HPPR) Register

The register address is `0x07:0x002`.

The Host Port Direct Page (`HPPR`) register should be set up to contain the most significant bits of the address that will be accessed (9-bits of memory page, bits `15-7`). Use this register also to configure the memory space (memory or boot) that will be accessed.

Host Port DMA Error (HPDER) Register

The register address is `0x07:0x003`. It is a 1-bit register. Bit 0 is set to zero, indicating DMA error; that is, a write to DMA Control register while DMA is active (data transfer is ongoing). The reset state is `0x0000`.

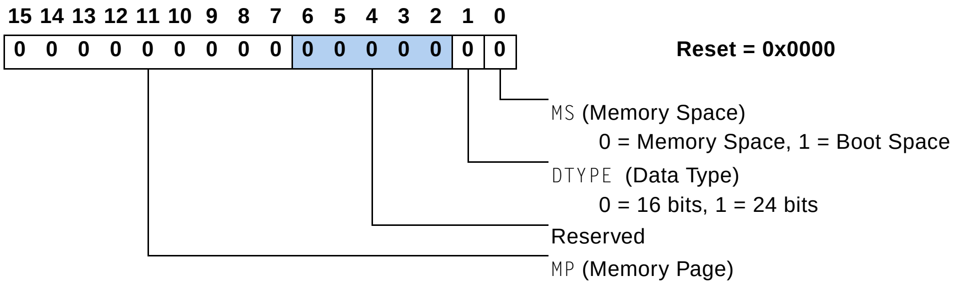


Figure B-40. Host Port Direct Page (HPPR) Register Bits

Host Port Semaphore (HPSMPHx) Registers

There are two semaphore bit registers. Their addresses are:

```
HPSMPHA 0x07:0x0FC
HPSMPHB 0x07:0x0FD
```

The 1-bit semaphore bit registers are used to implement a DMA ownership bit. This allows the processors, Host or DSP core, to determine if someone already uses the DMA controller. Semaphore register bits, bit 0 in each register, are set on "read" and are being reset when writing a 1 to the bit. The reset state is `0x0000`.

The DSP uses the `HPSMPHA` semaphore register when booting the DSP through the host port.

Host Port DMA Pointer (HOSTD_PTR) Register

The register address is `0x07:0x100`. This is a read-only register that holds the pointer to the current descriptor block for the HPI DMA operation. The reset state is `0x0000`.

Host Port DMA Configuration (HOSTD_CFG) Register

The register address is `0x07:0x101`.

Figure B-41 on page B-114 provides Host Port DMA Configuration register bit descriptions

If Bit 14, "DMA Interrupt on Completion" bit (bit 14) and Bit 0, the "DMA Enable" bit, are set, the HPI DMA Controller generates an interrupt when the Host Port DMA Word Count (`HOSTD_CNT`) register contents transitions from a one to a zero. Correct initial programming of the word count registers is essential to ensure that partial buffer contents (words) are not allowed to corrupt subsequent DMA transfers.

If the "DMA Interrupt on Completion" bit (DMA Completion = 0) is set and the "DMA Enable" bit is cleared, the HPI DMA Controller generates an interrupt prior to shutting down.

If the "DMA Interrupt on Error" bit (DMA Completion = 1) is set and the DMA operation completes with an error (bit 14), the HPI DMA Controller generates an error interrupt prior to disabling the DMA engine and shutting down.

Host Port DMA Start Page (HOSTD_SRP) Register

The register address is `0x07:0x102`. The 16-bit Host Port DMA Start Page (`HOSTD_SRP`) register holds a running pointer to the DMA address that is being accessed and the memory space being used for a DMA block transfer.

Host Port DMA Start Address (HOSTD_SRA) Register

The register address is `0x07:0x103`. This 16-bit read-only register holds the DMA block transfer start address. The reset state is `0x0000`.

