ADSP-BF50x Blackfin® Processor Hardware Reference

Revision 1.0, December 2010

Part Number
82-100101-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

MEMORY

CHIP BUS HIERARCHY

EXTERNAL BUS INTERFACE UNIT

INTERNAL FLASH MEMORY

DIRECT MEMORY ACCESS

DYNAMIC POWER MANAGEMENT

CORE TIMER

WATCHDOG TIMER

PWM CONTROLLER

UART PORT CONTROLLERS

SPI-COMPATIBLE PORT CONTROLLER

SPORT CONTROLLER

PARALLEL PERIPHERAL INTERFACE

SYSTEM RESET AND BOOTING

SYSTEM DESIGN

SYSTEM MMR ASSIGNMENTS

TEST FEATURES

Contents

PREFACE

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

Purpose of This Manual

The *ADSP-BF50x Blackfin Processor Hardware Reference* provides architectural information about the ADSP-BF50x processors. This hardware reference provides the main architectural information about these processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see the *Blackfin Processor Programming Reference.* For timing, electrical, and package specifications, see the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

Intended Audience

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

Manual Contents

This manual consists of one volume:

- Chapter 1, “Introduction”
  Provides a high level overview of the processor, including peripherals, power management, and development tools.
- Chapter 2, “Memory”
  Describes processor-specific memory topics, including L1 memories and processor-specific memory MMRs.
- Chapter 3, “Chip Bus Hierarchy”
  Describes on-chip buses, including how data moves through the system.
- Chapter 4, “System Interrupts”
  Describes the system peripheral interrupts, including setup and clearing of interrupt requests.
- Chapter 5, “External Bus Interface Unit”
  Describes the external bus interface unit of the processor and memory interface.
- Chapter 6, “Internal Flash Memory”
  Describes the internal flash memory and programmable features.
- Chapter 7, “Direct Memory Access”
  Describes the peripheral DMA and Memory DMA controllers. Includes performance, software management of DMA, and DMA errors.
- Chapter 8, “Dynamic Power Management”
  Describes the clocking, including the PLL, and the dynamic power management controller.

- Chapter 9, “General-Purpose Ports”
  Describes the general-purpose I/O ports, including the structure of each port, multiplexing, configuring the pins, and generating interrupts.
- Chapter 10, “General-Purpose Timers”
  Describes the eight general-purpose timers.
- Chapter 11, “Core Timer”
  Describes the core timer.
- Chapter 12, “Watchdog Timer”
  Describes the watchdog timer.
- Chapter 13, “General-Purpose Counter”
  Describes the Rotary (up/down) Counter. This counter provides support for manually controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial or motor-control type of wheels.
- Chapter 14, “PWM Controller”
  Describes the The PWM controller—a flexible, programmable, three-phase PWM waveform generator that can be programmed to generate the required switching patterns to drive a three-phase voltage source inverter for ac induction motor (ACIM) or permanent magnet synchronous motor (PMSM) control.
- Chapter 15, “UART Port Controllers”
  Describes the Universal Asynchronous Receiver/Transmitter port that converts data between serial and parallel formats. The UART supports the half-duplex IrDA® SIR protocol as a mode-enabled feature.
- Chapter 16, “Two Wire Interface Controller”
  Describes the Two Wire Interface (TWI) controller, which allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000.

- Chapter 17, “CAN Module”
  Describes the CAN module, a low bit rate serial interface intended for use in applications where bit rates are typically up to 1Mbit/s.
- Chapter 18, “SPI-Compatible Port Controller”
  Describes the Serial Peripheral Interface (SPI) port that provides an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 19, “SPORT Controller”
  Describes the independent, synchronous Serial Port Controller which provides an I/O interface to a variety of serial peripheral devices.
- Chapter 20, “Parallel Peripheral Interface”
  Describes the Parallel Peripheral Interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and is used for digital video and data converter applications.
- Chapter 21, “Removable Storage Interface”
  Describes the RSI interface for multimedia cards (MMC), secure digital memory cards (SD), secure digital input/output cards (SDIO) and consumer electronic ATA devices (CE-ATA).
- Chapter 22, “ADC Control Module (ACM)”
  Describes the ADC control module (ACM), which provides an interface to synchronize the controls between the processor and the internal analog-to-digital converter (ADC) module.
- Chapter 23, “Analog/Digital Converter (ADC)”
  Describes the internal ADC, which is a dual, 12-bit, high speed, low power, successive approximation ADC that operates from a single 2.7 V to 5.25 V power supply and features throughput rates up to 1.66 MSPS. The device contains two ADCs, each preceded by a 3-channel multiplexer, and a low noise, wide bandwidth track-and-hold amplifier that can handle input frequencies in excess of 30 MHz.

- Chapter 24, “System Reset and Booting”
  Describes the booting methods, booting process and specific boot modes for the processor.
- Chapter 25, “System Design”
  Describes how to use the processor as part of an overall system. It includes information about bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Appendix A, “System MMR Assignments”
  Lists the memory-mapped registers included in this manual, their addresses, and cross-references to text.
- Appendix B, “Test Features”
  Describes test features for the processor, discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.

This hardware reference is a companion document to the *Blackfin Processor Programming Reference*.

What’s New in This Manual

This revision (1.0) is the third release of the *ADSP-BF50x Blackfin Processor Hardware Reference*. This revision corrects the following issues:

- Reset value for the PLL_CTL register in Chapter 8, “Dynamic Power Management”
- Pin and port control information in Chapter 15, “UART Port Controllers”
- Decoupling capacitor diagram in Chapter 25, “System Design”

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technical_support`
- E-mail tools questions to
  `processor.tools.support@analog.com`
- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
  ```
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA
  ```

Supported Processors

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

Blackfin (ADSP-BFxxx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families ADSP-BF50x, ADSP-BF51x, ADSP-BF52x, ADSP-BF53x, ADSP-BF54x, and ADSP-BF561 processors.

TigerSHARC® (ADSP-TSxxx) Processors

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

SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC families: ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, ADSP-2137x, ADSP-2146x, ADSP-2147x, and ADSP-2148x.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

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

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

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

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

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC, TigerSHARC, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/technical_library, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

EngineerZone

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

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

Social Networking Web Sites

You can now follow Analog Devices Blackfin development on Twitter and LinkedIn. To access:

- Twitter: http://twitter.com/blackfin
- LinkedIn: Network with the LinkedIn group, Analog Devices Blackfin: http://www.linkedin.com

Notation Conventions

Text conventions used in this manual are identified and described as follows. 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 required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.

Example	Description
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
filename	Non-keyword placeholders appear in text with italic style format.
	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

The ADSP-BF50x processors are members of the Blackfin processor family that offer significant high performance and low power features while retaining their ease-of-use benefits. The ADSP-BF504, ADSP-BF504F, and ADSP-BF506F processors have differing peripheral features. For details, see Table 1-1. Note that the ADSP-BF504 and ADSP-BF504F are pin-compatible.

This hardware reference is a companion document to the *Blackfin Processor Programming Reference*.

General Description of Processor

The ADSP-BF50x processor is a member of the Blackfin® family of products, incorporating the Analog Devices/Intel Micro Signal Architecture (MSA). Blackfin processors combine a dual-MAC state-of-the-art signal processing engine, the advantages of a clean, orthogonal RISC-like microprocessor instruction set, and single-instruction, multiple-data (SIMD) multimedia capabilities into a single instruction-set architecture.

The ADSP-BF50x processor is completely code compatible with other Blackfin processors. ADSP-BF50x processors offer performance up to 400 MHz and reduced static power consumption. The processor features are shown in Table 1-1.

Table 1-1. Processor Comparison

Feature		ADSP-BF504	ADSP-BF504F	ADSP-BF506F
Up/Down/Rotary Counters		2	2	2
Timer/Counters with PWM		8	8	8
3-Phase PWM Units		2	2	2
SPORTs		2	2	2
SPIs		2	2	2
UARTs		2	2	2
Parallel Peripheral Interface		1	1	1
Removable Storage Interface		1	1	1
CAN		1	1	1
TWI		1	1	1
Internal 32M Bit Flash		–	1	1
ADC Control Module (ACM)		1	1	1
Internal ADC		–	–	1
GPIOs		35	35	35
Memory (bytes)	L1 Instruction SRAM	16K	16K	16K
	L1 Instruction SRAM/Cache	16K	16K	16K
	L1 Data SRAM	16K	16K	16K
	L1 Data SRAM/Cache	16K	16K	16K
	L1 Scratchpad	4K	4K	4K
	L3 Boot ROM	4K	4K	4K

By integrating a rich set of industry-leading system peripherals and memory, Blackfin processors are the platform of choice for next-generation applications that require RISC-like programmability, multimedia support, and leading-edge signal processing in one integrated package.

Portable Low-Power Architecture

Blackfin processors provide world-class power management and performance. They are produced with a low-power and low-voltage design methodology and feature on-chip dynamic power management, which provides the ability to vary both the voltage and frequency of operation to significantly lower overall power consumption. This capability can result in a substantial reduction in power consumption, compared with just varying the frequency of operation. This allows longer battery life for portable appliances.

System Integration

The ADSP-BF50x processors are highly integrated system-on-a-chip solutions for the next generation of embedded industrial, instrumentation, and power/motion control applications. By combining industry-standard interfaces with a high-performance signal processing core, cost-effective applications can be developed quickly, without the need for costly external components. The system peripherals include a watchdog timer; two 32-bit up/down counters with rotary support; eight 32-bit timers/ counters with PWM support; two pairs of three-phase 16-bit center-based PWM units; two dual-channel, full-duplex synchronous serial ports (SPORTs); two serial peripheral interface (SPI) compatible ports; two UARTs with IrDA support; a parallel peripheral interface (PPI); a removable storage interface (RSI) controller; an internal ADC with 12 channels, 12 bits, up to 2 MSPS, an ACM controller; a controller area network (CAN) controller; a two-wire interface (TWI) controller; and an internal 32M bit flash.

Peripherals

The ADSP-BF50x processors contain a rich set of peripherals connected to the core via several high-bandwidth buses, providing flexibility in system configuration as well as excellent overall system performance. (See Figure 1-1.) Most of the peripherals are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces. Multiple on-chip buses provide enough bandwidth to keep the processor core running even when there is also activity on all of the on-chip and external peripherals.

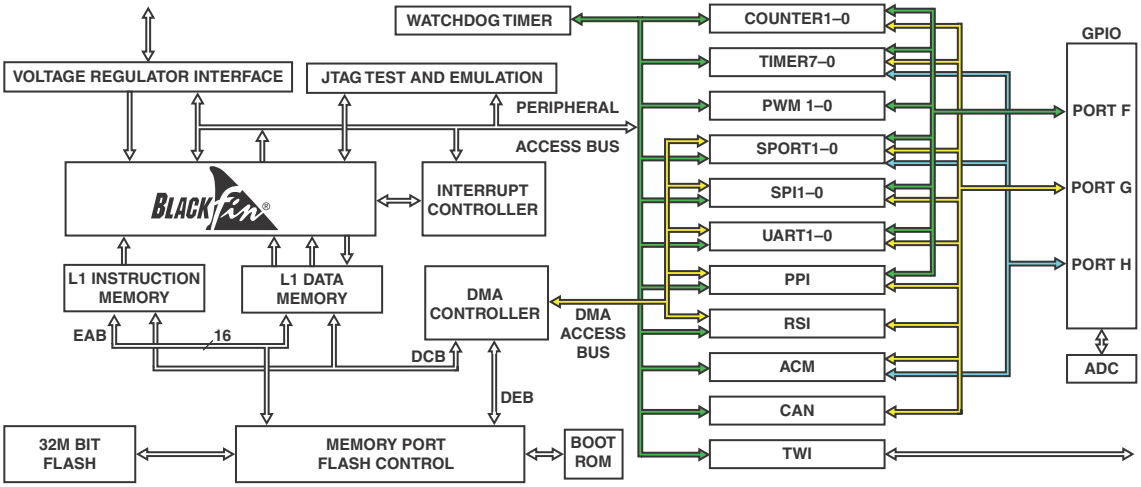


Figure 1-1. ADSP-BF50x Processor Block Diagram

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this

address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems. Table 1-2 shows the memory for the ADSP-BF50x processors.

Table 1-2. Memory Configurations

Type of Memory	ADSP-BF50x
Instruction SRAM/cache, lockable by way or line	16K byte
Instruction SRAM	16K byte
Data SRAM/cache	16K byte
Data SRAM	16K byte
Data scratchpad SRAM	4K byte
L3 Boot ROM	4K byte
Total	72K byte

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the external bus interface unit (EBIU), provides expansion with flash memory on the ADSP-BF504F and ADSP-BF506F processors.

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

Internal Memory

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

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

External Memory

External memory is accessed via the EBIU memory port. This 16-bit interface provides a glue-less connection to the internal flash memory and boot ROM. The EBIU on the processor interfaces with an internal flash memory on the ADSP-BF504F and ADSP-BF506F devices. The internal chip flash memory is a 32M bit (×16, multiple bank, burst) memory. The features of this memory include:

- Synchronous/asynchronous read
- Synchronous burst read mode: 50 MHz
- Asynchronous/synchronous read mode
- Random access times: 70 ns
- Synchronous burst read suspend
- Memory blocks
- Multiple bank memory array: 4 Mbit banks

- Parameter blocks (top location)
- Dual operations
- Program erase in one bank while read in others
- No delay between read and write operations
- Block locking
- All blocks locked at power-up
- Any combination of blocks can be locked or locked down
- Security
- 128-bit user programmable OTP cells
- 64-bit unique device number
- Common flash interface (CFI)
- 100 000 program/erase cycles per block

Flash memory ships from the factory in an erased state except for block 0 of the parameter bank. Block 0 of the flash memory parameter bank ships from the factory in an unknown state. An erase operation should be performed prior to programming this block.

I/O Memory Space

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

DMA Support

The processor has multiple, independent DMA channels that support automated data transfers with minimal overhead for the processor core. DMA transfers can occur between the processor's internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interface. DMA-capable peripherals include the SPORTs, SPI ports, UARTs, RSI, and PPI. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

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

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

Examples of DMA types supported by the processor DMA controller include:

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

In addition to the dedicated peripheral DMA channels, there are two memory DMA channels, which are provided for transfers between the various memories of the processor system with minimal processor intervention. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

General-Purpose I/O (GPIO)

Because of the rich set of peripherals, the processor groups the many peripheral signals to three ports—Port F, Port G, and Port H. Most of the associated pins are shared by multiple signals. The ports function as multiplexer controls.

The processor has 35 bidirectional, general-purpose I/O (GPIO) pins allocated across three separate GPIO modules—PORTFIO, PORTGIO, and PORTHIO, associated with Port F, Port G, and Port H, respectively. Each GPIO-capable pin shares functionality with other processor peripherals via a multiplexing scheme; however, the GPIO functionality is the default state of the device upon power-up. Neither GPIO output nor input drivers are active by default. Each general-purpose port pin can be individually controlled by manipulation of the port control, status, and interrupt registers:

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

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

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

Two-Wire Interface

The Two-Wire Interface (TWI) is fully compatible with the widely used I^2C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth, the TWI controller can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I^2C bus protocol. The *Philips I^2C Bus Specification version 2.1* covers many variants of I^2C. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master data arbitration
- 7-bit addressing
- 100K bits/second and 400K bit/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up
- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*

RSI Interface

The removable storage interface (RSI) controller acts as the host interface for multi-media cards (MMC), secure digital memory cards (SD Card), secure digital input/output cards (SDIO), and CE-ATA hard disk drives.

The following list describes the main features of the RSI controller:

- Support for a single MMC, SD memory, SDIO card or CE-ATA hard disk drive
- Support for 1-bit and 4-bit SD modes
- Support for 1-bit, 4-bit and 8-bit MMC modes
- Support for 4-bit and 8-bit CE-ATA hard disk drives
- A ten-signal external interface with clock, command, and up to eight data lines
- Card detection using one of the data signals
- Card interface clock generation from SCLK
- SDIO interrupt and read wait features
- CE-ATA command completion signal recognition and disable

General-Purpose (GP) Counter

Two 32-bit GP counters are provided that can sense 2-bit quadrature or binary codes as typically emitted by industrial drives or manual thumb wheels. Each counter can also operate in general-purpose up/down count modes. Then, count direction is either controlled by a level-sensitive input signal or by two edge detectors. A third input can provide flexible zero marker support and can alternatively be used to input the push-button signal of thumb wheels. All three signals have a programmable debouncing circuit. An internal signal forwarded to the GP timer unit enables one timer to measure the intervals between count events. Boundary registers enable auto-zero operation or simple system warning by interrupts when programmable count values are exceeded.

3-Phase PWM Unit

The processors integrate two flexible and programmable 3-phase PWM waveform generators that can each be programmed to generate the required switching patterns to drive a 3-phase voltage source inverter for ac induction (ACIM) or permanent magnet synchronous (PMSM) motor control. In addition, each PWM block contains special functions that considerably simplify the generation of the required PWM switching patterns for control of the electronically commutated motor (ECM) or brushless dc motor (BDCM). Software can enable a special mode for switched reluctance motors (SRM).