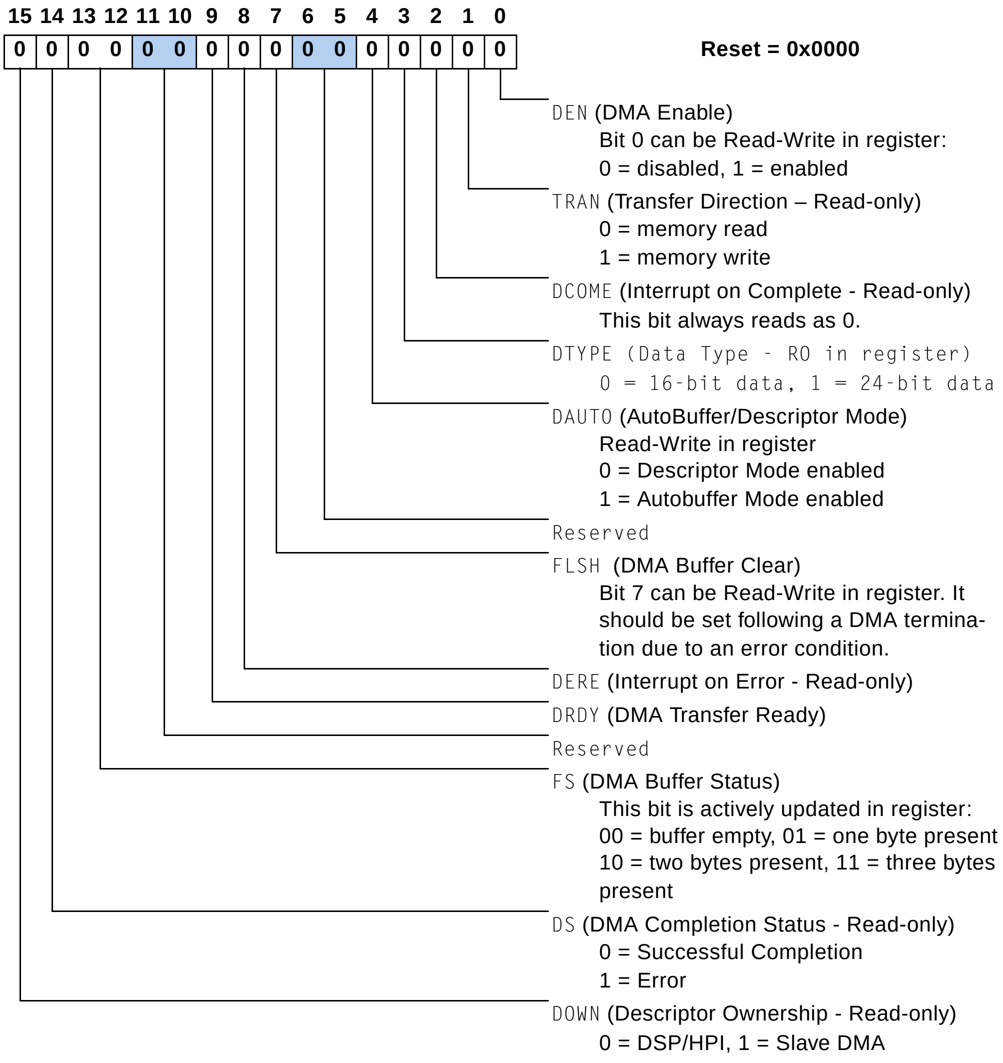

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DEN (DMA Enable)
Bit 0 can be Read-Write in register:
0 = disabled, 1 = enabled
TRAN (Transfer Direction – Read-only)
0 = memory read
1 = memory write
DCOME (Interrupt on Complete - Read-only)
This bit always reads as 0.
DTYPE (Data Type - RO in register)
0 = 16-bit data, 1 = 24-bit data
DAUTO (AutoBuffer/Descriptor Mode)
Read-Write in register
0 = Descriptor Mode enabled
1 = Autobuffer Mode enabled
Reserved
FLSH (DMA Buffer Clear)
Bit 7 can be Read-Write in register. It should be set following a DMA termination due to an error condition.
DERE (Interrupt on Error - Read-only)
DRDY (DMA Transfer Ready)
Reserved
FS (DMA Buffer Status)
This bit is actively updated in register:
00 = buffer empty, 01 = one byte present
10 = two bytes present, 11 = three bytes present
DS (DMA Completion Status - Read-only)
0 = Successful Completion
1 = Error
DOWN (Descriptor Ownership - Read-only)
0 = DSP/HPI, 1 = Slave DMA

Figure B-41. Host Port DMA Configuration (HOSTD_CFG) Register Bits

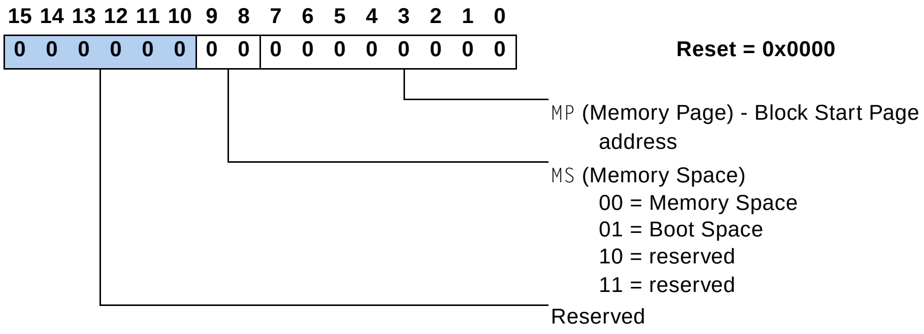


Figure B-42. Host Port DMA Start Page (HOSTD_SRP) Register Bits

Host Port DMA Word Count (HOSTD_CNT) Register

The register address is `0x07:0x104`. The 16-bit Write Count read-only register holds the number of words in the DMA block transfer. The reset state is `0x0000`.

Host Port DMA Chain Pointer (HOSTD_CP) Register

The register address is `0x07:0x105`. This 16-bit register holds the pointer to address of the next descriptor for a DMA transfer. The reset state is `0x0000`.

Host Port DMA Chain Pointer Ready (HOSTD_CPR) Register

The register address is `0x07:0x106`. Bit `0` in this 16-bit read-write register sets the status of the descriptor operation. If bit `0 is 0`,the status is descriptor ready. Bits `15:1` are not used. The reset state is `0x0000`.

Host Port DMA Interrupt (HOSTD_IRQ) Register

The register address is `0x07:0x107`. Bit `0` indicates when a DMA interrupt is pending (if set, =1) or is not pending (if cleared, =0, reset value). Bits `15:1` are not used. The reset state is `0x0000`.

Register and Bit #define File (def2191.h)

The following example definitions file is for the ADSP-2191 DSP. For the most current definitions file, use the version of this file that comes with the software development tools.