Features of each 3-phase PWM generation unit are:

- 16-bit center-based PWM generation unit
- Programmable PWM pulse width
- Single/double update modes
- Programmable dead time and switching frequency
- Twos-complement implementation which permits smooth transition to full ON and full OFF states
- Possibility to synchronize the PWM generation to an external synchronization
- Special provisions for BDCM operation (crossover and output enable functions)
- Wide variety of special switched reluctance (SR) operating modes
- Output polarity and clock gating control
- Dedicated asynchronous PWM shutdown signal

The six PWM output signals in each PWM controller consist of three high-side drive signals (PWM_AH, PWM_BH, and PWM_CH) and three low-side drive signals (PWM_AL, PWM_BL, and PWM_CL). The polarity of the generated PWM signal can be set with software, so that either active high or active low PWM patterns can be produced. The switching frequency of the generated PWM pattern is programmable. The PWM generator can operate in single update mode or double update mode. In single update mode the duty cycle values are programmable only once per PWM period, so that the resultant PWM patterns are symmetrical about the midpoint of the PWM period. In the double update mode, a second updating of the PWM registers is implemented at the midpoint of the PWM period. In this mode, it is possible to produce asymmetrical PWM patterns that produce lower harmonic distortion in 3-phase PWM inverters.

Parallel Peripheral Interface

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

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

Three distinct ITU-R 656 modes are supported:

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

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

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

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

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

SPORT Controllers

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

- Bidirectional, I^2S capable operation

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

- Buffered (eight-deep) transmit and receive ports

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

- Clocking

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

- Word length

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

- Framing

  Each transmit and receive port can run with or without frame sync signals for each data word. Frame sync signals can be generated internally or externally, active high or low, and with either of two pulse widths and early or late frame sync.

- Companding in hardware

  Each SPORT can perform A-law or μ-law companding according to ITU recommendation G.711. Companding can be selected on the transmit and/or receive channel of the SPORT without additional latencies.

- DMA operations with single cycle overhead

  Each SPORT can automatically receive and transmit multiple buffers of memory data. The processor can link or chain sequences of DMA transfers between a SPORT and memory.

- Interrupts

  Each transmit and receive port generates an interrupt upon completing the transfer of a data word or after transferring an entire data buffer or buffers through DMA.

- Multichannel capability

  Each SPORT supports 128 channels out of a 1024-channel window and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.

Serial Peripheral Interface (SPI) Ports

The processor has two SPI-compatible ports that enable the processor to communicate with multiple SPI-compatible devices.

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

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

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

Timers

There are nine general-purpose programmable timer units in the processor. Eight timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the TMRCLK/PPI_CLK pin or to the internal SCLK.

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

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

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

UART Ports

The ADSP-BF50x Blackfin processors provide two full-duplex universal asynchronous receiver/transmitter (UART) ports. Each UART port provides a simplified UART interface to other peripherals or hosts, enabling full-duplex, DMA-supported, asynchronous transfers of serial data. A UART port includes support for five to eight data bits, one or two stop bits, and none, even, or odd parity. Each UART port supports two modes of operation:

- PIO (programmed I/O). The processor sends or receives data by writing or reading I/O-mapped UART registers. The data is double-buffered on both transmit and receive.
- DMA (direct memory access). The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. Each UART has two dedicated DMA channels, one for transmit and one for receive. These DMA channels have lower default priority than most DMA channels because of their relatively low service rates. Flexible interrupt timing options are available on the transmit side.

Each UART port's baud rate, serial data format, error code generation and status, and interrupts are programmable:

- Supporting bit rates ranging from (f_{SCLK}/1,048,576) to (f_{SCLK}) bits per second.
- Supporting data formats from 7 to 12 bits per frame.
- Both transmit and receive operations can be configured to generate maskable interrupts to the processor.

The UART port's clock rate is calculated as

$$UART\ Clock\ Rate = \frac{f_{SCLK}}{16^{(1-EDBO)} \times UART\_Divisor}$$

Where the 16-bit UART divisor comes from the UARTx_DLH register (most significant 8 bits) and UARTx_DLL register (least significant eight bits), and the EDBO is a bit in the UARTx_GCTL register.

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

The UARTs feature a pair of $\overline{\text{UAx\_RTS}}$ (request to send) and $\overline{\text{UAx\_CTS}}$ (clear to send) signals for hardware flow purposes. The transmitter hardware is automatically prevented from sending further data when the $\overline{\text{UAx\_CTS}}$ input is deasserted. The receiver can automatically deassert its $\overline{\text{UAx\_RTS}}$ output when the enhanced receive FIFO exceeds a certain high-water level. The capabilities of the UARTs are further extended with support for the Infrared Data Association (IrDA®) Serial Infrared Physical Layer Link Specification (SIR) protocol.

Controller Area Network (CAN) Interface

The ADSP-BF50x processors provide a CAN controller that is a communication controller implementing the Controller Area Network (CAN) V2.0B protocol. This protocol is an asynchronous communications protocol used in both industrial and automotive control systems. CAN is well suited for control applications due to its capability to communicate reliably over a network since the protocol incorporates CRC checking, message error tracking, and fault node confinement.

The CAN controller is based on a 32-entry mailbox RAM and supports both the standard and extended identifier (ID) message formats specified in the CAN protocol specification, revision 2.0, part B.

Each mailbox consists of eight 16-bit data words. The data is divided into fields, which includes a message identifier, a time stamp, a byte count, up to 8 bytes of data, and several control bits. Each node monitors the messages being passed on the network. If the identifier in the transmitted message matches an identifier in one of its mailboxes, the module knows that the message was meant for it, passes the data into its appropriate mailbox, and signals the processor of message arrival with an interrupt.

The CAN controller can wake up the processor from sleep mode upon generation of a wake-up event, such that the processor can be maintained in a low-power mode during idle conditions. Additionally, a CAN wake-up event can wake up the on-chip internal voltage regulator from the powered-down hibernate state.

The electrical characteristics of each network connection are very stringent. Therefore, the CAN interface is typically divided into two parts: a controller and a transceiver. This allows a single controller to support different drivers and CAN networks. The ADSP-BF50x CAN module represents the controller part of the interface. This module's network I/O

is a single transmit output and a single receive input, which connect to a line transceiver.

The CAN clock is derived from the processor system clock (SCLK) through a programmable divider and therefore does not require an additional crystal.

ACM Interface

The ADC control module (ACM) provides an interface that synchronizes the controls between the processor and analog-to-digital converter (ADC) modules like the internal ADC of the ADSP-BF506F. The analog-to-digital conversions are initiated by the processor, based on external or internal events.

The ACM allows for flexible scheduling of sampling instants and provides precise sampling signals to the ADC. The ACM synchronizes the ADC conversion process; generating the ADC controls, the ADC conversion start signal, and other signals. The actual data acquisition from the ADC is done by SPORT peripherals.

Internal ADC

The ADSP-BF506F processor includes an ADC. All internal ADC signals are connected out to package pins to enable maximum flexibility in mixed signal applications.

The internal ADC is a dual, 12-bit, high speed, low power, successive approximation ADC that operates from a single 2.7 V to 5.25 V power supply and features throughput rates up to 2 MSPS. The device contains two ADCs, each preceded by a 3-channel multiplexer, and a low noise, wide bandwidth track-and-hold amplifier that can handle input frequencies in excess of 30 MHz.

Watchdog Timer

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

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

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

Clock Signals

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

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

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip Phase Locked Loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (0.5× to

64×) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 6×, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

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

Dynamic Power Management

The processor provides five operating modes, each with a different performance/power profile. In addition, dynamic power management provides the control functions to dynamically alter the processor core supply voltage, further reducing power dissipation. When configured for a 0 volt core supply voltage, the processor enters the hibernate state. Control of clocking to each of the processor peripherals also reduces power consumption. See Table 1-3 for a summary of the power settings for each mode.

Full-On Operating Mode—Maximum Performance

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

Active Operating Mode—Moderate Dynamic Power Savings

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

In the active mode, it is possible to disable the control input to the PLL by setting the PLL_OFF bit in the PLL control register. This register can be accessed with a user-callable routine in the on-chip ROM called bfrom_SysControl(). If disabled, the PLL control input must be re-enabled before transitioning to the full-on or sleep modes.

Table 1-3. Power Settings

Mode/State	PLL	PLL Bypassed	Core Clock (CCLK)	System Clock (SCLK)	Core Power
Full On	Enabled	No	Enabled	Enabled	On
Active	Enabled/Disabled	Yes	Enabled	Enabled	On
Sleep	Enabled	—	Disabled	Enabled	On
Deep Sleep	Disabled	—	Disabled	Disabled	On
Hibernate	Disabled	—	Disabled	Disabled	Off

For more information about PLL controls, see the "Dynamic Power Management" on page 8-1.

Sleep Operating Mode—High Dynamic Power Savings

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

DMA accesses to L1 memory are not supported in sleep mode.

Deep Sleep Operating Mode—Maximum Dynamic Power Savings

The deep sleep mode maximizes dynamic power savings by disabling the clocks to the processor core (CCLK) and to all synchronous peripherals (SCLK). Asynchronous peripherals may still be running but cannot access internal resources or external memory. Deep sleep mode can be exited only by a hardware reset event, by a wakeup event on a programmable flag pin (including PH0, PF8, or PF9), or by a wakeup event on the programmable flag pin associated with the CAN_RX signal (PG1). A programmable flag event causes the processor to transition to active mode, and execution resumes at the program counter value at which the processor entered deep sleep mode. Assertion of $\overline{\text{RESET}}$ while in deep sleep mode causes the processor to transition to the full on mode.

Hibernate State—Maximum Static Power Savings

The hibernate state maximizes static power savings by disabling the voltage and clocks to the processor core (CCLK) and to all of the peripherals (SCLK). This setting sets the internal power supply voltage (V_{DDINT}) to 0 V to provide the lowest static power dissipation. Any critical information stored internally (for example, memory contents, register contents, and other information) must be written to a non-volatile storage device prior to removing power if the processor state is to be preserved. Writing 0 to the HIBERNATEB bit causes EXT_WAKE to transition low, which can be used to signal an external voltage regulator to shut down.

Since V_{DDEXT} can still be supplied in this mode, all of the external pins three-state, unless otherwise specified. This allows other devices that may be connected to the processor to still have power applied without drawing unwanted current.

The processor can be woken up by asserting the $\overline{\text{RESET}}$ pin. All hibernate wakeup events initiate the hardware reset sequence. Individual sources are

enabled by the VR_CTL register. The EXT_WAKE signal indicates the occurrence of a wakeup event.

As long as V_{DDEXT} is applied, the VR_CTL register maintains its state during hibernation. All other internal registers and memories, however, lose their content in the hibernate state.

Instruction Set Description

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

The assembly language, which takes advantage of the processor's unique architecture, offers these advantages:

- Embedded 16/32-bit microcontroller features, such as arbitrary bit and bit field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data types; and separate user and supervisor stack pointers
- Seamlessly integrated DSP/CPU features optimized for both 8-bit and 16-bit operations

- A multi-issue load/store modified Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle
- All registers, I/O, and memory mapped into a unified 4G byte memory space, providing a simplified programming model

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

Development Tools

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

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

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

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

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

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

The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning, when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, pre-emptive,

cooperative and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system.

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

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

In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processor family. Hardware tools include the ADSP-BF50x EZ-KIT Lite standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 MEMORY

This chapter discusses memory population specific to the ADSP-BF50x processors. Functional memory architecture is described in the *Blackfin Processor Programming Reference*.

Memory Architecture

Figure 2-1 provides an overview of the ADSP-BF50x processor system memory map. For a detailed discussion of how to use them, see the *Blackfin Processor Programming Reference*.

Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

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

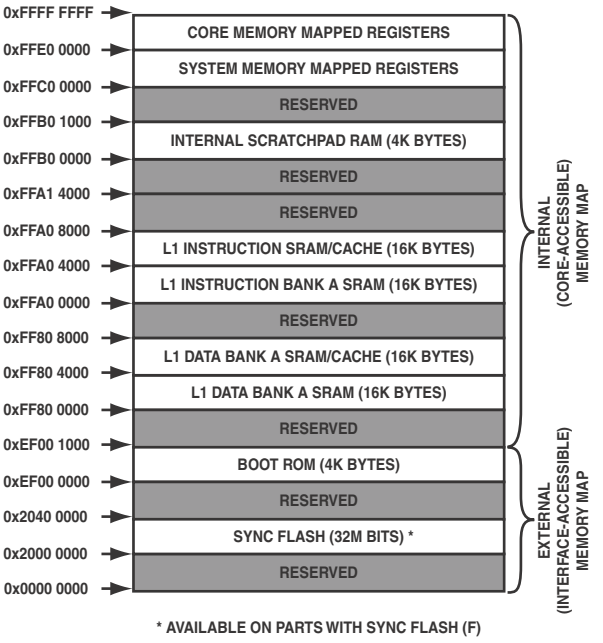


Figure 2-1. ADSP-BF50x Memory Map

L1 Instruction SRAM

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

Table 2-1 lists the memory start locations of the L1 instruction memory subbanks.

Table 2-1. L1 Instruction Memory Subbanks

Memory Subbank	Memory Start Location for ADSP-BF50x Processors
0	0xFFA0 0000
1	0xFFA0 1000
2	0xFFA0 2000
3	0xFFA0 3000
4	0xFFA0 4000
5	0xFFA0 5000
6	0xFFA0 6000
7	0xFFA0 7000

L1 Data SRAM

Table 2-2 shows how the subbank organization is mapped into memory.

Table 2-2. L1 Data Memory SRAM Subbank Start Addresses

Memory Bank and Subbank	ADSP-BF50x Processors
Data Bank A, Subbank 0	0xFF80 0000
Data Bank A, Subbank 1	0xFF80 1000
Data Bank A, Subbank 2	0xFF80 2000
Data Bank A, Subbank 3	0xFF80 3000
Data Bank A, Subbank 4	0xFF80 4000
Data Bank A, Subbank 5	0xFF80 5000
Data Bank A, Subbank 6	0xFF80 6000
Data Bank A, Subbank 7	0xFF80 7000

L1 Data Cache

When data cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), 16K byte of data bank A can be set to serve as cache.

Boot ROM

There are 4K bytes of memory space occupied by the boot ROM, starting from address 0xEF00 0000. This 16-bit boot ROM is not part of the L1 memory module. Read accesses take one SCLK cycle and no wait states are required. The read-only memory can be read by the core as well as by DMA. It can be cached and protected by CPLD blocks like external memory. The boot ROM not only contains boot-strap loader code, it also provides some subfunctions that are user-callable at runtime. For more information, see "System Reset and Booting" in Chapter 24, System Reset and Booting.

External Memory

External memory (shown in Figure 2-1) is accessed via the EBIU memory port. This 16-bit interface provides a glue-less connection to the internal flash memory (on ADSP-BF504F and ADSP-BF506F devices) and boot ROM. Internal flash memory ships from the factory in an erased state except for block 0 of the parameter bank.

Block 0 of the flash memory parameter bank ships from the factory in an unknown state. An erase operation should be performed prior to programming this block.

Processor-Specific MMRs

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

DMEM_CONTROL Register

The data memory control register (DMEM_CONTROL), shown in Figure 2-2, contains control bits for the L1 data memory.

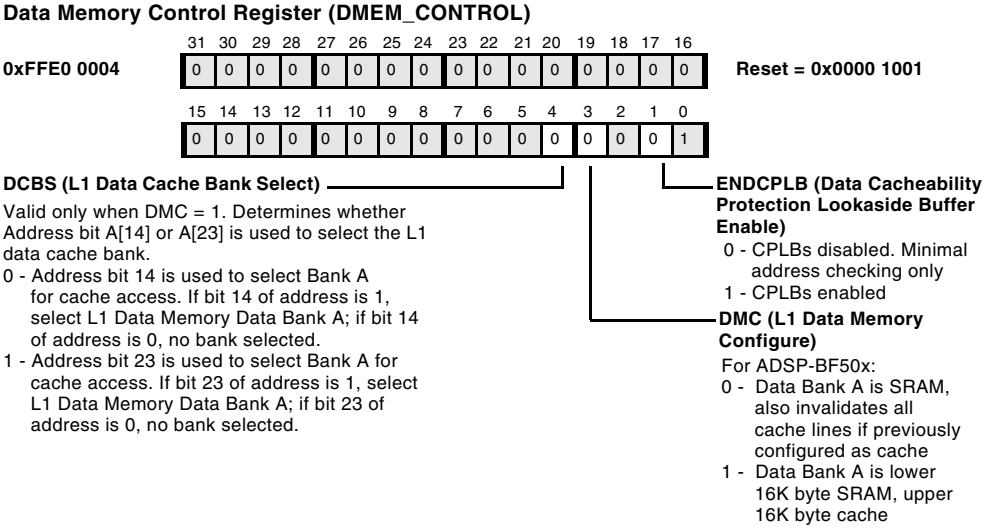


Figure 2-2. L1 Data Memory Control Register

Note that both DAG 0 and 1 use Port-A for non-cacheable fetches.

DTEST_COMMAND Register

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the data test data registers (DTEST DATA[1:0]). This register is shown in Figure 2-3.