The version of the file in this appendix is included as a guide only.

```
/************************************************************
 *
 * def2191.h : $Revision: 1.7.4.1 $
 *
 * (c) Copyright 1998-2002 Analog Devices, Inc.
 *     All rights reserved.
 *
************************************************************/

/*
** System register bit and address defines to symbolic names
** for the ADSP-2191 DSP.
*/

#ifndef __DEF2191_H_
#define __DEF2191_H_

// Begin with a 219x CORE
#include <def219x.h>
```

```
//-------------------------------------------------------------
//                  I/O Processor Register Map
//-------------------------------------------------------------

// DMA Bus Bridge; these are on IOPG=0x00

#define DMA_Bus_Bridge_Page 0x00
#define D_ADLO_0  0x044  // DMA Bridge Addr FIFO 0 Reg (15-0)
#define D_ADLO_1  0x045  // DMA Bridge Addr FIFO 1 Reg (15-0)
#define D_ADLO_2  0x046  // DMA Bridge Addr FIFO 2 Reg (15-0)
#define D_ADLO_3  0x047  // DMA Bridge Addr FIFO 3 Reg (15-0)
#define D_DALO_0  0x048  // DMA Bridge Data FIFO 0 (15-0)
#define D_DALO_1  0x049  // DMA Bridge Data FIFO 1 (15-0)
#define D_DALO_2  0x04a  // DMA Bridge Data FIFO 2 (15-0)
#define D_DALO_3  0x04b  // DMA Bridge Data FIFO 3 (15-0)
#define ADDAHI_0  0x04c  // DMA Bridge Addr & Data FIFO 0 (23-16)
#define ADDAHI_1  0x04d  // DMA Bridge Addr & Data FIFO 1 (23-16)
#define ADDAHI_2  0x04e  // DMA Bridge Addr & Data FIFO 2 (23-16)
#define ADDAHI_3  0x04f  // DMA Bridge Addr & Data FIFO 3 (23-16)

// External Access Bridge; these are on IOPG=0x00

#define External_Access_Bridge_Page 0x00
#define E_STAT   0x080   // EAB config/status register

// Trace Buffer; these are on IOPG=0x00

#define Trace_Buffer_Page 0x00
#define TCSR    0x0c0   // Trace Control/Status Register
#define STBUF0  0x0c1   // Source Top of Stack reg (Low bits)
#define STBUF1  0x0c2   // Source Top of Stack reg (High bits)
#define DTBUF0  0x0c3   // Destin Top of Stack reg (Low bits)
#define DTBUF1  0x0c4   // Destin Top of Stack reg (High bits)
```

```
// JTAG Debug; these are on IOPG=0x00

#define JTAG_and_Debug_Page 0x00
#define INDATA    0x0E0   // INDATA register
#define OUTDATA   0x0E1   // OUTDATA Register
#define JDCSR     0x0E2   // JDCC Control/Status register

// ADSP-2191 On-Chip System IO Space

// Clock and System Control;
// these are on IOPG=0x00 (0x00200-0x003FF)

#define Clock_and_System_Control_Page 0x00
#define PLLCTL    0x200   // PLL Control register
#define LOCKCNT   0x201   // PLL Lock Counter
#define SWRST     0x202   // Software Reset Register
#define NXTSCR    0x203   // Next System Configuration register
#define SYSCR     0x204   // System Configuration register

// Reserved  (0x00400-0x005FF)

// Interrupt Controller; these are on IOPG=0x01 (0x00600-0x007FF)

#define Interrupt_Controller_Page 0x01
#define IPR0     0x200  // Interrupt Priority Register 0
#define IPR1     0x201  // Interrupt Priority Register 1
#define IPR2     0x202  // Interrupt Priority Register 2
#define IPR3     0x203  // Interrupt Priority Register 3
#define INTRD0   0x204  // Source Interrupt Register 0
#define INTRD1   0x205  // Source Interrupt Register 1
#define INTRD2   0x206  // Source Interrupt Register 2
#define INTRD3   0x207  // Source Interrupt Register 3
#define INTRD4   0x208  // Source Interrupt Register 4
#define INTRD5   0x209  // Source Interrupt Register 5
```

```
#define INTRD6   0x20A  // Source Interrupt Register 6
#define INTRD7   0x20B  // Source Interrupt Register 7
#define INTRD8   0x20C  // Source Interrupt Register 8
#define INTRD9   0x20D  // Source Interrupt Register 9
#define INTRD10  0x20E  // Source Interrupt Register 10
#define INTRD11  0x20F  // Source Interrupt Register 11

// Memory DMA Controller;
// these are on IOPG=0x02 (0x00800-0x009FF)

#define Memory_DMA_Controller_Page 0x02
#define DMACW_PTR 0x100   // Mem Wr Channel - DMA Current Pointer
#define DMACW_CFG 0x101   // Mem Wr Channel - DMA Configuration
#define DMACW_SRP 0x102   // Mem Wr Channel - DMA Start Page
#define DMACW_SRA 0x103   // Mem Wr Channel - DMA Start Address
#define DMACW_CNT 0x104   // Mem Wr Channel - DMA Count
#define DMACW_CP  0x105   // Mem Wr Channel - DMA Next Desc Ptr
#define DMACW_CPR 0x106   // Mem Wr Channel - DMA Descript Ready
#define DMACW_IRQ 0x107   // Mem Wr Channel - DMA Interrupt Reg

#define DMACR_PTR 0x180   // Mem Rd Channel - DMA Current Pointer
#define DMACR_CFG 0x181   // Mem Rd Channel - DMA Configuration
#define DMACR_SRP 0x182   // Mem Rd Channel - DMA Start Page
#define DMACR_SRA 0x183   // Mem Rd Channel - DMA Start Address
#define DMACR_CNT 0x184   // Mem Rd Channel - DMA Count
#define DMACR_CP  0x185   // Mem Rd Channel - DMA Next Descr Ptr
#define DMACR_CPR 0x186   // Mem Rd Channel - DMA Descr Ready
#define DMACR_IRQ 0x187   // Mem Rd Channel - DMA Int Register

// SPORT0 Controller; these are on IOPG=0x02 (0x00A00-0x00BFF)

#define SPORT0_Controller_Page 0x02
#define SP0_TCR   0x200   // SPORT0 Transmit Config Register
#define SP0_RCR   0x201   // SPORT0 Receive Config Register
```

```
#define SP0_TX     0x202   // SPORT0 TX transmit Register
#define SP0_RX     0x203   // SPORT0 RX Receive register
#define SP0_TSCKDIV  0x204  // SPORT0 Trans Serial Clock Divider
#define SP0_RSCKDIV  0x205  // SPORT0 Rec Serial Clock Divider
#define SP0_TFSDIV   0x206  // SPORT0 Transmit Frame Sync Divider
#define SP0_RFSDIV   0x207  // SPORT0 Receive Frame Sync Divider
#define SP0_STATR    0x208  // SPORT0 Status Register
#define SP0_MTCS0    0x209  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS1    0x20A  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS2    0x20B  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS3    0x20C  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS4    0x20D  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS5    0x20E  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS6    0x20F  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MTCS7    0x210  // SPORT0 Multi-Chan Trans Sel Reg
#define SP0_MRCS0    0x211  // SPORT0 Multi-Chan Rec Sel Reg
#define SP0_MRCS1    0x212  // SPORT0 Multi-Chan Rec Sel Reg
#define SP0_MRCS2    0x213  // SPORT0 Multi-Chan Rec Sel Reg
#define SP0_MRCS3    0x214  // SPORT0 Multi-Chan Rec Sel Reg
#define SP0_MRCS4    0x215  // SPORT0 Multi-Chan Rec Sel Reg
#define SP0_MRCS5    0x216  // Multi-Chan Rec Sel Register
#define SP0_MRCS6    0x217  // SPORT0 Multi-Chan Rec Sel Register
#define SP0_MRCS7    0x218  // SPORT0 Multi-Chan Rec Sel Register
#define SP0_MCMC1    0x219  // SPORT0 Multi-Chan Config Reg 1
#define SP0_MCMC2    0x21A  // SPORT0 Multi-Chan Config Reg 2

#define SP0DR_PTR    0x300  // SPORT0 -RCV DMA Current Pointer
#define SP0DR_CFG    0x301  // SPORT0 -RCV DMA Configuration
#define SP0DR_SRP    0x302  // SPORT0 -RCV DMA Start Page
#define SP0DR_SRA    0x303  // SPORT0 -RCV DMA Start Address
#define SP0DR_CNT    0x304  // SPORT0 -RCV DMA Count
#define SP0DR_CP     0x305  // SPORT0 -RCV DMA Next Desc Pointer
#define SP0DR_CPR    0x306  // SPORT0 -RCV DMA Descriptor Ready
#define SP0DR_IRQ    0x307  // SPORT0 -RCV DMA Interrupt Register
```

```
#define SP0DT_PTR    0x380  // SPORT0 -XMT DMA Current Pointer
#define SP0DT_CFG    0x381  // SPORT0 -XMT DMA Configuration
#define SP0DT_SRP    0x382  // SPORT0 -XMT DMA Start Page
#define SP0DT_SRA    0x383  // SPORT0 -XMT DMA Start Address
#define SP0DT_CNT    0x384  // SPORT0 -XMT DMA Count
#define SP0DT_CP     0x385  // SPORT0 -XMT DMA Next Descr Pointer
#define SP0DT_CPR    0x386  // SPORT0 -XMT DMA Descriptor Ready
#define SP0DT_IRQ    0x387  // SPORT0 -XMT DMA Interrupt Register

// SPORT1 Controller; these are on IOPG=0x03 (0x00C00-0x00DFF)

#define SPORT1_Controller_Page 0x03
#define SP1_TCR    0x000  // SPORT1 Transmit Config Register
#define SP1_RCR    0x001  // SPORT1 Receive Config Register
#define SP1_TX     0x002  // SPORT1 TX transmit Register
#define SP1_RX     0x003  // SPORT1 RX Receive register
#define SP1_TSCKDIV  0x004  // SPORT1 Trans Serial Clock Divider
#define SP1_RSCKDIV  0x005  // SPORT1 Rec Serial Clock Divider
#define SP1_TFSDIV   0x006  // SPORT1 Transmit Frame Sync Divider
#define SP1_RFSDIV  0x007  // SPORT1 Receive Frame Sync Divider
#define SP1_STATR   0x008  // SPORT1 Status Register
#define SP1_MTCS0   0x009  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS1   0x00A  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS2   0x00B  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS3   0x00C  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS4   0x00D  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS5   0x00E  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS6   0x00F  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MTCS7   0x010  // SPORT1 Multi-Chan Trans Sel Reg
#define SP1_MRCS0   0x011  // SPORT1 Multi-Chan Rec Sel Reg
#define SP1_MRCS1   0x012  // SPORT1 Multi-Chan Rec Sel Register
#define SP1_MRCS2   0x013  // SPORT1 Multi-Chan Rec Select Reg
#define SP1_MRCS3   0x014  // SPORT1 Multi-Chan Rec Select Reg
```

```
#define SP1_MRCS4   0x015  // SPORT1 Multi-Chan Rec Select Reg
#define SP1_MRCS5   0x016  // SPORT1 Multi-Chan Rec Select Reg
#define SP1_MRCS6   0x017  // SPORT1 Multi-Chan Rec Select Reg
#define SP1_MRCS7   0x018  // SPORT1 Multi-Chan Rec Select Reg
#define SP1_MCMC1   0x019  // SPORT1 Multi-Chan Config Reg 1
#define SP1_MCMC2   0x01A  // SPORT1 Multi-Chan Config Reg 2

#define SP1DR_PTR   0x100  // SPORT1 -DMA RCV Current Pointer
#define SP1DR_CFG   0x101  // SPORT1 -RCV DMA Configuration
#define SP1DR_SRP   0x102  // SPORT1 -RCV DMA Start Page
#define SP1DR_SRA   0x103  // SPORT1 -RCV DMA Start Address
#define SP1DR_CNT   0x104  // SPORT1 -RCV DMA Count
#define SP1DR_CP    0x105  // SPORT1 -RCV DMA Next Descr Pointer
#define SP1DR_CPR   0x106  // SPORT1 -RCV DMA Descriptor Ready
#define SP1DR_IRQ   0x107  // SPORT1 -RCV DMA Interrupt Register

#define SP1DT_PTR   0x180  // SPORT1 -XMT DMA Current Pointer
#define SP1DT_CFG   0x181  // SPORT1 -XMT DMA Configuration
#define SP1DT_SRP   0x182  // SPORT1 -XMT DMA Start Page
#define SP1DT_SRA   0x183  // SPORT1 -XMT DMA Start Address
#define SP1DT_CNT   0x184  // SPORT1 -XMT DMA Count
#define SP1DT_CP    0x185  // SPORT1 -XMT DMA Next Descr Pointer
#define SP1DT_CPR   0x186  // SPORT1 -XMT DMA Descriptor Ready
#define SP1DT_IRQ   0x187  // SPORT1 -XMT DMA Interrupt Reg

// SPORT2 Controller; these are on IOPG=0x03 (0x00E00-0x00FFF)

#define SPORT2_Controller_Page 0x03
#define SP2_TCR    0x200  // SPORT2 Transmit Configuration Reg
#define SP2_RCR    0x201  // SPORT2 Receive Configuration Reg
#define SP2_TX     0x202  // SPORT2 TX transmit Register
#define SP2_RX     0x203  // SPORT2 RX Receive register
#define SP2_TSCKDIV  0x204  // SPORT2 Trans Serial Clock Divider
#define SP2_RSCKDIV  0x205  // SPORT2 Rec Serial Clock Divider
```

```
#define SP2_TFSDIV   0x206  // SPORT2 Transmit Frame Sync Divider
#define SP2_RFSDIV   0x207  // SPORT2 Receive Frame Sync Divider
#define SP2_STATR    0x208  // SPORT2 Status Register
#define SP2_MTCS0    0x209  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS1    0x20A  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS2    0x20B  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS3    0x20C  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS4    0x20D  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS5    0x20E  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS6    0x20F  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MTCS7    0x210  // SPORT2 Multi-Chan Trans Sel Reg