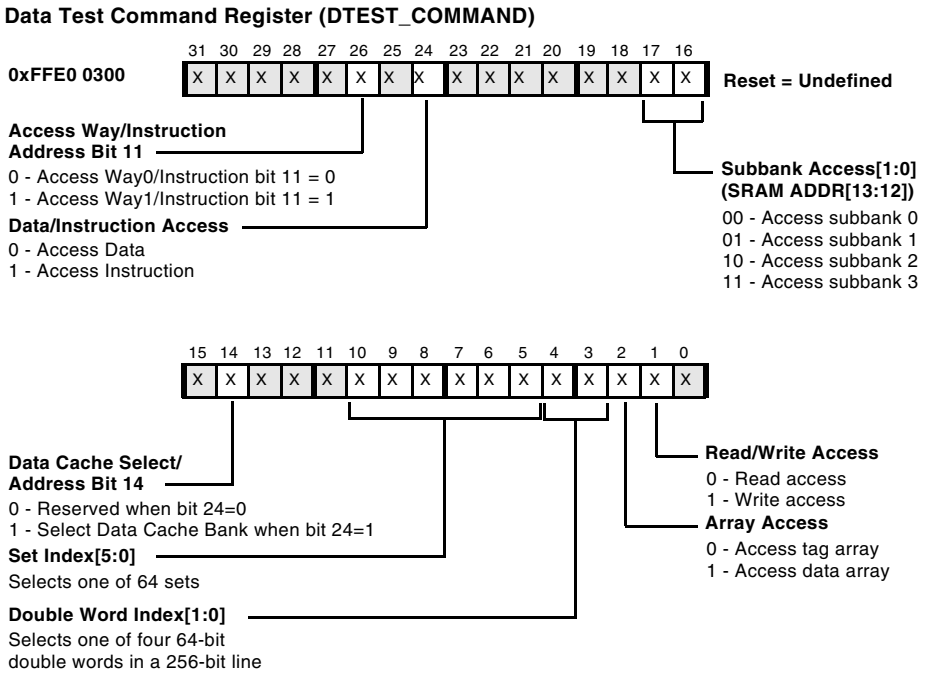


Figure 2-3. Data Test Command Register

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

Note that ITEST_COMMAND must be used to access to L1 Instruction SRAM from 0xFA00 4000 to 0xFFA0 7FFF.

3 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, how data moves through the system, and other factors that determine the system organization. Following an overview and a list of key features is a block diagram of the chip bus hierarchy and a description of its operation. The chapter concludes with details about the system interconnects and associated system buses.

This chapter provides

Chip Bus Hierarchy Overview

ADSP-BF50x Blackfin processors feature a powerful chip bus hierarchy on which all data movement between the processor core, internal memory, external memory, and its rich set of peripherals occurs. The chip bus hierarchy includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock domain conversion is provided to support clock domain transactions between the core and the system.

The processor system includes:

- The peripheral set including GP timers and counters, ACM, TWI, RSI, UARTs, SPORTs, SPIs, PPI, watchdog timer, and PWM units. The ADSP-BF506F processor peripherals include an ADC and a flash memory, and the ADSP-BF504F processor peripherals include a flash memory (but does not include an ADC).
- The External Bus Interface Unit (EBIU)
- The Direct Memory Access (DMA) controller
- The interfaces between these, the system, and the optional external (off-chip) resources

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

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

Interface Overview

Figure 3-1 shows the core processor and system boundaries as well as the interfaces between them.

Internal Clocks

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

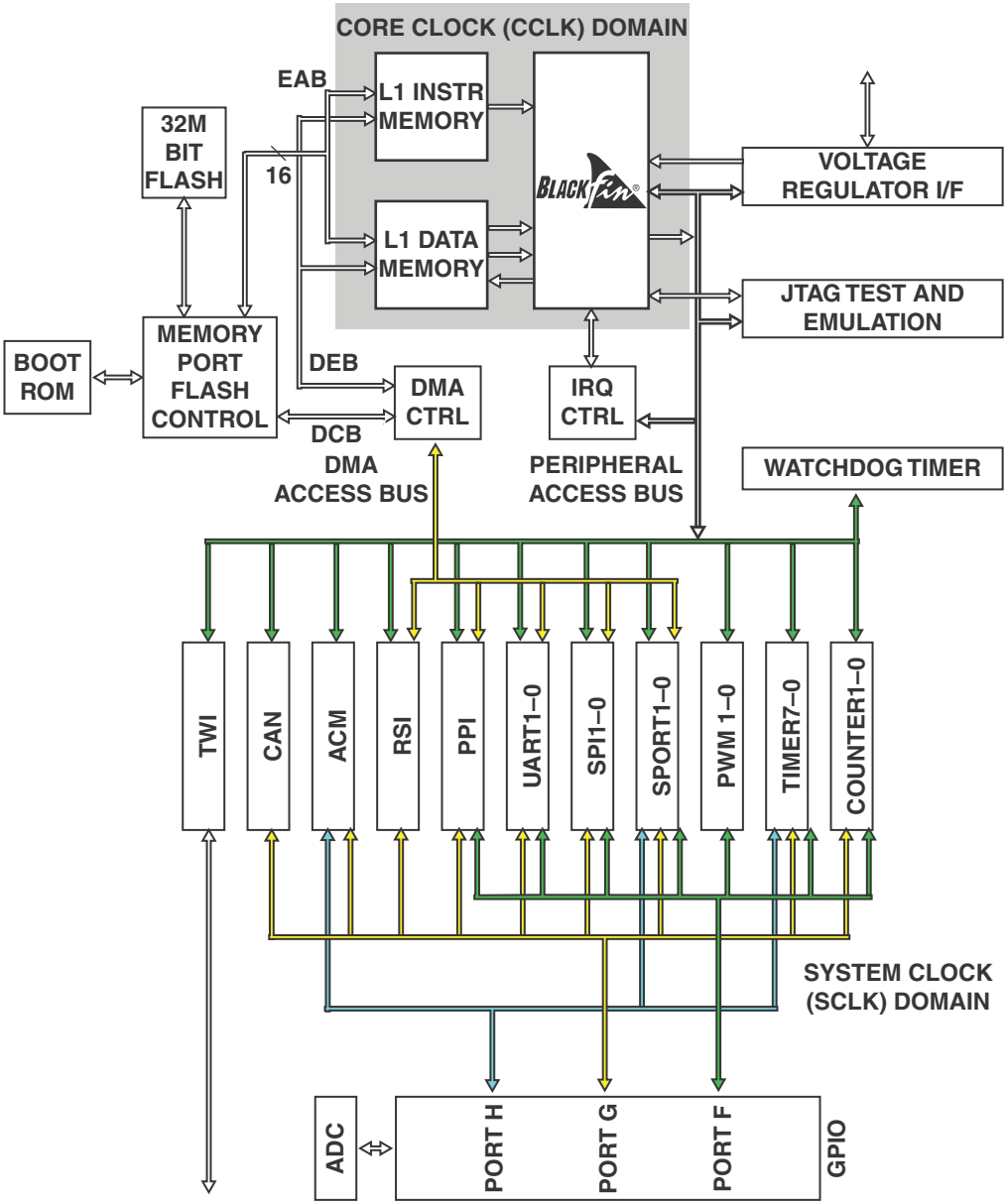


Figure 3-1. Processor Bus Hierarchy

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

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

Core Bus Overview

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

Figure 3-2 shows the core processor and its interfaces to the peripherals and external memory resources.

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

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

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

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

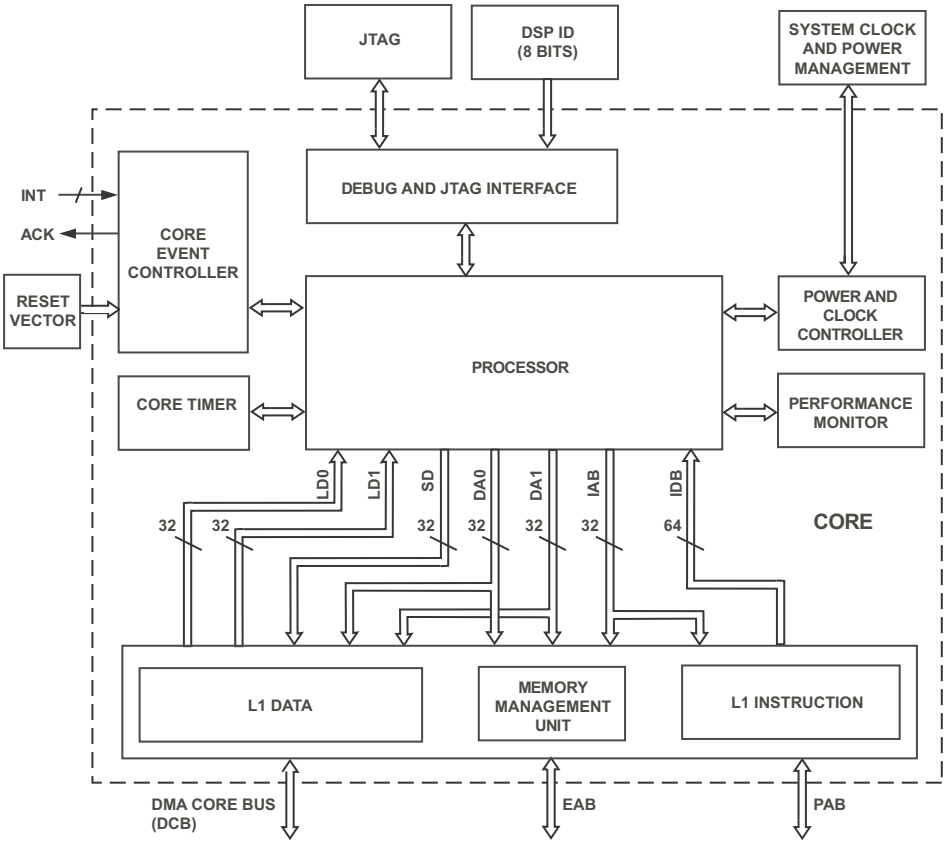


Figure 3-2. Core Block Diagram

Peripheral Access Bus (PAB)

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

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

PAB Arbitration

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

PAB Agents (Masters, Slaves)

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

The slaves on the PAB bus are:

- System event controller
- Clock and power management controller
- Watchdog timer
- Counters
- Timer 0–7
- SPORT0–1
- SPI0–1
- Ports
- UART0–1
- PPI
- TWI

- ACM
- PWM
- RSI
- DMA controller

PAB Performance

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

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

The PAB has a maximum frequency of SCLK.

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

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

DAB, DCB, and DEB Arbitration

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

The DCB has priority over the core processor on arbitration into L1 configured as SRAM. For external memory (flash memory on ADSP-BF50xF processors), the core (by default) has priority over the DEB for accesses to the EPB. The processor has a programmable priority arbitration policy on the DAB. Table 3-1 shows the default arbitration priority. In addition, by setting the CDPRIO bit in the EBIU_AMGCTL register, all DEB transactions to the EPB have priority over core accesses to external memory. Use of this bit is application-dependent. For example, if you are polling a peripheral mapped to asynchronous memory with long access times, by default the core will "win" over DMA requests. By setting the CDPRIO bit, the core would be held off until DMA requests were serviced.

Table 3-1. DAB, DCB, and DEB Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
PPI receive or transmit	0 - highest
RSI receive or transmit	1
SPORT0 receive	2
SPORT0 transmit	3
SPORT1 receive	4
SPORT1 transmit	5
SPI0 receive or transmit	6
SPI1 receive or transmit	7
UART0 receive	8
UART0 transmit	9
UART1 receive	10
UART1 transmit	11
Mem DMA D0 has no peripheral mapping	None
Mem DMA S0 has no peripheral mapping	None
Mem DMA D1 has no peripheral mapping	None
Mem DMA S1 has no peripheral mapping	None

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 3-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

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

DAB, DCB, and DEB Performance

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

The DAB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4K byte size for L1). If there is a conflict, DMA is the highest priority requester, followed by the core.

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.

DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. Latencies caused by these stalls are in addition to any arbitration latencies.

The core processor and the DAB must arbitrate for access to external memory through the EBIU. This additional arbitration latency added to the latency required to read off-chip memory devices can significantly degrade DAB throughput, potentially causing peripheral data buffers to underflow or overflow. If you use DMA peripherals other than the memory DMA controller, and you target

external memory for DMA accesses, you need to carefully analyze your specific traffic patterns. Make sure that isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

External Access Bus (EAB)

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

Arbitration of the External Bus

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

DEB/EAB Performance

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

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

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

Table 3-2 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations. The numbers in the table are theoretical values. These values may be higher when they are measured on actual hardware due to a variety of reasons relating to the device that is connected to the EBIU.

Table 3-2. Performance of DMA Access to External Memory

Source	Destination	Approximate SCLKs For n Words (from start of DMA to interrupt at end)
16-bit internal burst flash memory[1]	L1 data memory	(s * n + 2) + r * [(x + 1) * n + b * (n - 1)] where: • **s** is the number of setup/hold SCLK cycles (minimum of 2) • **r** is the ratio BCLK/SCLK (BCLK is the burst clock frequency of the internal parallel burst flash) • **x** is the number of wait states cycles in NORCLK • **b** is the number of NORCLK cycles between bursts
L1 data memory	L1 data memory	2n + 12

1 For ADSP-BF50xF (flash memory) products only.

4 SYSTEM INTERRUPTS

This chapter discusses the system interrupt controller (SIC). While this chapter does refer to features of the core event controller (CEC), it does not cover all aspects of it. Please refer to the *Blackfin Processor Programming Reference* for more information on the CEC.

Specific Information for the ADSP-BF50x

For details regarding the number of system interrupts for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

To determine how each of the system interrupts is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Chapter A, "System MMR Assignments".

System interrupt behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF50x Processor" on page 4-15.

Overview

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

Features

The Blackfin architecture provides a two-level interrupt processing scheme:

- The core event controller (CEC) runs in the CCLK clock domain. It interacts closely with the program sequencer and manages the event vector table (EVT). The CEC processes not only core-related interrupts such as exceptions, core errors, and emulation events; it also supports software interrupts.
- The system interrupt controller (SIC) runs in the SCLK clock domain. It masks, groups, and prioritizes interrupt requests signalled by on-chip or off-chip peripherals and forwards them to the CEC.

Description of Operation

The following sections describe the operation of the system interrupts.

Events and Sequencing

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

The CEC of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)

- Exceptions
- Interrupts

Note the word *event* describes all five types of activities. The CEC manages fifteen different events in all: emulation, reset, NMI, exception, and eleven interrupts.

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

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

The CEC supports nine general-purpose interrupts (`IVG7` – `IVG15`) in addition to the dedicated interrupt and exception events that are described in Table 4-1. It is common for applications to reserve the lowest or the two lowest priority interrupts (`IVG14` and `IVG15`) for software interrupts, leaving eight or seven prioritized interrupt inputs (`IVG7` – `IVG13`) for peripheral purposes. Refer to Table 4-1.

Table 4-1. System and Core Event Mapping

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

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

Event Source	Core Event Name
System interrupts	IVG7–IVG13
Software interrupt 1	IVG14
Software interrupt 2 (lowest priority)	IVG15

System Peripheral Interrupts

To service the rich set of peripherals, the SIC has multiple interrupt request inputs and outputs that go to the CEC. The primary function of the SIC is to mask, group, and prioritize interrupt requests and to forward them to the nine general-purpose interrupt inputs of the CEC (`IVG7`–`IVG15`). Additionally, the SIC controller can enable individual peripheral interrupts to wake up the processor from Idle or power-down state.

The nine general-purpose interrupt inputs (`IVG7`–`IVG15`) of the core event controller have fixed priority. Of this group, the `IVG7` channel has the highest priority and `IVG15` has the lowest priority. Therefore, the interrupt assignment in the `SIC_IAR` registers not only groups peripheral interrupts; it also programs their priority by assigning them to individual IVG channels. However, the relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the `SIC_IAR` register settings shown in Figure 4-2 on page 4-11 and the tables in Chapter A, "System MMR Assignments". If more than one interrupt source is mapped to the same interrupt, they are logically OR'ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

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

The core timer has a dedicated input to the CEC controller. Its interrupt is not routed through the SIC controller and always has higher priority than requests from all peripherals.

The SIC_IMASK register allows software to mask any peripheral interrupt source at the SIC level. This functionality is independent of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of the SIC_IMASK register are all 0s to mask off all peripheral interrupts. Turning off a system interrupt mask and enabling the particular interrupt is performed by writing a 1 to a bit location in the SIC_IMASK register.

The SIC includes one or more read-only SIC_ISR registers with individual bits which correspond to the interrupt status of one of the peripheral interrupt sources. When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input has been deasserted, the respective bit in the system interrupt status register is cleared. Note for some peripherals, such as general-purpose I/O asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt has been deasserted.

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

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

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

The SIC_ISR register is not affected by the state of the SIC_IMASK register and can be read at any time. Writes to the SIC_ISR register have no effect on its contents.