#define SP2_MRCS0    0x211  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS1    0x212  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS2    0x213  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS3    0x214  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS4    0x215  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS5    0x216  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS6    0x217  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MRCS7    0x218  // SPORT2 Multi-Chan Rec Sel Register
#define SP2_MCMC1    0x219  // SPORT2 Multi-Chan Config Reg 1
#define SP2_MCMC2    0x21A  // SPORT2 Multi-Chan Config Reg 2

#define SP2DR_PTR    0x300  // SPORT2 -DMA RCV Current Ptr reg
#define SP2DR_CFG    0x301  // SPORT2 -RCV DMA Config register
#define SP2DR_SRP    0x302  // SPORT2 -RCV DMA Start Page reg
#define SP2DR_SRA    0x303  // SPORT2 -RCV DMA Start Address reg
#define SP2DR_CNT    0x304  // SPORT2 -RCV DMA Count register
#define SP2DR_CP     0x305  // SPORT2 -RCV DMA Next Descr Ptr reg
#define SP2DR_CPR    0x306  // SPORT2 -RCV DMA Descriptor Ready
#define SP2DR_IRQ    0x307  // SPORT2 -RCV DMA Interrupt Reg

#define SP2DT_PTR  0x380  // SPORT2 -XMT DMA Current Ptr register
#define SP2DT_CFG  0x381  // SPORT2 -XMT DMA Config register
#define SP2DT_SRP  0x382  // SPORT2 -XMT DMA Start Page register
```

```
#define SP2DT_SRA  0x383  // SPORT2 -XMT DMA Start Address reg
#define SP2DT_CNT  0x384  // SPORT2 -XMT DMA Count register
#define SP2DT_CP   0x385  // SPORT2 -XMT DMA Next Desc Ptr reg
#define SP2DT_CPR  0x386  // SPORT2 -XMT DMA Descriptor Ready
#define SP2DT_IRQ  0x387  // SPORT2 -XMT DMA Int Register

// SPI0 Controller; these are on IOPG=0x04 (0x01000-0x011FF)

#define SPI0_Controller_Page 0x04
#define SPICTL0    0x000  // SPI0 Control Register
#define SPIFLG0    0x001  // SPI0 Flag register
#define SPIST0     0x002  // SPI0 Status register
#define TDBR0      0x003  // SPI0 Transmit Data Buffer Register
#define RDBR0      0x004  // SPI0 Receive Data Buffer Register
#define SPIBAUD0   0x005  // SPI0 Baud rate Register
#define RDBRS0     0x006  // SPI0 Rec Data Buffer Shadow Register

#define SPI0D_PTR   0x100  // SPI 0 -DMA Current Pointer register
#define SPI0D_CFG   0x101  // SPI 0 -DMA Configuration register
#define SPI0D_SRP   0x102  // SPI 0 -DMA Start Page register
#define SPI0D_SRA   0x103  // SPI 0 -DMA Start Address register
#define SPI0D_CNT   0x104  // SPI 0 -DMA Count register
#define SPI0D_CP    0x105  // SPI 0 -DMA Next Descriptor Pointer
#define SPI0D_CPR   0x106  // SPI 0 -DMA Descriptor Ready
#define SPI0D_IRQ   0x107  // SPI 0 -DMA Interrupt register

// SPI1 Controller; these are on IOPG=0x04 (0x01200-0x013FF)

#define SPI1_Controller_Page 0x04
#define SPICTL1    0x200  // SPI1 Control Register
#define SPIFLG1    0x201  // SPI1 Flag register
#define SPIST1     0x202  // SPI1 Status register
#define TDBR1      0x203  // SPI1 Transmit Data Buffer Register
#define RDBR1      0x204  // SPI1 Receive Data Buffer Register
```

```
#define SPIBAUD1   0x205  // SPI1 Baud rate Register
#define RDBRS1     0x206  // SPI1 Receive Data Buffer Shadow Reg

#define SPI1D_PTR   0x300  // SPI 1 -DMA Current Pointer register
#define SPI1D_CFG   0x301  // SPI 1 -DMA Configuration register
#define SPI1D_SRP   0x302  // SPI 1 -DMA Start Page register
#define SPI1D_SRA   0x303  // SPI 1 -DMA Start Address register
#define SPI1D_CNT   0x304  // SPI 1 -DMA Count register
#define SPI1D_CP    0x305  // SPI 1 -DMA Next Descriptor Pointer
#define SPI1D_CPR   0x306  // SPI 1 -DMA Descriptor Ready
#define SPI1D_IRQ   0x307  // SPI 1 -DMA Interrupt register

// UART Controller; these are on IOPG=0x05 (0x01400-0x015FF)

#define UART_Controller_Page 0x05
#define THR        0x000   // UART - Transmit Holding register
#define RBR        0x000   // UART - Receive Buffer register
#define DLL        0x000   // UART - Divisor Latch (Low-Byte)
#define IER        0x001   // UART - Interrupt Enable Register
#define DLH        0x001   // UART - Divisor Latch (High-Byte)
#define IIR        0x002   // UART - Inter Identification Reg
#define LCR        0x003   // UART - Line Control Register
#define MCR        0x004   // UART - Module Control Register
#define LSR        0x005   // UART - Line Status Register
#define MSR        0x006   // UART - Modem Status Register
#define SCR        0x007   // UART - Scratch Register

#define UARDR_PTR  0x100   // UART -DMA RCV Current Ptr reg
#define UARDR_CFG  0x101   // UART -RCV DMA Config register
#define UARDR_SRP  0x102   // UART -RCV DMA Start Page register
#define UARDR_SRA  0x103   // UART -RCV DMA Start Address reg
#define UARDR_CNT  0x104   // UART -RCV DMA Count register
#define UARDR_CP   0x105   // UART -RCV DMA Next Descr Ptr reg
#define UARDR_CPR  0x106   // UART -RCV DMA Descriptor Ready
```

```
#define UARDR_IRQ  0x107   // UART -RCV DMA Interrupt Register

#define UARDT_PTR  0x180   // UART -XMT DMA Current Ptr reg
#define UARDT_CFG  0x181   // UART -XMT DMA Config register
#define UARDT_SRP  0x182   // UART -XMT DMA Start Page register
#define UARDT_SRA  0x183   // UART -XMT DMA Start Address reg
#define UARDT_CNT  0x184   // UART -XMT DMA Count register
#define UARDT_CP   0x185   // UART -XMT DMA Next Descr Ptr reg
#define UARDT_CPR  0x186   // UART -XMT DMA Descriptor Ready
#define UARDT_IRQ  0x187   // UART -XMT DMA Inter register

// Timer; these are on IOPG=0x05 (0x01600-0x017FF)

#define Timer_Page 0x05
#define T_GSR0     0x200  // Timer 0 Global Status & Sticky Reg
#define T_CFGR0    0x201  // Timer 0 configuration Register
#define T_CNTL0    0x202  // Timer 0 Counter Register (low word)
#define T_CNTH0    0x203  // Timer 0 Counter Register (high word)
#define T_PRDL0    0x204  // Timer 0 Period Register (low word)
#define T_PRDH0    0x205  // Timer 0 Period Register (high word)
#define T_WLR0     0x206  // Timer 0 Width Register (low word)
#define T_WHR0     0x207  // Timer 0 Width Register (high word)
#define T_GSR1     0x208  // Timer 1 Global Status & Sticky Reg
#define T_CFGR1    0x209  // Timer 1 configuration register
#define T_CNTL1    0x20A  // Timer 1 Counter Register (low word)
#define T_CNTH1    0x20B  // Timer 1 Counter Register (high word)
#define T_PRDL1    0x20C  // Timer 1 Period Register (low word)
#define T_PRDH1    0x20D  // Timer 1 Period Register (high word)
#define T_WLR1     0x20E  // Timer 1 Width Register (low word)
#define T_WHR1     0x20F  // Timer 1 Width Register (high word)
#define T_GSR2     0x210  // Timer 2 Global Status & Sticky Reg
#define T_CFGR2    0x211  // Timer 2 configuration register
#define T_CNTL2    0x212  // Timer 2 Counter Register (low word)
#define T_CNTH2    0x213  // Timer 2 Counter Register (high word)
```

```
#define T_PRDL2    0x214  // Timer 2 Period Register (low word)
#define T_PRDH2    0x215  // Timer 2 Period Register (high word)
#define T_WLR2     0x216  // Timer 2 Width Register (low word)
#define T_WHR2     0x217  // Timer 2 Width Register (high word)

// The first version of the documentation had errors in the names
// used for the above definitions, so both variants are defined.
#define GSR0       0x200  // Timer 0 Global Status & Sticky Reg
#define CFGR0      0x201  // Timer 0 configuration Register
#define CNTL0      0x202  // Timer 0 Counter Register (low word)
#define CNTH0      0x203  // Timer 0 Counter Register (high word)
#define PRDL0      0x204  // Timer 0 Period Register (low word)
#define PRDH0      0x205  // Timer 0 Period Register (high word)
#define WLR0       0x206  // Timer 0 Width Register (low word)
#define WHR0       0x207  // Timer 0 Width Register (high word)
#define GSR1       0x208  // Timer 1 Global Status & Sticky Reg
#define CFGR1      0x209  // Timer 1 configuration register
#define CNTL1      0x20A  // Timer 1 Counter Register (low word)
#define CNTH1      0x20B  // Timer 1 Counter Register (high word)
#define PRDL1      0x20C  // Timer 1 Period Register (low word)
#define PRDH1      0x20D  // Timer 1 Period Register (high word)
#define WLR1       0x20E  // Timer 1 Width Register (low word)
#define WHR1       0x20F  // Timer 1 Width Register (high word)
#define GSR2       0x210  // Timer 2 Global Status & Sticky Reg
#define CFGR2      0x211  // Timer 2 configuration register
#define CNTL2      0x212  // Timer 2 Counter Register (low word)
#define CNTH2      0x213  // Timer 2 Counter Register (high word)
#define PRDL2      0x214  // Timer 2 Period Register (low word)
#define PRDH2      0x215  // Timer 2 Period Register (high word)
#define WLR2       0x216  // Timer 2 Width Register (low word)
#define WHR2       0x217  // Timer 2 Width Register (high word)

// General Purpose IO; these are on IOPG=0x06 (0x01800-0x019FF)
```

```
#define General_Purpose_IO 0x06
#define DIR        0x000  // Peripheral Flag Direction Register
#define FLAGC      0x002  // Peripheral Int Flag Register (clear)
#define FLAGS      0x003  // Peripheral Int Flag Register (set)
#define MASKAC     0x004  // Flag Mask Inter A Register (clear)
#define MASKAS     0x005  // Flag Mask Interrupt A Register (set)
#define MASKBC     0x006  // Flag Mask Int B Register (clear)
#define MASKBS     0x007  // Flag Mask Interrupt B Register (set)
#define FSPR       0x008  // Flag Source Polarity Register
#define FSSR       0x00A  // Flag Source Sensitivity Register
#define FSSRS      0x00B  // Flag Source Sensitivity Reg (set)
#define FSBER      0x00C  // Flag Set on BOTH Edges Register

// External Memory Interface;
// these are on IOPG=0x06 (0x01A00-0x01BFF)

// 0x01A00  Reserved

#define External_Memory_Interface_Page 0x06
#define EMICTL   0x201  // EMI control Register
#define BMSCTL   0x202  // Boot Space Access Control Register
#define MS0CTL   0x203  // Memory Space Bank 0 Access Control Reg
#define MS1CTL   0x204  // Memory Space Bank 1 Access Control Reg
#define MS2CTL   0x205  // Memory Space Bank 2 Access Control Reg
#define MS3CTL   0x206  // Memory Space Bank 3 Access Control Reg
#define IOMSCTL  0x207  // IO Space Access Control Register
#define EMISTAT  0x208  // External Port Status Register
#define MEMPG10  0x209  // Memory Page Register 1/0
#define MEMPG32  0x20A  // Memory Page Register 3/2

// Host Port Bus Interface;
// these are on IOPG=0x07 (0x01C00-0x01DFF)

// 0x01C00  Reserved
```

```
#define Host_Port_Bus_Interface_Page 0x07
#define HPCR       0x001  // Host Port Configuration Register
#define HPPR       0x002  // Host Port Direct Page Register
#define HPDER      0x003  // Host Port DMA Error Register

#define HPSMPHA    0x0FC  // Host Port Semaphore A Register
#define HPSMPHB    0x0FD  // Host Port Semaphore B Register
#define HPSMPHC    0x0FE  // Host Port Semaphore C Register
#define HPSMPHD    0x0FF  // Host Port Semaphore D Register

#define HOSTD_PTR  0x100  // Host Port DMA current Pointer
#define HOSTD_CFG  0x101  // Host Port DMA Configuration
#define HOSTD_SRP  0x102  // Host Port DMA Start Page
#define HOSTD_SRA  0x103  // Host Port DMA Start Address
#define HOSTD_CNT  0x104  // Host Port DMA Word Count
#define HOSTD_CP   0x105  // Host Port DMA Next Descriptor
#define HOSTD_CPR  0x106  // Host Port DMA Descriptor Ready
#define HOSTD_IRQ  0x107  // Host Port DMA Interrupt Register

#endif
```