Peripheral DMA channels are mapped in a fixed manner to the peripheral interrupt IDs. However, the assignment between peripherals and DMA channels is freely programmable with the DMA_PERIPHERAL_MAP registers. Table 4-1 on page 4-3 and Table 4-2 on page 4-11 show the default DMA assignment. Once a peripheral has been assigned to any other DMA channel it uses the new DMA channel's interrupt ID regardless of whether DMA is enabled or not. Therefore, clean DMA_PERIPHERAL_MAP management is required even if the DMA is not used. The default setup should be the best choice for all non-DMA applications.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the SIC_IWR register (refer to Table 4-1 on page 4-3 and Table 4-2 on page 4-11). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see the Dynamic Power Management chapter.

The SIC_IWR register has no effect unless the core is idled. By default, all interrupts generate a wakeup request to the core. However, for some

applications it may be desirable to disable this function for some peripherals, such as for a SPORT transmit interrupt. The SIC_IWR register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

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

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 4-2 on page 4-11.

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

Programming Model

The programming model for the system interrupts is described in the following sections.

System Interrupt Initialization

If the default peripheral-to-IVG assignments shown in Table 4-1 on page 4-3 and Table 4-2 on page 4-11 are acceptable, then interrupt initialization involves only:

- Initialization of the core event vector table (EVT) vector address entries
- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts that the system requires in the SIC_IMASK register

System Interrupt Processing Summary

Referring to Figure 4-1 on page 4-10, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IAR registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 - IVG15), determine the core priority of interrupt A.
5. ILAT adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.

6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's interrupt service routine (ISR).
8. When the event vector for interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.
9. When the interrupt service routine (ISR) for interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

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

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

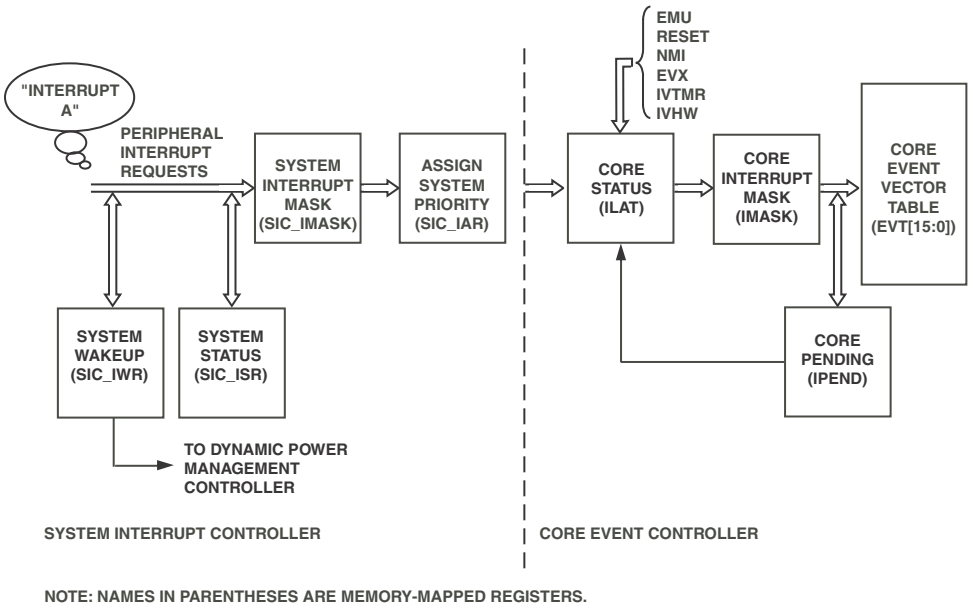


Figure 4-1. Interrupt Processing Block Diagram

System Interrupt Controller Registers

The SIC registers are described in the following sections.

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

System Interrupt Assignment (SIC_IAR) Register

The SIC_IAR register maps each peripheral interrupt ID to a corresponding IVG priority level. This is accomplished with 4-bit groupings that translate to IVG levels as shown in Table 4-2 and Figure 4-2. In other words, Table 4-2 defines the value to write in a 4-bit field within SIC_IAR in order to configure a peripheral interrupt ID for a particular IVG priority. Refer to Table 4-1 on page 4-3 for information on SIC_IAR mappings for this specific processor.

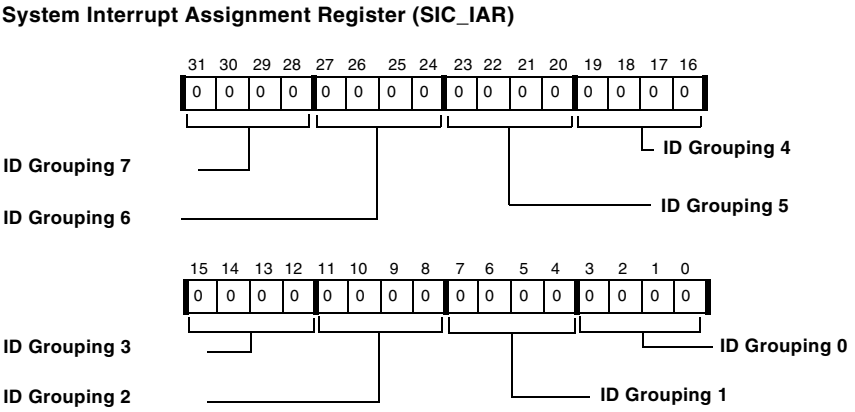


Figure 4-2. System Interrupt Assignment Register

Table 4-2. IVG Select Definitions

General-Purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5

Table 4-2. IVG Select Definitions (Cont'd)

General-Purpose Interrupt	Value in SIC_IAR
IVG13	6
IVG14	7
IVG15	8

System Interrupt Mask (SIC_IMASK) Register

The SIC_IMASK register masks or enables peripheral interrupts at the system level. A "0" in a bit position masks off (disables) interrupts for that particular peripheral interrupt ID. A "1" enables interrupts for that interrupt ID. Refer to Table 4-1 on page 4-3 and Table 4-2 for information on how peripheral interrupt IDs are mapped to the SIC_IMASK register(s) for this particular processor.

System Interrupt Status (SIC_ISR) Register

The SIC_ISR register keeps track of system interrupts that are asserted but not yet serviced. A "0" in a bit position indicates that a particular interrupt is deasserted. A "1" indicates that it is asserted. Refer to Table 4-1 on page 4-3 and Table 4-2 for information on how peripheral interrupt IDs are mapped to the SIC_ISR register(s) for this particular processor.

System Interrupt Wakeup-Enable (SIC_IWR) Register

The SIC_IWR register allows an interrupt request to wake up the processor core from an idled state. A "0" in a bit position indicates that a particular peripheral interrupt ID is not configured to wake the core (upon assertion of the interrupt request). A "1" indicates that it is configured to do so. Refer to Table 4-1 on page 4-3 and Table 4-2 for information on how

peripheral interrupt IDs are mapped to the SIC_IWR register(s) for this particular processor.

Programming Examples

The following section provides an example for servicing interrupt requests.

Clearing Interrupt Requests

When the processor services a core event it automatically clears the requesting bit in the ILAT register and no further action is required by the interrupt service routine. It is important to understand that the SIC controller does not provide any interrupt acknowledgment feedback mechanism from the CEC controller back to the peripherals. Although the ILAT bits clear in the same way when a peripheral interrupt is serviced, the signalling peripheral does not release its level-sensitive request until it is explicitly instructed by software. If however, the peripheral keeps requesting, the respective ILAT bit is set again immediately and the service routine is invoked again as soon as its first run terminates by an RTI instruction.

Every software routine that services peripheral interrupts must clear the signalling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. Receive interrupts, for example, are cleared when received data is read from the respective buffers. Transmit requests typically clear when software (or DMA) writes new data into the transmit buffers. These implicit acknowledge mechanisms avoid the need for cycle-consuming software handshakes in streaming interfaces. Other peripherals such as timers, GPIOs, and error requests require explicit acknowledge instructions, which are typically performed by efficient W1C (write-1-to-clear) operations.

Listing 4-1 shows a representative example of how a GPIO interrupt request might be serviced.

Listing 4-1. Servicing GPIO Interrupt Request

```
#include <defBF527.h>
/*ADSP-BF527 product is used as an example*/
.section program;
_portg_a_isr:
  /* push used registers */
  [--sp] = (r7:7, p5:5);
  /* clear interrupt request on GPIO pin PG2 */
  /* no matter whether used A or B channel */
  p5.l = lo(PORTGIO_CLEAR);
  p5.h = hi(PORTGIO_CLEAR);
  r7 = PG2;
  w[p5] = r7;

  /* place user code here */

  /* sync system, pop registers and exit */
  ssync;
  (r7:7, p5:5) = [sp++];
  rti;
_portg_a_isr.end:
```

The W1C instruction shown in this example may require several SCLK cycles to complete, depending on system load and instruction history. The program sequencer does not wait until the instruction completes and continues program execution immediately. The SSYNC instruction ensures that the W1C command indeed cleared the request in the GPIO peripheral before the RTI instruction executes. However, the SSYNC instruction does not guarantee that the release of interrupt request has also been recognized by the CEC controller, which may require a few more CCLK cycles depending on the CCLK-to-SCLK ratio. In service routines consisting of a few

instructions only, two SSYNC instructions are recommended between the clear command and the RTI instruction. However, one SSYNC instruction is typically sufficient if the clear command performs in the very beginning of the service routine, or the SSYNC instruction is followed by another set of instructions before the service routine returns. Commonly, a pop-multiple instruction is used for this purpose as shown in Listing 4-1.

The level-sensitive nature of peripheral interrupts enables more than one of them to share the same IVG channel and therefore the same interrupt priority. This is programmable using the assignment registers. Then a common service routine typically interrogates the SIC_ISR register to determine the signalling interrupt source. If multiple peripherals are requesting interrupts at the same time, it is up to the service routine to either service all requests in a single pass or to service them one by one. If only one request is serviced and the respective request is cleared by software before the RTI instruction executes, the same service routine is invoked another time because the second request is still pending. While the first approach may require fewer cycles to service both requests, the second approach enables higher priority requests to be serviced more quickly in a non-nested interrupt system setup.

Unique Information for the ADSP-BF50x Processor

This section describes Interfaces and System Peripheral Interrupts that are unique to the ADSP-BF50x processor.

Interfaces

Figure 4-3 and Figure 4-4 provide an overview of how the individual peripheral interrupt request lines connect to the SIC. These figures show how the seven SIC_IAR registers control the assignment to the nine available peripheral request inputs of the CEC.

The memory-mapped ILAT, IMASK, and IPEND registers are part of the CEC controller.

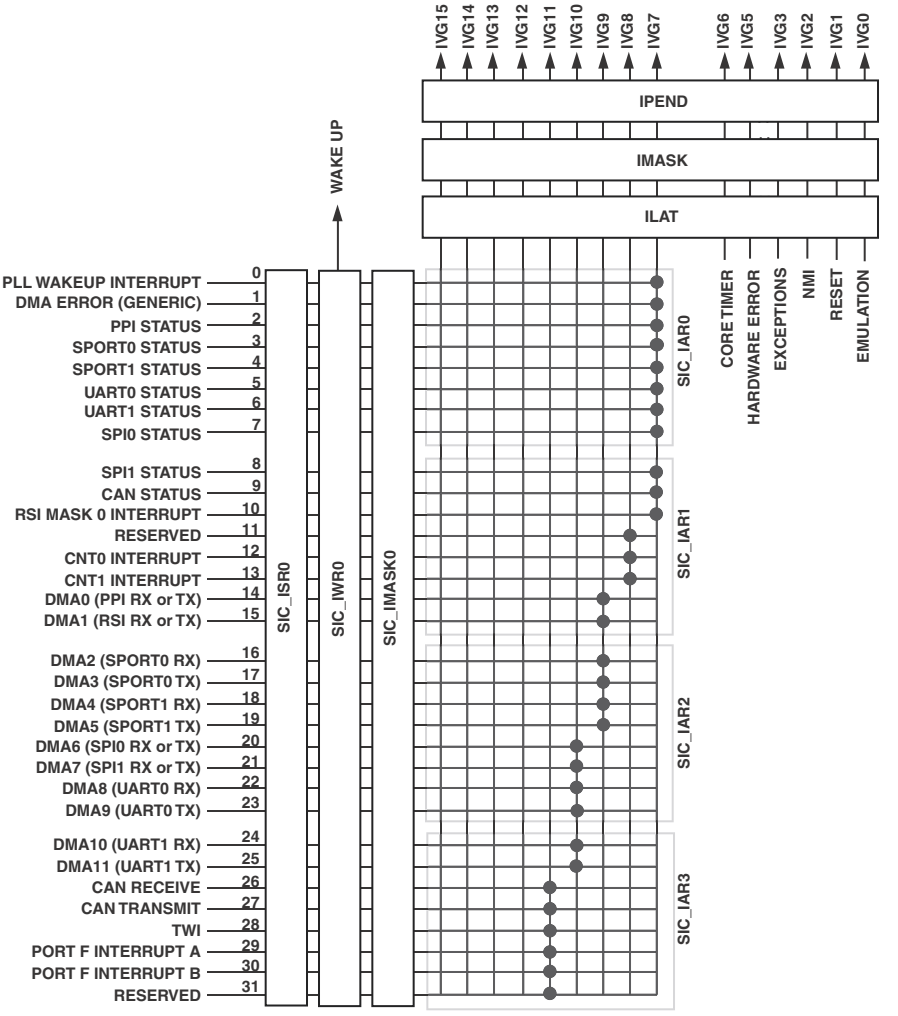


Figure 4-3. Interrupt Routing Overview (Part 1)

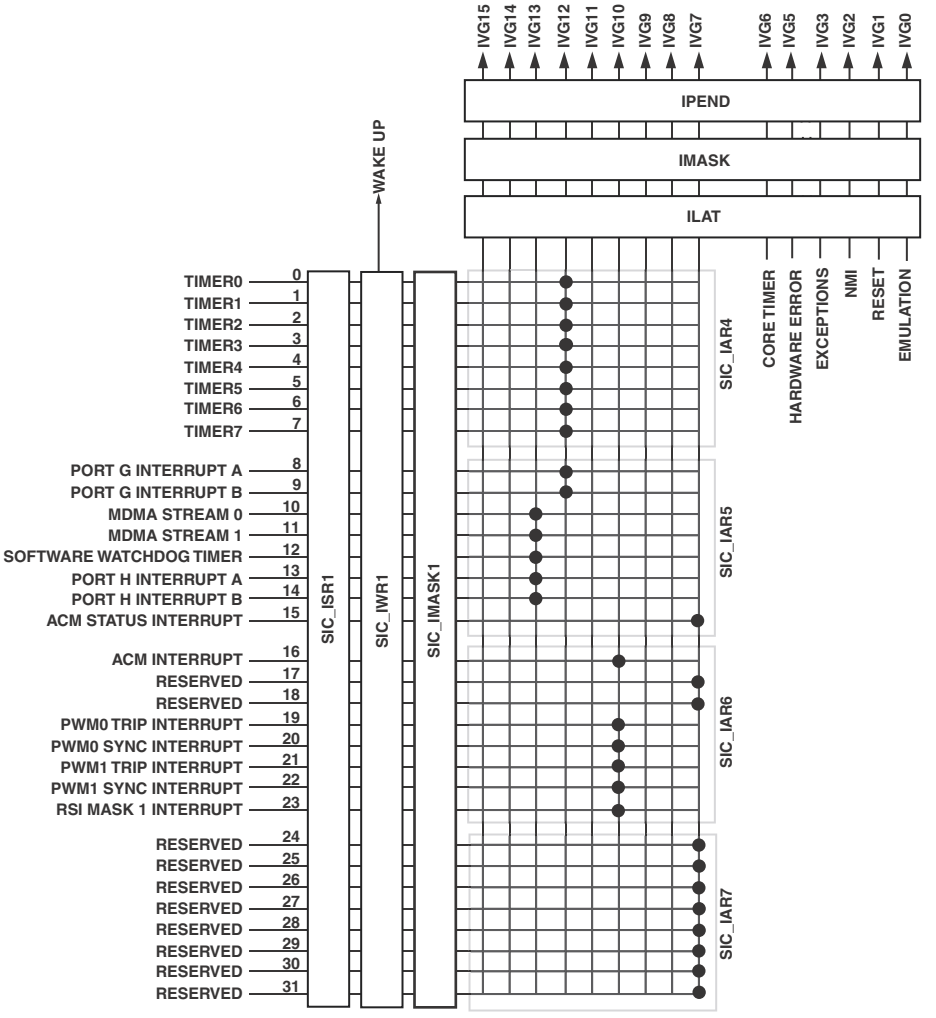


Figure 4-4. Interrupt Routing Overview (Part 2)

System Peripheral Interrupts

Table 4-3 and Table 4-4 show the peripheral interrupt events, the default mapping of each event, the peripheral interrupt ID used in the system interrupt assignment registers (SIC_IAR), and the core interrupt ID.

Note that the system interrupt to core event mappings shown are the default values at reset and can be changed by software. Where there is more than one DMA interrupt source for a given interrupt ID number, the default DMA source mapping is listed first in parentheses.

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 4-3 and Table 4-4.