C NUMERIC FORMATS

ADSP-219x family processors support 16-bit fixed-point data in hardware. Special features in the computation units allow programs to support other formats in software. This appendix describes various aspects of the 16-bit data format. It also describes how to implement a block floating-point format in software.

This appendix provides the following topics:

Un/Signed: Twos Complement Format

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

Signed numbers supported by the ADSP-219x family are in twos complement format. Signed-magnitude, ones complement, BCD, or excess-n formats are not supported.

Integer or Fractional

The ADSP-219x DSP family supports both fractional and integer data formats, with the exception that ADSP-2100 processors do not perform integer multiplication.

In an integer, the radix point is assumed to lie to the right of the LSB, so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure C-1. In twos complement format, the sign bit has a negative weight.

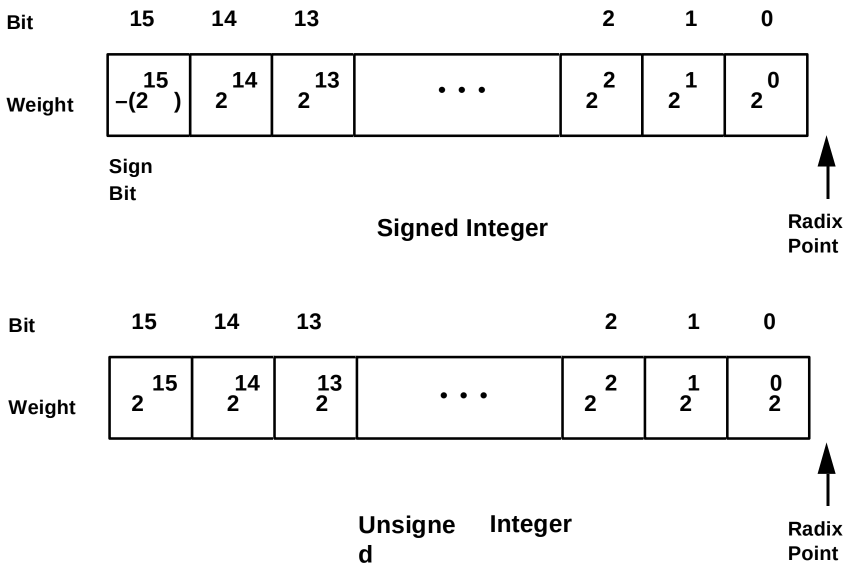


Figure C-1. Integer Format

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 C-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The notation used to describe a format consists two numbers separated by a period (.); the first number is the number of bits to the left of 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 C-2 is 13.3.

Table C-1 shows the ranges of numbers representable in the fractional formats that are possible with 16 bits.

Table C-1. Fractional Formats and Ranges

Format	# of Integer Bits	# of Fractio nal 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

Table C-1. Fractional Formats and Ranges (Cont'd)

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
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

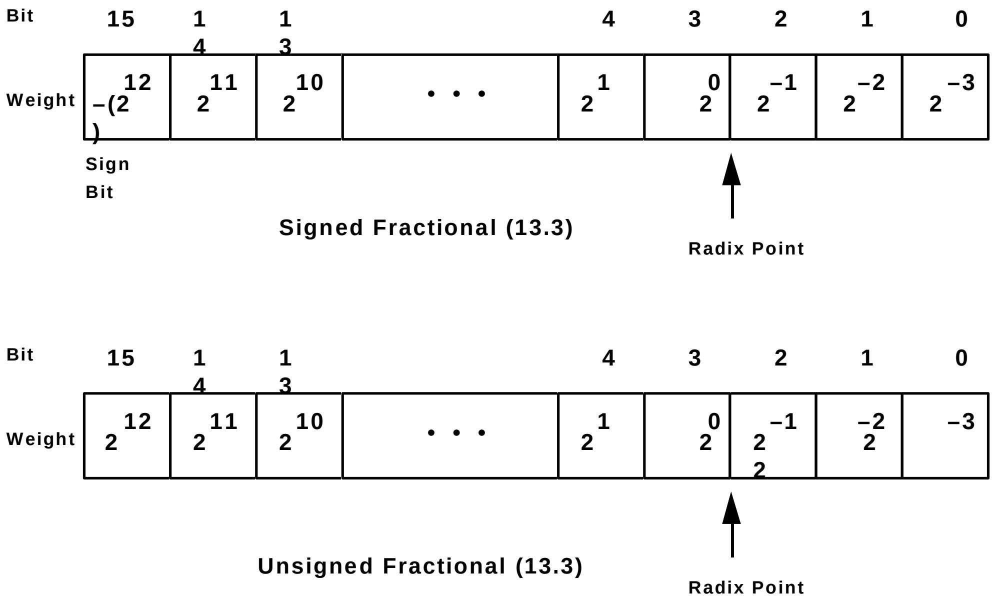


Figure C-2. Example of Fractional Format

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 ADSP-219x DSP family assembly language allows programs 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 C-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.