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

Table 4-3. Peripheral Interrupt Events (Part 1)

Peripheral ID Number	Bit Position for SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3-0	Interrupt Source	Default Mapping
31	Bit 31	SIC_IAR3[31:28]	Reserved	IVG11
30	Bit 30	SIC_IAR3[27:24]	Port F Interrupt B	IVG11
29	Bit 29	SIC_IAR3[23:20]	Port F Interrupt A	IVG11
28	Bit 28	SIC_IAR3[19:16]	TWI	IVG11
27	Bit 27	SIC_IAR3[15:12]	CAN Transmit	IVG11
26	Bit 26	SIC_IAR3[11:8]	CAN Receive	IVG11
25	Bit 25	SIC_IAR3[7:4]	DMA11 (UART1 TX)	IVG10
24	Bit 24	SIC_IAR3[3:0]	DMA10 (UART1 RX)	IVG10
23	Bit 23	SIC_IAR2[31:28]	DMA9 (UART0 TX)	IVG10
22	Bit 22	SIC_IAR2[27:24]	DMA8 (UART0 RX)	IVG10
21	Bit 21	SIC_IAR2[23:20]	DMA7 (SPI1 RX or TX)	IVG10
20	Bit 20	SIC_IAR2[19:16]	DMA6 (SPI0 RX or TX)	IVG10
19	Bit 19	SIC_IAR2[15:12]	DMA5 (SPORT1 TX)	IVG9
18	Bit 18	SIC_IAR2[11:8]	DMA4 (SPORT1 RX)	IVG9
17	Bit 17	SIC_IAR2[7:4]	DMA3 (SPORT0 TX)	IVG9
16	Bit 16	SIC_IAR2[3:0]	DMA2 (SPORT0 RX)	IVG9
15	Bit 15	SIC_IAR1[31:28]	DMA1 (RSI RX or TX)	IVG9
14	Bit 14	SIC_IAR1[27:24]	DMA0 (PPI RX or TX)	IVG9
13	Bit 13	SIC_IAR1[23:20]	CNT1 Interrupt	IVG8
12	Bit 12	SIC_IAR1[19:16]	CNT0 Interrupt	IVG8
11	Bit 11	SIC_IAR1[15:12]	Reserved	IVG8
10	Bit 10	SIC_IAR1[11:8]	RSI Mask 0 Interrupt	IVG7
9	Bit 9	SIC_IAR1[7:4]	CAN Status	IVG7
8	Bit 8	SIC_IAR1[3:0]	SPI1 Status	IVG7

Table 4-3. Peripheral Interrupt Events (Part 1) (Cont'd)

Peripheral ID Number	Bit Position for SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3-0	Interrupt Source	Default Mapping
7	Bit 7	SIC_IAR0[31:28]	SPI0 Status	IVG7
6	Bit 6	SIC_IAR0[27:24]	UART1 Status	IVG7
5	Bit 5	SIC_IAR0[23:20]	UART0 Status	IVG7
4	Bit 4	SIC_IAR0[19:16]	SPORT1 Status	IVG7
3	Bit 3	SIC_IAR0[15:12]	SPORT0 Status	IVG7
2	Bit 2	SIC_IAR0[11:8]	PPI Status	IVG7
1	Bit 1	SIC_IAR0[7:4]	DMA Error (generic)	IVG7
0	Bit 0	SIC_IAR0[3:0]	PLL Wakeup Interrupt	IVG7

Table 4-4. Peripheral Interrupt Events (Part 2)

Peripheral ID Number	Bit Position for SIC_ISR1, SIC_IMASK1, SIC_IWR1	SIC_IAR7–4	Interrupt Source	Default Mapping
63	Bit 31	SIC_IAR7[31:28]	Reserved	IVG7
62	Bit 30	SIC_IAR7[27:24]	Reserved	IVG7
61	Bit 29	SIC_IAR7[23:20]	Reserved	IVG7
60	Bit 28	SIC_IAR7[19:16]	Reserved	IVG7
59	Bit 27	SIC_IAR7[15:12]	Reserved	IVG7
58	Bit 26	SIC_IAR7[11:8]	Reserved	IVG7
57	Bit 25	SIC_IAR7[7:4]	Reserved	IVG7
56	Bit 24	SIC_IAR7[3:0]	Reserved	IVG7
55	Bit 23	SIC_IAR6[31:28]	RSI Mask 1 Interrupt	IVG10
54	Bit 22	SIC_IAR6[27:24]	PWM1 Sync Interrupt	IVG10
53	Bit 21	SIC_IAR6[23:20]	PWM1 Trip Interrupt	IVG10

Table 4-4. Peripheral Interrupt Events (Part 2) (Cont'd)

Peripheral ID Number	Bit Position for SIC_ISR1, SIC_IMASK1, SIC_IWR1	SIC_IAR7–4	Interrupt Source	Default Mapping
52	Bit 20	SIC_IAR6[19:16]	PWM0 Sync Interrupt	IVG10
51	Bit 19	SIC_IAR6[15:12]	PWM0 Trip Interrupt	IVG10
50	Bit 18	SIC_IAR6[11:8]	Reserved	IVG7
49	Bit 17	SIC_IAR6[7:4]	Reserved	IVG7
48	Bit 16	SIC_IAR6[3:0]	ACM Interrupt	IVG10
47	Bit 15	SIC_IAR5[31:28]	ACM Status Interrupt	IVG7
46	Bit 14	SIC_IAR5[27:24]	Port H Interrupt B	IVG13
45	Bit 13	SIC_IAR5[23:20]	Port H Interrupt A	IVG13
44	Bit 12	SIC_IAR5[19:16]	Software Watchdog Timer	IVG13
43	Bit 11	SIC_IAR5[15:12]	MDMA Stream 1	IVG13
42	Bit 10	SIC_IAR5[11:8]	MDMA Stream 0	IVG13
41	Bit 9	SIC_IAR5[7:4]	Port G Interrupt B	IVG12
40	Bit 8	SIC_IAR5[3:0]	Port G Interrupt A	IVG12
39	Bit 7	SIC_IAR4[31:28]	Timer 7	IVG12
38	Bit 6	SIC_IAR4[27:24]	Timer 6	IVG12
37	Bit 5	SIC_IAR4[23:20]	Timer 5	IVG12
36	Bit 4	SIC_IAR4[19:16]	Timer 4	IVG12
35	Bit 3	SIC_IAR4[15:12]	Timer 3	IVG12
34	Bit 2	SIC_IAR4[11:8]	Timer 2	IVG12
33	Bit 1	SIC_IAR4[7:4]	Timer 1	IVG12
32	Bit 0	SIC_IAR4[3:0]	Timer 0	IVG12

5 EXTERNAL BUS INTERFACE UNIT

The external bus interface unit (EBIU) on the ADSP-BF50x Blackfin processors provides glue-less interface to the internal parallel flash memory, which is available on ADSP-BF504F and ADSP-BF506F Blackfin processors, and to the processor boot ROM.

On the ADSP-BF50x Blackfin processors, the parallel synchronous internal flash memory is internal to the product package, but this memory is external to the processor. The interface to the internal flash memory is referred to as the *External Bus Interface Unit*. Despite the *external* in its name, the EBIU does *not* provide access to off-chip memories.

EBIU Overview

The EBIU services requests for the optional parallel internal flash memory and boot ROM memories from the core or from a DMA channel. The priority of the requests is determined by the external bus controller.

The DMA controller provides high-bandwidth data movement capability. The Memory DMA (MDMA) channels can perform block transfers of code or data between the internal L1 SRAM memories and the optional internal parallel flash memory.

The EBIU is clocked by the system clock (`SCLK`). All synchronous memories interfaced to the processor operate at frequencies derived from `SCLK`. The ratio between core clock frequency (`CCLK`) and `SCLK` frequency is programmable using a phase locked loop (PLL) system memory-mapped

register (MMR). For more information, see “Core Clock/System Clock Ratio Control” on page 8-5.

The external memory space is shown in Figure 5-1.

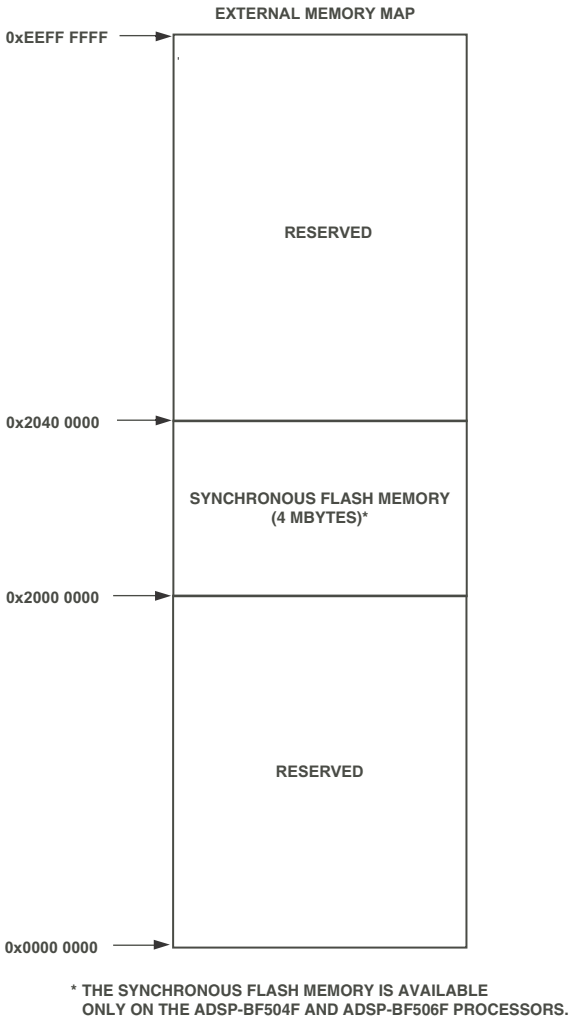


Figure 5-1. ADSP-BF50x External Memory Map

Note that, as shown in Figure 5-1, the region in external memory space from address 0x0000 0000 up to address 0x2000 0000 is reserved.

On ADSP-BF50x Blackfin processors that feature internal parallel flash memories (ADSP-BF504F and ADSP-BF506F), the region from 0x2000 0000 to 0x2040 0000 is dedicated to supporting the 4M Bytes internal parallel synchronous flash memory.

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

Block Diagram

Figure 5-2 is a conceptual block diagram of the EBIU and its interfaces.

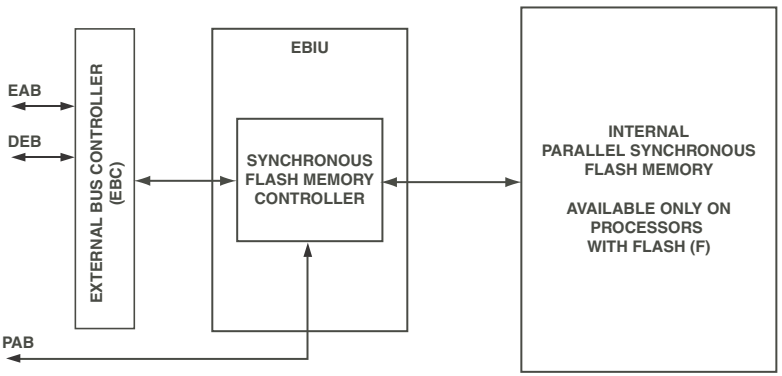


Figure 5-2. External Bus Interface Unit (EBIU)

The external bus interface unit (EBIU) interfaces to the processor busses on one side and to the internal flash memory on the other side.

Note that—because the EBIU memory-interface signals do not come out to package pins—no external memory devices can be supported by the EBIU in ADSP-BF50x Blackfin processors.

Internal Memory Interfaces

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

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

These are synchronous interfaces, clocked by `SCLK`, as is the EBIU. The EAB provides access to external memory.

The peripheral access bus (PAB) is used only to access the memory-mapped control and status registers of the EBIU. It does not need to arbitrate with, nor take access cycles from, the EAB bus.

The External Bus Controller (EBC) logic must arbitrate access requests for external memory coming from the EAB and DEB buses. Transactions from the core have priority over DMA accesses in most circumstances. However, if the DMA controller detects an excessive backup of transactions, it can request its priority to be temporarily raised above the core.

Registers

The EBIU has a number of control and status registers. They include:

- Asynchronous memory global control register (`EBIU_AMGCTL`)
- Asynchronous memory bank control register (`EBIU_AMBCTL`)

- Mode control register (EBIU_MODE)
- Parameter control register (EBIU_FCTL)

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

Error Detection

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

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

AMC Overview and Features

The following sections describe the features of the AMC. On ADSP-BF50xF Blackfin processors that include an internal flash memory, the asynchronous memory controller (AMC) provides a glue-less interface to the internal flash memory device.

Features

The EBIU AMC features include:

- 16-bit I/O width
- 3.3 V I/O supply
- Supports instruction fetch
- Allows booting

Asynchronous Memory Interface

The asynchronous memory interface allows a glue-less interface to internal flash memory.

Asynchronous Memory Address Decode

The address range allocated per bank is fixed at 4M bytes.

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

AMC Description of Operation

The following sections describe the operation of the AMC.

Avoiding Bus Contention

Be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure.

One case where contention can occur is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read.

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

AMC Programming Model

The asynchronous memory global control register (EBIU_AMGCTL) configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be programmed while the AMC is in use. The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

The AMC interface is used to access the internal flash memory on ADSP-BF50x processors containing a flash. For more information, see "Internal Flash Memory" on page 6-1.

Additional information for the EBIU_AMGCTL register bits includes:

- Asynchronous memory clock enable (AMCKEN)

  The external clock signal (CLKOUT), which is an inverted version of the system clock signal SCLK, can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

- Asynchronous memory bank enable (AMBEN).

  If a bus operation accesses a disabled asynchronous memory bank, the EBIU responds by acknowledging the transfer and asserting the error signal on the requesting bus. The error signal propagates back to the requesting bus master. This generates a hardware exception to the core, if it is the requester. For DMA mastered requests, the error is captured in the respective status register. If a bank is not fully populated with memory, then the memory likely aliases into multiple address regions within the bank. This aliasing condition is not detected by the EBIU, and no error response is asserted.

- Core/DMA priority (CDPRIO).

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

The `CDPRIO` bit also applies to the SDC.

The EBIU asynchronous memory controller has an asynchronous memory bank control registers (`EBIU_AMBCTL`). This register contains bits for counters for setup, access, and hold time; bits to determine memory type and size; and bits to configure use of `ARDY`. This register should not be programmed while the AMC is in use.

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

- Setup: the time between the beginning of a memory cycle (AMS low) and the read-enable assertion (ARE low) or write-enable assertion (AWE low).
- Read access: the time between read-enable assertion (ARE low) and deassertion (ARE high).
- Write access: the time between write-enable assertion (AWE low) and deassertion (AWE high).
- Hold: the time between read-enable deassertion (ARE high) or write-enable deassertion (AWE high) and the end of the memory cycle (AMS high).

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

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

EBIU Registers

The following sections describe the EBIU registers.

EBIU_AMGCTL Register

Figure 5-3 shows the asynchronous memory global control register (EBIU_AMGCTL).

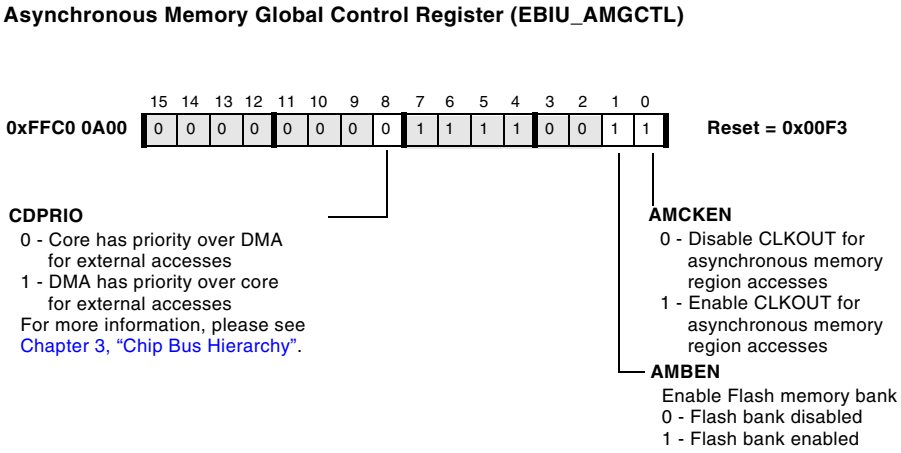


Figure 5-3. Asynchronous Memory Global Control Register

EBIU_AMBCTL Register

Figure 5-4 shows the asynchronous memory bank control register (EBIU_AMBCTL).

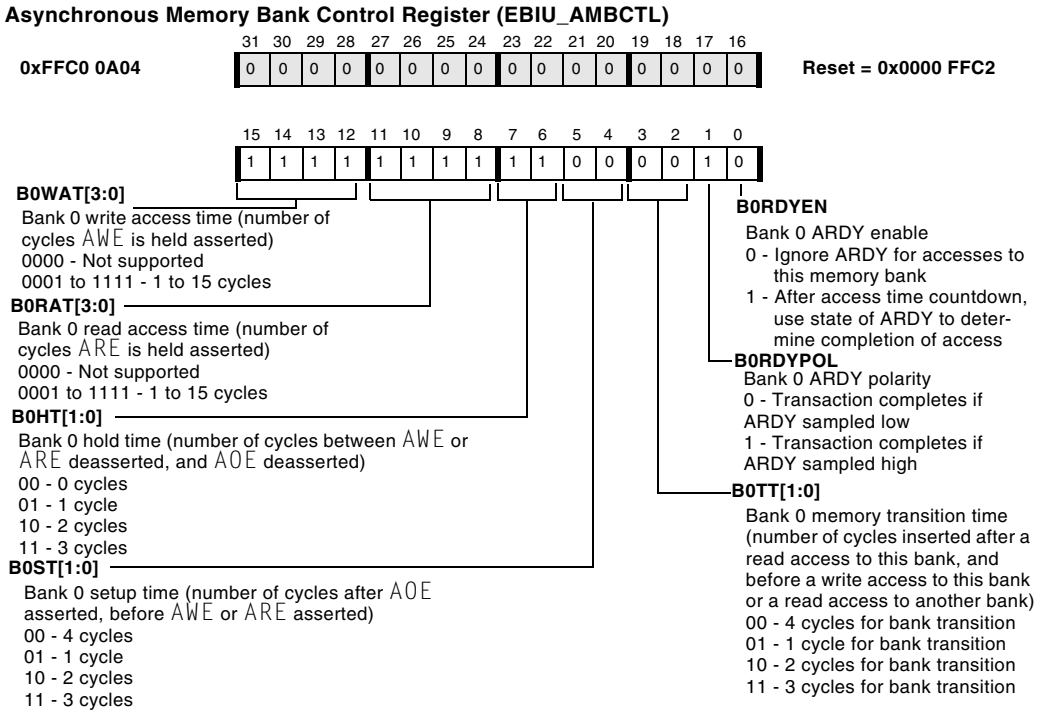


Figure 5-4. Asynchronous Memory Bank Control Register

EBIU_MODECTL Register

Figure 5-5 shows the asynchronous memory mode control register (EBIU_MODECTL).

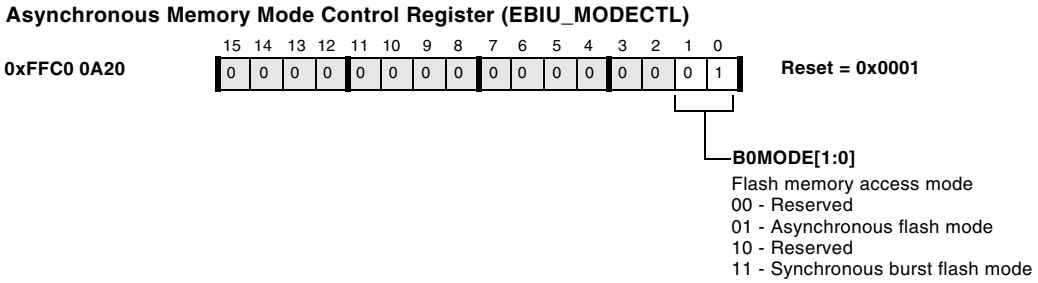


Figure 5-5. Asynchronous Memory Mode Control Register

EBIU_FCTL Register

Figure 5-6 shows the asynchronous internal flash memory parameter control register (EBIU_FCTL).

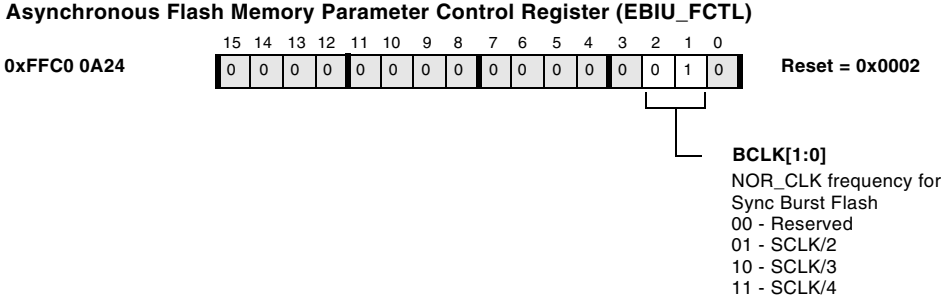


Figure 5-6. Asynchronous Internal Flash Memory Parameter Control Register

6 INTERNAL FLASH MEMORY

ADSP-BF50xF Blackfin processors interface to a 32M bit (2M x 16) device.

The internal flash memory has an array of 71 blocks, and is divided into 4M bit banks. There are 7 banks each containing 8 main blocks of 32K words, and one parameter bank containing 8 parameter blocks of 4K words and 7 main blocks of 32K words.

The AMC interface is used to access the internal flash memory on ADSP-BF50x processors containing a flash. For more information, see "External Bus Interface Unit" on page 5-1.

Overview

The internal flash memory related pins are shown in Table 6-1. Use this information when referring to the read and write waveforms shown in Figure 6-11 on page 6-82 and Figure 6-12 on page 6-87.

Table 6-1. EBIU/Internal Flash Memory Internal Pin Connections

EBIU Pins	Stacked Flash Pins	Comment
A21-A1	A20-A0	Address pins
D15-D0	D15-D0	Data
$\overline{\text{AMS0}}$	$\overline{\text{E}}$	Chip enable
$\overline{\text{ARE}}$	$\overline{\text{G}}$	Output enable
$\overline{\text{ADV}}$	$\overline{\text{L}}$	Latch enable (address valid)
NOR_CLK	K	Burst clock
$\overline{\text{ARDY}}$	WAIT	Wait
$\overline{\text{AWE}}$	$\overline{\text{W}}$	Write enable
–	$\overline{\text{WP}}$	Write protect (tied low)
$\overline{\text{RP}}$[1]	$\overline{\text{RP}}$	Reset
V_{PP}[1]	V_{PP}	Global program/erase protect

1 These are controlled through the internal flash memory control register. See "Internal Flash Memory Control (FLASH_CONTROL) Register" on page 6-88.

The ADSP-BF50xF processors contain an internal flash memory. It is a 32M bit (2M bit × 16) non-volatile flash memory. The internal flash memory can be erased electrically at block level and programmed in-system on a word-by-word basis using a 1.7 V to 2 V $V_{DDFLASH}$ supply for the circuitry and a 2.7 V to 3.3 V $V_{DDFLASH}$ supply for the input/output pins. Figure 6-1 and Table 6-2 show the internal connections to the flash memory.

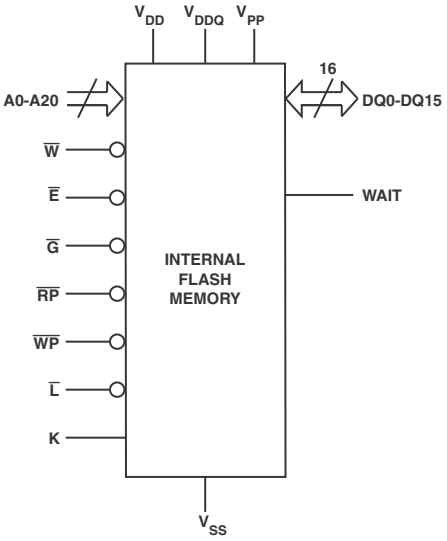


Figure 6-1. Internal Flash Memory Connections

Table 6-2. Internal Flash Memory Signal Names

Signal Name	Function	Direction
A0-A20	Address inputs	Inputs
D0-D15	Data input/outputs, command inputs	I/O
$\overline{E}$	Chip Enable	Input
$\overline{G}$	Output Enable	Input
$\overline{W}$	Write Enable	Input
$\overline{RP}$	Reset	Input
$\overline{WP}$	Write Protect	Input
K	Clock	Input
$\overline{L}$	Latch Enable	Input
WAIT	Wait	Output
V_{DD}	Supply voltage	Input

Table 6-2. Internal Flash Memory Signal Names (Cont'd)

Signal Name	Function	Direction
$V_{DDFLASH}$	Supply voltage for input/output buffers	Input
V_{PP}	Global program/erase protect	Input
V_{SS}	Ground	

The internal flash memory features an asymmetrical block architecture.

The internal flash memory has an array of 71 blocks, and is divided into 4M bit banks. There are 7 banks each containing 8 main blocks of 32K words, and one parameter bank containing 8 parameter blocks of 4K words and 7 main blocks of 32K words.

The multiple bank architecture allows dual operations. While programming or erasing in one bank, read operations are possible in other banks. Only one bank at a time is allowed to be in program or erase mode. It is possible to perform burst reads that cross bank boundaries. The bank architectures are summarized in Table 6-3.

Table 6-3. Internal Flash Memory Bank Architecture

Number	Bank Size	Parameter Blocks	Main Blocks
Parameter bank	4M bit	8 blocks of 4K word	7 blocks of 32K word
Bank 1	4M bit	-	8 blocks of 32K word
Bank 2	4M bit	-	8 blocks of 32K word
Bank 3	4M bit	-	8 blocks of 32K word
Bank 4	4M bit	-	8 blocks of 32K word
Bank 5	4M bit	-	8 blocks of 32K word
Bank 6	4M bit	-	8 blocks of 32K word
Bank 7	4M bit	-	8 blocks of 32K word

Each block can be erased separately. Erase can be suspended to perform program in any other block, and then resumed. Program can be suspended to read data in any other block and then resumed. Each block can be programmed and erased over 100,000 cycles.

Program and erase commands are written to the command interface of the memory. An internal program/erase controller manages the timings necessary for program and erase operations. The end of a program or erase operation can be detected and any error conditions identified in the status register. The command set required to control the memory is consistent with JEDEC standards.

The device supports synchronous burst read and asynchronous read from all blocks of the memory array; at power-up the device is configured for asynchronous read. In synchronous burst mode, data is output on each clock cycle at frequencies of up to 50 MHz. The synchronous burst read operation can be suspended and resumed.

The device features an automatic standby mode. When the bus is inactive during asynchronous read operations, the device automatically switches to the automatic standby mode. In this condition the power consumption is reduced to the standby value I_{DD4} and the outputs are still driven.

The internal flash memory features an instant, individual block locking scheme that allows any block to be locked or unlocked with no latency, enabling instant code and data protection. All blocks have three levels of protection. They can be locked and locked-down individually preventing any accidental programming or erasure. There is additional hardware protection against program and erase. When $V_{PP} \leq V_{PPLK}$, all blocks are protected against program or erase. All blocks are locked at power-up.

The device includes a protection register to increase the protection of a system's design. The protection register is divided into two segments: a 64-bit segment containing a unique device number and a 128-bit segment one-time-programmable (OTP) by the user. The user programmable

segment can be permanently protected. Figure 6-2 on page 6-18 shows the protection register memory map.

Command Interface to Internal Flash Memory

All bus write operations to the internal flash memory device are interpreted by the command interface. Commands consist of one or more sequential bus write operations. An internal program/erase controller manages all timings and verifies the correct execution of the program and erase commands. The program/erase controller provides a status register whose output may be read at any time to monitor the progress or the result of the operation.

The command interface is reset to read mode when power is first applied or when exiting from reset. Command sequences must be followed exactly. Any invalid combination of commands is ignored.

Refer to Table 6-4 and "Command Interface State Tables" on page 6-68 for a summary of the command interface.

Table 6-4. Command Codes

Hex Code	Command
0x01	Block Lock Confirm
0x03	Set Configuration Register Confirm
0x10	Alternative Program Setup
0x20	Block Erase Setup
0x2F	Block Lock-Down Confirm
0x40	Program Setup
0x50	Clear Status Register

Table 6-4. Command Codes (Cont'd)

Hex Code	Command
0x60	Block Lock Setup, Block Unlock Setup, Block Lock Down Setup and Set Configuration Register Setup
0x70	Read Status Register
0x90	Read Electronic Signature
0x98	Read CFI Query
0xB0	Program/Erase Suspend
0xC0	Protection Register Program
0xD0	Program/Erase Resume, Block Erase Confirm or Block Unlock Confirm
0xFF	Read Array

Command Interface – Standard Commands

The following commands are the basic commands used to read, write to, and configure the device. Refer to Table 6-5 on page 6-16 in conjunction with the following descriptions in this section.

Read Array Command

The read array command returns the addressed bank to read array mode. One bus write cycle is required to issue the read array command and return the addressed bank to read array mode. Subsequent read operations read the addressed location and output the data. A read array command can be issued in one bank while programming or erasing in another bank. However, if a read array command is issued to a bank currently executing a program or erase operation the command is executed but the output data is not guaranteed.

Read Status Register Command

The status register indicates when a program or erase operation is complete and the success or failure of operation itself. Issue a read status

register command to read the status register content. The read status register command can be issued at any time, even during program or erase operations.

The following read operations output the content of the status register of the addressed bank. The status register is latched on the falling edge of $\overline{E}$ or $\overline{G}$ signals, and can be read until $\overline{E}$ or $\overline{G}$ returns to logic high. Either $\overline{E}$ or $\overline{G}$ must be toggled to update the latched data. See Table 6-7 on page 6-23 for the description of the status register bits. This mode supports asynchronous or single synchronous reads only.

Read Electronic Signature Command

The read electronic signature command reads the manufacturer and device codes, the block locking status, the protection register, and the configuration register.

The read electronic signature command consists of one write cycle to an address within one of the banks. A subsequent read operation in the same bank outputs the manufacturer code, the device code, the protection status of the blocks in the targeted bank, the protection register, or the configuration register (see Table 6-9 on page 6-28).

Dual operations between the parameter bank and the electronic signature locations are not allowed (see Table 6-14 on page 6-38).

If a read electronic signature command is issued in a bank that is executing a program or erase operation, the bank goes into read electronic signature mode, subsequent bus read cycles output the electronic signature data, and the program/erase controller continues to program or erase in the background. This mode supports asynchronous or single synchronous reads only; it does not support synchronous burst reads.

Read CFI Query Command

The read CFI query command reads data from the common flash interface (CFI). The read CFI query command consists of one bus write cycle to an address within one of the banks. Once the command is issued subsequent bus read operations in the same bank read from the common flash interface.

If a read CFI query command is issued in a bank that is executing a program or erase operation, the bank goes into read CFI query mode, subsequent bus read cycles output the CFI data, and the program/erase controller continues to program or erase in the background. This mode supports asynchronous or single synchronous reads only; it does not support page mode or synchronous burst reads.

The status of the other banks is not affected by the command (see Table 6-12 on page 6-37). After issuing a read CFI query command, a read array command should be issued to the addressed bank to return the bank to read array mode.

Dual operations between the parameter bank and the CFI internal flash memory space are not allowed (see Table 6-14 on page 6-38 for details).

See "Common Flash Interface" on page 6-45 for details on the information contained in the common flash interface memory area.

Clear Status Register Command

The clear status register command resets (set to '0') error bits `SR1`, `SR3`, `SR4` and `SR5` in the status register. One bus write cycle is required to issue the clear status register command. The clear status register command does not change the read mode of the bank.

The error bits in the status register do not automatically return to '0' when a new command is issued. The error bits in the status register should be cleared before attempting a new program or erase command.

Block Erase Command

The block erase command erases a block. It sets all the bits within the selected block to '1'. All previous data in the block is lost. If the block is protected then the erase operation aborts, the data in the block does not change, and the status register outputs the error. The block erase command can be issued at any moment, regardless of whether the block has been programmed or not.

Two bus write cycles are required to issue the command:

- The first bus cycle sets up the erase command
- The second latches the block address in the program/erase controller and starts it

If the second bus cycle is not write erase confirm (0xD0), status register bits SR4 and SR5 are set and the command aborts. Erase aborts if reset is asserted ($\overline{RP}$ driven low). As data integrity cannot be guaranteed when the erase operation is aborted, the block must be erased again.

Once the command is issued, the device outputs the status register data when any address within the bank is read. At the end of the operation the bank remains in read status register mode until a read array, read CFI query, or read electronic signature command is issued.

During erase operations the bank containing the block being erased only accepts the read array, read status register, read electronic signature, read CFI query and the program/erase suspend commands; all other commands are ignored. Refer to "Dual Operations and Multiple Bank Architecture" on page 6-36 for detailed information about simultaneous operations allowed in banks not being erased. Typical erase times are given in the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

See Figure 6-7 on page 6-60 and Listing 6-3 on page 6-61 for a suggested flowchart and pseudo code for using the block erase command.

Program Command

The internal flash memory device array can be programmed word-by-word. Only one word in one bank can be programmed at any one time. If the block is protected, the program operation aborts, the data in the block does not change, and the status register outputs the error.

Two bus write cycles are required to issue the program command:

- The first bus cycle sets up the program command
- The second latches the address and the data to be written and starts the program/erase controller

After programming has started, read operations in the bank being programmed output the status register content.

During program operations the bank being programmed only accepts the read array, read status register, read electronic signature, read CFI query and the program/erase suspend commands. Refer to "Dual Operations and Multiple Bank Architecture" on page 6-36 for detailed information about simultaneous operations allowed in banks not being programmed. Typical program times are given in the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

Programming aborts if reset is asserted ($\overline{\text{RP}}$ driven low). As data integrity cannot be guaranteed when the program operation is aborted, the internal flash memory device location must be reprogrammed.