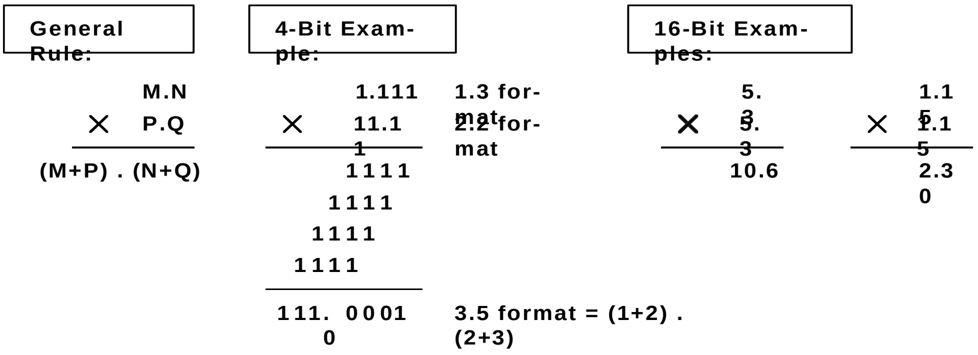


Figure C-3. Format of Multiplier Result

Fractional Mode and Integer Mode

The product of two twos-complement numbers has two sign bits. Since one of these bits is redundant, programs 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 ADSP-219x DSP family provides a fractional mode in 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 correctly formatted result.

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. If using a fractional data format, it is most convenient to use the 1.15 format.

In the integer mode, the left shift does not occur. This is the mode to use if both operands are integers (in the 16.0 format). The 32-bit multiplier result is in 32.0 format, also an integer.

In all ADSP-219x DSPs, fractional and integer modes are controlled by a bit in the `MSTAT` register. At reset, these processors default to the fractional mode.

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. Some additional programming is required to maintain a block floating-point format, however.

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. To convert a block of fixed-point values to block floating-point format, a program would shift each value left by the same amount and store the shift value as the block exponent. Typically, block floating-point format allows programs to shift out non-significant MSBs, increasing the precision available in each value. Programs can also use block floating-point format to eliminate the possibility of a data value overflowing. Figure C-4 shows an example. The three data samples each have at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing; thus, these bits are called guard bits. If it is known that a process will not cause any value to grow by more than these two bits, then the process can be run without loss of data. Afterward, however, the block must be adjusted to replace the guard bits before the next process.

Figure C-5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows. 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` is if the number of redundant sign bits is less than 2. In this example, `SB`=–1 after processing, indicating that 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.

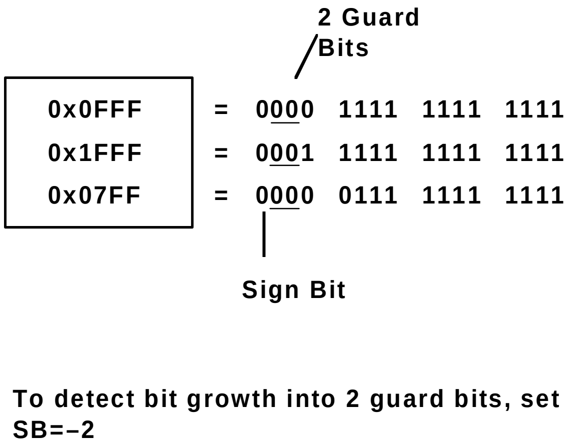

2 Guard Bits
0x0FFF = 0000 1111 1111 1111
0x1FFF = 0001 1111 1111 1111
0x07FF = 0000 0111 1111 1111
Sign Bit
To detect bit growth into 2 guard bits, set SB=–2

Figure C-4. Data with Guard Bits

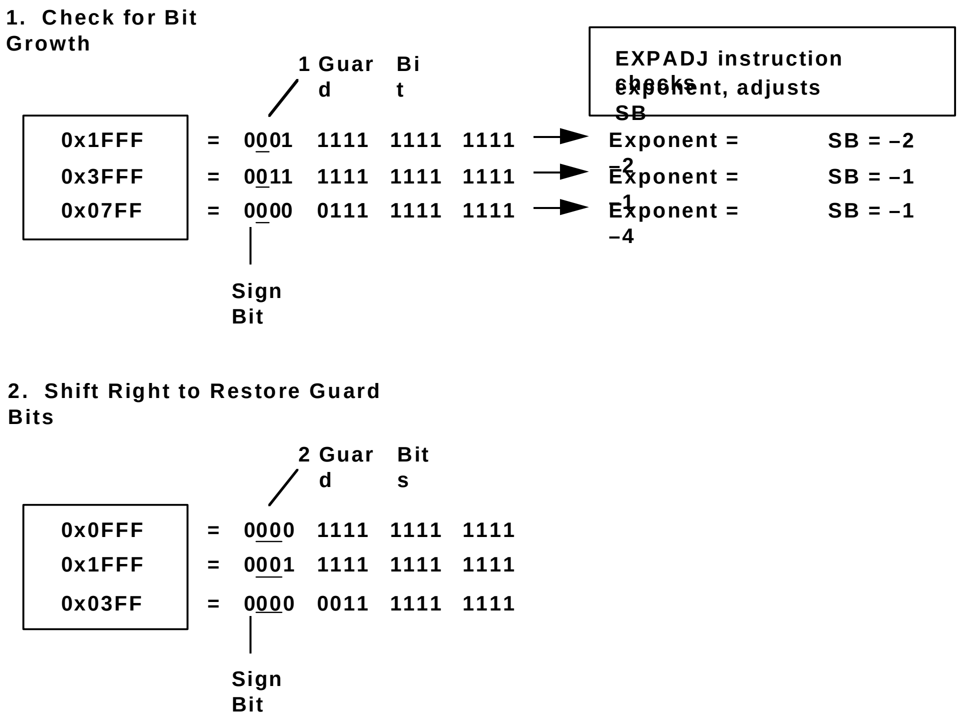


Figure C-5. Block Floating-Point Adjustment

I INDEX