See Figure 6-5 on page 6-56 and Listing 6-1 on page 6-57 for a flowchart and pseudo code for using the program command.

Program/Erase Suspend Command

The program/erase suspend command pauses a program or block erase operation.

One bus write cycle is required to issue the program/erase suspend command. Once the program/erase controller has paused bits SR7, SR6 and/or SR2 of the status register are set to '1'. The command can be addressed to any bank.

During program/erase suspend the command interface accepts the program/erase resume, read array (cannot read the erase-suspended block or the program-suspended word), read status register, read electronic signature, clear status register, and read CFI query commands. In addition, if the suspended operation is erase then the set configuration register, program, block lock, block lock-down or block unlock commands are also accepted. The block being erased may be protected by issuing the block lock, or block lock-down commands. Only the blocks not being erased may be read or programmed correctly. When the program/erase resume command is issued the operation completes. Refer to "Dual Operations and Multiple Bank Architecture" on page 6-36 for detailed information about simultaneous operations allowed during program/erase suspend.

During a program/erase suspend, the device is placed in standby mode by taking chip enable ($\overline{E}$) to logic high. Program/erase is aborted if reset is asserted ($\overline{RP}$ driven low).

See Figure 6-6 on page 6-58, Listing 6-2 on page 6-59, Figure 6-8 on page 6-62, and Listing 6-4 on page 6-63 for flowcharts that illustrate usage of the program/erase suspend command.

Program/Erase Resume Command

The program/erase resume command restarts the program/erase controller after a program/erase suspend command has paused it. One bus write cycle is required to issue the command. The command can be written to any address.

The program/erase resume command does not change the read mode of the banks. If the suspended bank is in read status register, read electronic signature or read CFI query mode the bank remains in that mode and

outputs the corresponding data. If the bank is in read array mode, subsequent read operations output invalid data.

If a program command is issued during a block erase suspend, the erase cannot be resumed until the programming operation has completed. It is possible to accumulate suspend operations. For example, it is possible to suspend an erase operation, start a programming operation, suspend the programming operation, and then read the array. See Figure 6-6 on page 6-58, Listing 6-2 on page 6-59, Figure 6-8 on page 6-62, and Listing 6-4 on page 6-63 for flowcharts that illustrate usage of the program/erase suspend command.

Protection Register Program Command

The protection register program command programs the 128-bit user OTP segment of the protection register and the protection register lock. The segment is programmed 16 bits at a time. When shipped, all bits in the segment are set to '1'. The user can only program the bits to '0'.

Two write cycles are required to issue the protection register program command:

- The first bus cycle sets up the protection register program command.
- The second latches the address and the data to be written to the protection register and starts the program/erase controller.

Read operations output the status register content after the programming has started.

The segment can be protected by programming bit 1 of the protection lock register (see Figure 6-2 on page 6-18). Attempting to program a previously protected protection register results in a status register error. The protection of the protection register is not reversible. The protection register program cannot be suspended. Dual operations between the parameter

bank and the protection register internal flash memory space are not allowed (see Table 6-14 on page 6-38).

The Set Configuration Register Command

The set configuration register command writes a new value to the configuration register, which defines the burst length, type, X latency, synchronous/asynchronous read mode, and the valid clock edge configuration.

Two bus write cycles are required to issue the set configuration register command:

- The first cycle writes the setup command and the address corresponding to the configuration register content.
- The second cycle writes the configuration register data and the confirm command.

Read operations output the internal flash memory device array content after the set configuration register command is issued.

The value for the configuration register is always presented on `A0-A15`. `CR0` is on `A0`, `CR1` on `A1`, and so on; the other address bits are ignored.

Block Lock Command

The block lock command locks a block and prevents program or erase operations from changing the data in it. All blocks are locked at power-up or reset.

Two bus write cycles are required to issue the block lock command:

- The first bus cycle sets up the block lock command.
- The second bus write cycle latches the block address.

The lock status can be monitored for each block using the read electronic signature command. Table 16 shows the lock status after issuing a block lock command.

The block lock bits are volatile; once set they remain set until a hardware reset or power-down/power-up. They are cleared by a block unlock command. Refer to "Block Locking" on page 6-38 for a detailed explanation. See Figure 6-9 on page 6-64 for a flowchart for using the lock command.

Block Unlock Command

The block unlock command unlocks a block, allowing the block to be programmed or erased. Two bus write cycles are required to issue the block unlock command:

- The first bus cycle sets up the block unlock command.
- The second bus write cycle latches the block address.

The lock status can be monitored for each block using the read electronic signature command. Table 16 shows the protection status after issuing a block unlock command. Refer to "Block Locking" on page 6-38 for a detailed explanation, and Figure 6-9 on page 6-64 and Listing 6-5 on page 6-64 for a flowchart and pseudo code for using the unlock command.

Block Lock-Down Command

A locked or unlocked block can be locked down by issuing the block lock-down command. A locked-down block cannot be programmed or erased, or have its protection status changed when $\overline{WP}$ is low, which is always the case in ADSP-BF50xF processors. Refer to Table 6-1 on page 6-2 (EBIU/Internal Flash Memory Internal Pin Connections).

Two bus write cycles are required to issue the block lock-down command:

- The first bus cycle sets up the block lock command.
- The second bus write cycle latches the block address.

The lock status can be monitored for each block using the read electronic signature command. Locked-down blocks revert to the locked (and not locked-down) state when the device is reset on power-down. Table 6-15 on page 6-41 shows the lock status after issuing a block lock-down command. Refer to "Block Locking" on page 6-38 for a detailed explanation, and Figure 6-9 on page 6-64 and Listing 6-5 on page 6-64 for a flowchart and pseudo code for using the lock-down command.

Table 6-5. Standard Commands

Commands	Cycles	Bus Operations[1]					
		1st Cycle			2nd Cycle		
		Op.	Add	Data	Op.	Add	Data
Read Array	1+	Write	BKA	0xFF	Read	WA	RD
Read Status Register	1+	Write	BKA	0x70	Read	BKA[2]	SRD
Read Electronic Signature	1+	Write	BKA	0x90	Read	BKA[2]	ESD
Read CFI Query	1+	Write	BKA	0x98	Read	BKA[2]	QD
Clear Status Register	1	Write	X	0x50			
Block Erase	2	Write	BKA or BA[3]	0x20	Write	BA	0xD0
Program	2	Write	BKA or WA[3]	0x40or 0x10	Write	WA	PD
Program/Erase Suspend	1	Write	X	0xB0			
Program/Erase Resume	1	Write	X	0xD0			
Protection Register Program	2	Write	PRA	0xC0	Write	PRA	PRD
Set Configuration Register	2	Write	CRD	0x60	Write	CRD	0x03

Table 6-5. Standard Commands (Cont'd)

Commands	Cycles	Bus Operations[1]					
		1st Cycle			2nd Cycle		
		Op.	Add	Data	Op.	Add	Data
Block Lock	2	Write	BKA or BA[3]	0x60	Write	BA	0x01
Block Unlock	2	Write	BKA or BA[3]	0x60	Write	BA	0xD0
Block Lock-Down	2	Write	BKA or BA[3]	0x60	Write	BA	0x2F

1 X = 'don't care', WA = Word Address in targeted bank, RD = Read Data, SRD = Status Register Data, ESD = Electronic Signature Data, QD = Query Data, BA = Block Address, BKA = Bank Address, PD = Program Data, PRA = Protection Register Address, PRD = Protection Register Data, CRD = Configuration Register Data.
2 Must be same bank as in the first cycle. The signature addresses are listed in Table 6-6.
3 Any address within the bank can be used.

Table 6-6. Electronic Signature Codes

Code		Address (h)	Data (h)
Manufacturer Code		Bank address + 00	0020
Device Code	Top	Bank address + 01	8866
Block Protection	Locked	Block address + 02	0001
	Unlocked		0000
	Locked and locked-down		0003
	Unlocked and locked-down		0002
Reserved		Bank address + 03	Reserved
Configuration Register		Bank address + 05	CR[1]

Table 6-6. Electronic Signature Codes (Cont'd)

Code		Address (h)	Data (h)
Protection Register Lock	Factory default	Bank address + 80	0002
	OTP area permanently locked		0000
Protection Register		Bank address + 81 Bank address + 84	Unique device number
		Bank address + 85 Bank address + 8C	OTP Area

1 CR = Configuration Register

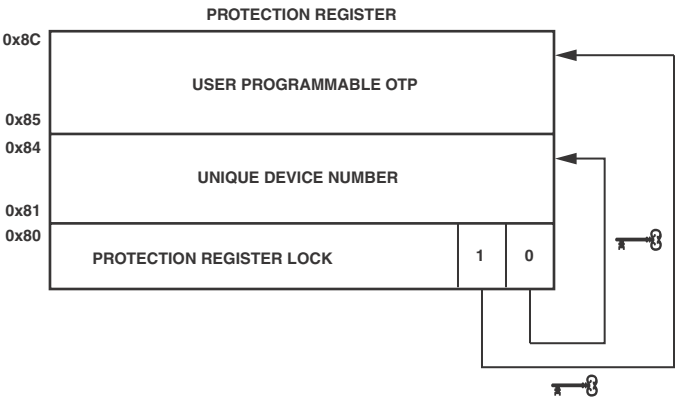


Figure 6-2. Protection Register Internal Flash Memory Map

Status Register

The status register provides information on the current or previous program or erase operations. Issue a read status register command to read the contents of the status register (refer to "Read Status Register Command" on page 6-7 for more details). To output the contents, the status register is latched and updated on the falling edge of the chip enable or output enable signals and can be read until chip enable or output enable are

deasserted. The status register can only be read using single asynchronous or single synchronous reads. Bus read operations from any address within the bank always read the status register during program and erase operations, as long as no read array command has been issued.

The various bits convey information about the status and any errors of the operation. Bits SR7, SR6, SR2 and SR0 provide information on the status of the device and are set and reset by the device. Bits SR5, SR4, SR3 and SR1 provide information on errors. They are set by the device but must be reset by issuing a clear status register command or a hardware reset. If an error bit is set to '1' the status register should be reset before issuing another command. SR7 to SR1 refer to the status of the device while SR0 refers to the status of the addressed bank.

The bits in the status register are summarized in Table 6-7 on page 6-23. Refer to Table 6-7 in conjunction with the descriptions in the following sections.

Program/Erase Controller Status Bit (SR7)

The program/erase controller status bit indicates whether the program/erase controller is active or inactive in any bank. When the program/erase controller status bit is low (set to '0'), the program/erase controller is active; when the bit is high (set to '1'), the program/erase controller is inactive, and the device is ready to process a new command.

The program/erase controller status is low immediately after a program/erase suspend command is issued until the program/erase controller pauses. After the program/erase controller pauses the bit is high.

During program and erase operations the program/erase controller status bit can be polled to find the end of the operation. Other bits in the status register should not be tested until the program/erase controller completes the operation and the bit is high.

After the program/erase controller completes its operation the erase status, program status, V_{PP} status and block lock status bits should be tested for errors.

Erase Suspend Status Bit (SR6)

The erase suspend status bit indicates that an erase operation has been suspended or is going to be suspended in the addressed block. When the erase suspend status bit is high (set to '1'), a program/erase suspend command has been issued and the internal flash memory is waiting for a program/erase resume command.

The erase suspend status should only be considered valid when the program/erase controller status bit is high (program/erase controller inactive). SR7 is set within the erase suspend latency time of the program/erase suspend command being issued, therefore, the internal flash memory may still complete the operation rather than entering the suspend mode.

When a program/erase resume command is issued the erase suspend status bit returns low.

Erase Status Bit (SR5)

The erase status bit identifies if the internal flash memory has failed to verify that the block has erased correctly. When the erase status bit is high (set to '1'), the program/erase controller has applied the maximum number of pulses to the block and still failed to verify that it has erased correctly. The erase status bit should be read once the program/erase controller status bit is high (program/erase controller inactive).

Once set high, the erase status bit can only be reset low by a clear status register command or a hardware reset. If set high it should be reset before a new program or erase command is issued, otherwise the new command appears to fail.

Program Status Bit (SR4)

The program status bit identifies a program failure.

When the program status bit is high (set to '1'), the program/erase controller has applied the maximum number of pulses to the byte and still failed to verify that it has programmed correctly.

The program status bit should be read once the program/erase controller status bit is high (program/erase controller inactive).

Once set high, the program status bit can only be reset low by a clear status register command or a hardware reset. If set high it should be reset before a new command is issued, otherwise the new command appears to fail.

V_{PP} Status Bit (SR3)

The V_{PP} status bit identifies an invalid voltage on the V_{PP} pin during program and erase operations. The V_{PP} pin is only sampled at the beginning of a program or erase operation. Indeterminate results can occur if V_{PP} becomes invalid during an operation.

When the V_{PP} status bit is low (set to '0'), the voltage on the V_{PP} pin was sampled at a valid voltage. When the V_{PP} status bit is high (set to '1'), the V_{PP} pin has a voltage that is below the V_{PP} lockout voltage, V_{PPLK}, the internal flash memory is protected and program and erase operations cannot be performed.

Once set high, the V_{PP} status bit can only be reset low by a clear status register command or a hardware reset. If set high it should be reset before a new program or erase command is issued, otherwise the new command appears to fail.

Program Suspend Status Bit (SR2)

The program suspend status bit indicates that a program operation has been suspended in the addressed block. When the program suspend status bit is high (set to '1'), a program/erase suspend command has been issued and the internal flash memory is waiting for a program/erase resume command. The program suspend status should only be considered valid when the program/erase controller status bit is high (program/erase controller inactive). SR2 is set within the program suspend latency time of the program/erase suspend command being issued, therefore, the internal flash memory may still complete the operation rather than entering the suspend mode.

When a program/erase resume command is issued, the program suspend status bit returns low.

Block Protection Status Bit (SR1)

The block protection status bit can be used to identify if a program or block erase operation has tried to modify the contents of a locked or locked-down block.

When the block protection status bit is high (set to '1'), a program or erase operation has been attempted on a locked or locked-down block.

Once set high, the block protection status bit can only be reset low by a clear status register command or a hardware reset. If set high it should be reset before a new command is issued, otherwise the new command appears to fail.

Bank Write Status Bit (SR0)

The bank write status bit indicates whether the addressed bank is programming or erasing. The bank write status bit should only be considered valid when the program/erase controller status SR7 is low (set to '0').

When both the program/erase controller status bit and the bank write status bit are low (set to ‘0’), the addressed bank is executing a program or erase operation. When the program/erase controller status bit is low (set to ‘0’) and the bank write status bit is high (set to ‘1’), a program or erase operation is being executed in a bank other than the one being addressed.

Refer to “Flowcharts and Pseudo Codes” on page 6-56 for status register usage.

Table 6-7. Status Register Bits

Bit	Name	Type	Logic level[1]	Definition
SR7	P/EC status	Status	'1'	Ready
			'0'	Busy
SR6	Erase suspend status	Status	'1'	Erase suspended
			'0'	Erase in progress or completed
SR5	Erase status	Error	'1'	Erase error
			'0'	Erase success
SR4	Program status	Error	'1'	Program error
			'0'	Program success
SR3	V_{PP} status	Error	'1'	V_{PP} invalid, abort
			'0'	V_{PP} OK
SR2	Program suspend status	Status	'1'	Program suspended
			'0'	Program in progress or completed
SR1	Block protection status	Error	'1'	Program/erase on protected block, abort
			'0'	No operation to protected blocks

Table 6-7. Status Register Bits (Cont'd)

Bit	Name	Type	Logic level[1]	Definition	
SR0	Bank write status	Status	'1'	SR7 = '1'	Not allowed
				SR7 = '0'	Program or erase operation in a bank other than the addressed bank
			'0'	SR7 = '1'	No program or erase operation in the device
				SR7 = '0'	Program or erase operation in addressed bank

1 Logic level '1' is High, '0' is Low.

Configuration Register

The configuration register configures the type of bus access that the internal flash memory performs. Refer to "Read Modes" on page 6-33 for details on read operations.

The configuration register is set through the command interface. After a reset or power-up the device is configured for asynchronous page read (`CR15 = 1`). The configuration register bits are described in Table 6-9 on page 6-28. They specify the selection of the burst length, burst type, burst X latency, and the read operation.

Since the internal flash device in ADSP-BF50xF processors can only be connected to the external bus interface unit (EBIU), some combinations of the flash configurations are not supported. Limitation on supported combinations are described in the section "Supported Configuration Register Combinations in ADSP-BF50xF Processors" on page 6-84.

Read Select Bit (CR15)

The read select bit, `CR15`, switches between asynchronous and synchronous bus read operations. When the read select bit is set to '1', read operations

are asynchronous; when the read select bit is set to ‘0’, read operations are synchronous. Synchronous burst read is supported in both parameter and main blocks and can be performed across banks.

On reset or power-up the read select bit is set to ‘1’ for asynchronous access.

X Latency Bits (CR13-CR11)

The X latency bits are used during synchronous read operations to set the number of clock cycles between the address being latched and the first data becoming available. For correct operation the X latency bits can only assume the values in Table 6-9 on page 6-28.

Table 6-8 shows how to set the X latency parameter, taking into account the frequency used to read the internal flash memory in synchronous mode.

Table 6-8. Latency Settings

$f_K$max	$t_K$min	X latency min
30 MHz	33 ns	2
40 MHz	25 ns	3
50 MHz	19 ns	4

Wait Polarity Bit (CR10)

In synchronous burst mode, the WAIT signal indicates whether the output data are valid or a wait state must be inserted. The wait polarity bit is used to set the polarity of the WAIT signal. When the wait polarity bit is set to ‘0’ the WAIT signal is active low. When the wait polarity bit is set to ‘1’ the WAIT signal is active high.

Data Output Configuration Bit (CR9)

The data output configuration bit determines whether the output remains valid for one or two clock cycles. When the data output configuration bit is ‘0’ the output data is valid for one clock cycle. When the data output configuration bit is ‘1’ the output data is valid for two clock cycles.

The data output configuration depends on the condition:

$$t_K > t_{KQV} + t_{QVK\_CPU}$$

where t_K is the clock period, t_{QVK_CPU} is the data setup time required by the system that is accessing the flash (for example, the processor) and t_{KQV} is the clock to data valid time. If this condition is not satisfied, the data output configuration bit should be set to ‘1’ (two clock cycles). Refer to Figure 6-3.

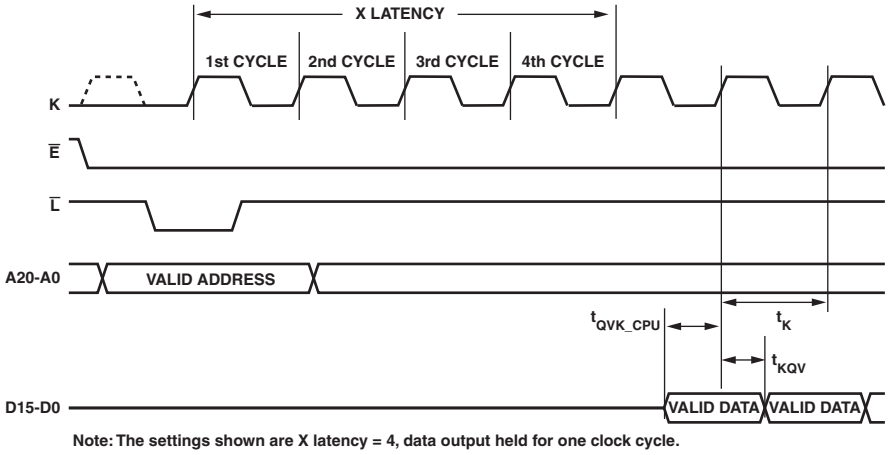


Figure 6-3. X Latency and Data Output Configuration Example

Wait Configuration Bit (CR8)

In burst mode, the wait bit controls the timing of the wait output pin, WAIT. When WAIT is asserted, data is not valid and when WAIT is deasserted, data is valid. When the wait bit is '0' the wait output pin is asserted during the wait state. When the wait bit is '1' the wait output pin is asserted one clock cycle before the wait state.

Burst Type Bit (CR7)

The burst type bit configures the sequence of addresses read as sequential or interleaved. When the burst type bit is '0' the internal flash memory outputs from interleaved addresses. When the burst type bit is '1' the internal flash memory outputs from sequential addresses. See Table 6-10 on page 6-30 and Table 6-11 on page 6-32 for the sequence of addresses output from a given starting address in each mode.

Valid Clock Edge Bit (CR6)

The valid clock edge bit, CR6, configures the active edge of the clock, K, during synchronous burst read operations. When the valid clock edge bit is '0' the falling edge of the clock is the active edge. When the valid clock edge bit is '1' the rising edge of the clock is active.

Wrap Burst Bit (CR3)

The burst reads can be confined inside the 4 or 8-word boundary (wrap) or overcome the boundary (no wrap). The wrap burst bit selects between wrap and no wrap. When the wrap burst bit is set to '0' the burst read wraps; when it is set to '1' the burst read does not wrap.

Burst Length Bits (CR2-CR0)

The burst length bits set the number of words to be output during a synchronous burst read operation as result of a single address latch cycle.

They can be set for 4 words, 8 words, 16 words or continuous burst, where all the words are read sequentially.

In continuous burst mode the burst sequence can cross bank boundaries.

In continuous burst mode or in 4, 8, 16 words no-wrap, depending on the starting address, the device asserts the WAIT output to indicate that a delay is necessary before the data is output.

If the starting address is aligned to a 4 word boundary no wait states are needed and the WAIT output is not asserted.

If the starting address is shifted by 1, 2 or 3 positions from the 4-word boundary, WAIT is asserted for 1, 2 or 3 clock cycles when the burst sequence crosses the first 16 word boundary to indicate that the device needs an internal delay to read the successive words in the array. WAIT is asserted only once during a continuous burst access. See also Table 6-10 on page 6-30 and Table 6-11 on page 6-32.

CR14, CR5 and CR4 are reserved for future use.

Table 6-9. Configuration Register Bits

Bit	Description	Value	Description
CR15	Read Select	0	Synchronous read
		1	Asynchronous read (default at power-on)
CR14	Reserved		
CR13-CR11	X Latency	010	2 clock latency
		011	3 clock latency
		100	4 clock latency
		101	5 clock latency
		111	Reserved (default)
		Other configurations reserved	

Table 6-9. Configuration Register Bits (Cont'd)

Bit	Description	Value	Description
CR10	Wait Polarity	0	WAIT is active low
		1	WAIT is active high (default)
CR9	Data Output Configuration	0	Data held for one clock cycle
		1	Data held for two clock cycles (default)
CR8	Wait Configuration	0	WAIT is active during wait state
		1	WAIT is active one data cycle before wait state (default)
CR7	Burst Type	0	Interleaved
		1	Sequential (default)
CR6	Valid Clock Edge	0	Falling clock edge
		1	Rising clock edge (default)
CR5-CR4	Reserved		
CR3	Wrap Burst	0	Wrap
		1	No wrap (default)
CR2-CR0	Burst Length	001	4 words
		010	8 words
		011	16 words
		111	Continuous (CR7 must be set to '1') (default)

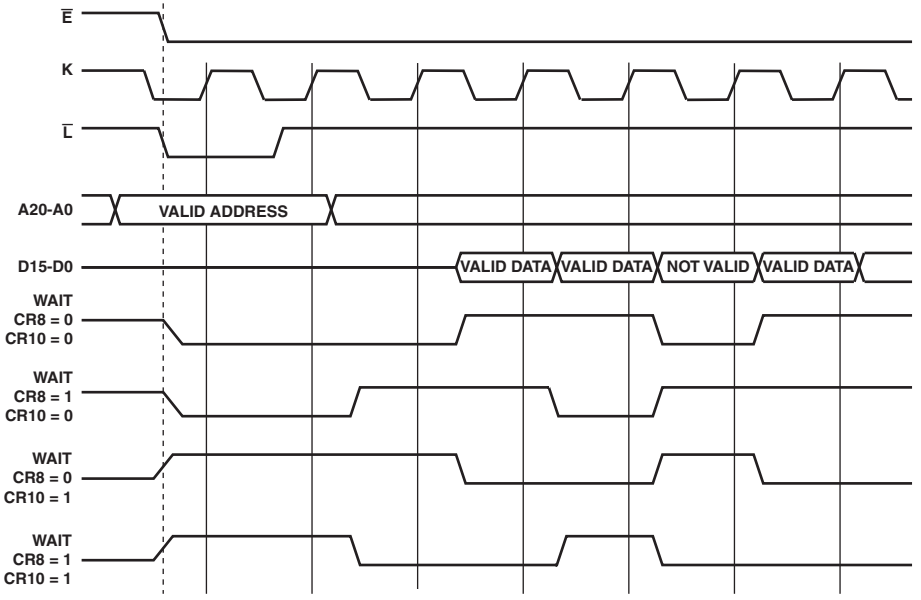


Figure 6-4. Wait Configuration Example

Table 6-10. Burst Type Definition (Wrap Mode)

Start Add	4 Words		8 Words		16 Words		Continuous Burst
	Sequential	Interleaved	Sequential	Interleaved	Sequential	Interleaved	
0	0-1-2-3	0-1-2-3	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15	0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15	0-1-2-3-4-5-6...
1	1-2-3-0	1-0-3-2	1-2-3-4-5-6-7-0	1-0-3-2-5-4-7-6	1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-0	1-0-3-2-5-4-7-6-9-8-11-10-13-12-15-14	1-2-3-4-5-6-7-...15-WAIT-16-17-18...
2	2-3-0-1	2-3-0-1	2-3-4-5-6-7-0-1	2-3-0-1-6-7-4-5	2-3-4-5-6-7-8-9-10-11-12-13-14-15-0-1	2-3-0-1-6-7-4-5-10-11-8-9-14-15-12-13	2-3-4-5-6-7...15-WAIT-WAIT-16-17-18...

Table 6-10. Burst Type Definition (Wrap Mode) (Cont'd)

Start Add	4 Words		8 Words		16 Words		Continuous Burst
	Sequential	Interleaved	Sequential	Interleaved	Sequential	Interleaved	
3	3-0-1-2	3-2-1-0	3-4-5-6-7-0-1-2	3-2-1-0-7-6-5-4	3-4-5-6-7-8-9-10-11-12-13-14-15-0-1-2	3-2-1-0-7-6-5-4-11-10-9-8-15-14-13-12	3-4-5-6-7...15-WAIT-WAIT-WAIT-16-17-18...
...							
7	7-4-5-6	7-6-5-4	7-0-1-2-3-4-5-6	7-6-5-4-3-2-1-0	7-8-9-10-11-12-13-14-15-0-1-2-3-4-5-6	7-6-5-4-3-2-1-0-15-14-13-12-11-10-9-8	7-8-9-10-11-12-13-14-15-WAIT-WAIT-WAIT-16-17...
...							
12							12-13-14-15-16-17-18...
13							13-14-15-WAIT-16-17-18...
14							14-15-WAIT-WAIT-16-17-18....
15							15-WAIT-WAIT-WAIT-16-17-18...

Table 6-11. Burst Type Definition (No-Wrap Mode)

Start Add	4 Words	8 Words	16 Words	Continuous Burst
	Sequential	Sequential	Sequential	
0	0-1-2-3	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15	Same as for Wrap (Wrap/No Wrap has no effect on Continuous Burst)
1	1-2-3-4	1-2-3-4-5-6-7-8	1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-WAIT-16	
2	2-3-4-5	2-3-4-5-6-7-8-9...	2-3-4-5-6-7-8-9-10-11-12-13-14-15-WAIT-WAIT-16-17	
3	3-4-5-6	3-4-5-6-7-8-9-10	3-4-5-6-7-8-9-10-11-12-13-14-15-WAITWAIT-WAIT-16-17-18	
...				
7	7-8-9-10	7-8-9-10-11-12-13-14	7-8-9-10-11-12-13-14-15-WAIT-WAIT-WAIT-16-17-18-19-20-21-22	
...				
12	12-13-14-15	12-13-14-15-16-17-18-19	12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27	
13	13-14-15-WAIT-16	13-14-15-WAIT-16-17-18-19-20	13-14-15-WAIT-16-17-18-19-20-21-22-23-24-25-26-27-28	
14	14-15-WAIT-WAIT-16-17	14-15-WAITWAIT-16-17-18-19-20-21	14-15-WAIT-WAIT-16-17-18-19-20-21-22-23-24-25-26-27-28-29	
15	15-WAIT-WAIT-WAIT-16-17-18	15-WAIT-WAIT-WAIT-16-17-18-19-20-21-22	15-WAIT-WAIT-WAIT-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30	

Read Modes

Read operations can be performed in two different ways depending on the settings in the configuration register. If the clock signal is ‘don’t care’ for the data output, the read operation is asynchronous. If the data output is synchronized with clock, the read operation is synchronous.

The read mode and data output format are determined by the configuration register (see “Configuration Register” on page 6-24 for details). All banks support both asynchronous and synchronous read operations. The multiple bank architecture allows read operations in one bank, while write operations are being executed in another (see Table 6-12 on page 6-37 and Table 6-13 on page 6-37).

Asynchronous Read Mode

In asynchronous read operations the clock signal is ‘don’t care’. The device outputs the data corresponding to the address latched, that is the internal flash memory array, status register, common flash interface or electronic signature, depending on the command issued. `CR15` in the configuration register must be set to ‘1’ for asynchronous operations.

During asynchronous read operations, after a bus inactivity of 150 ns, the device automatically switches to automatic standby mode. In this condition the power consumption is reduced to the standby value and the outputs are still driven.

In asynchronous read mode, the `WAIT` signal is always asserted.

Synchronous Burst Read Mode

In synchronous burst read mode the data is output in bursts synchronized with the clock. It is possible to perform burst reads across bank boundaries.

Synchronous burst read mode can only be used to read the internal flash memory array. For other read operations, such as read status register, read CFI, and read electronic signature, single synchronous read or asynchronous random access read must be used.

In synchronous burst read mode the flow of the data output depends on parameters that are configured in the configuration register.

A burst sequence is started at the first clock edge (rising or falling depending on valid clock edge bit CR6 in the configuration register) after the falling edge of latch enable or chip enable, whichever occurs last. Addresses are internally incremented and after a delay of 2 to 5 clock cycles (X latency bits CR13-CR11) the corresponding data is output on each clock cycle.

The number of words to be output during a synchronous burst read operation can be configured as 4, 8, 16 words, or continuous (burst length bits CR2-CR0). The data can be configured to remain valid for one or two clock cycles (data output configuration bit CR9).

The order of the data output can be modified through the burst type and the wrap burst bits in the configuration register. The burst sequence may be configured to be sequential or interleaved (CR7). The burst reads can be confined inside the 4, 8 or 16 word boundary (wrap) or overcome the boundary (no wrap). If the starting address is aligned to the burst length (4, 8 or 16 words) the wrapped configuration has no impact on the output sequence. Interleaved mode is not allowed in continuous burst read mode or with no wrap sequences.

A WAIT signal may be asserted to indicate to the system that an output delay occurs. This delay depends on the starting address of the burst sequence. The worst case delay occurs when the sequence is crossing a 16-word boundary and the starting address was at the end of a four word boundary.

WAIT is asserted during X latency, the wait state, and at the end of 4-, 8- or 16-word burst. It is only deasserted when output data are valid. In

continuous burst read mode a wait state occurs when crossing the first 16-word boundary. If the burst starting address is aligned to a 4-word page, the wait state does not occur.

The WAIT signal can be configured to be active low or active high by setting CR10 in the configuration register. The WAIT signal is meaningful only in synchronous burst read mode. In other modes, WAIT is always asserted (except for read array mode).

Synchronous Burst Read Suspend

A synchronous burst read operation can be suspended, freeing the data bus for other higher priority devices. It can be suspended during the initial access latency time (before data is output), or after the device has output data. When the synchronous burst read operation is suspended, internal array sensing continues and any previously latched internal data is retained. A burst sequence can be suspended and resumed as often as required as long as the operating conditions of the device are met.

A synchronous burst read operation is suspended when $\overline{E}$ is low and the current address has been latched (on a latch enable rising edge or on a valid clock edge). The clock signal is then halted at V_{IH} or at V_{IL}, and $\overline{G}$ goes high.

When $\overline{G}$ becomes low again and the clock signal restarts, the synchronous burst read operation is resumed exactly where it stopped.

WAIT being gated by $\overline{E}$ remains active and does not revert to high-impedance when $\overline{G}$ goes high. Therefore, if two or more devices are connected to the system's READY signal, to prevent bus contention the WAIT signal of the internal flash memory should not be directly connected to the system's READY signal.

Single Synchronous Read Mode

Single synchronous read operations are similar to synchronous burst read operations except that only the first data output after the X latency is valid. Synchronous single reads are used to read the electronic signature, status register, CFI, block protection status, configuration register status or protection register status. When the addressed bank is in read CFI, read status register or read electronic signature mode, the WAIT signal is always asserted.

Dual Operations and Multiple Bank Architecture

The multiple bank architecture of the internal flash device provides flexibility for software developers by allowing code and data to be split with 4M bit granularity. The dual operations feature simplifies the software management of the device and allows code to be executed from one bank while another bank is being programmed or erased.

The dual operations feature means that while programming or erasing in one bank, read operations are possible in another bank with zero latency (only one bank at a time is allowed to be in program or erase mode). If a read operation is required in a bank that is programming or erasing, the program or erase operation can be suspended. Also, if the suspended operation is erase then a program command can be issued to another block. This means the device can have one block in erase suspend mode, one programming, and other banks in read mode. Bus read operations are allowed in another bank between setup and confirm cycles of program or erase operations. The combination of these features means that read operations are possible at any moment.

Dual operations between the parameter bank and either the CFI, OTP, or the electronic signature internal flash memory space are not allowed. However, Table 6-14 on page 6-38 shows dual operations that are allowed between the CFI, OTP, electronic signature locations, and the internal flash memory array.

Table 6-12 and Table 6-13 show the dual operations possible in other banks and in the same bank. For a complete list of possible commands refer to "Command Interface State Tables" on page 6-68.

Table 6-12. Dual Operations Allowed in Other Banks

Status of Bank	Commands Allowed in Another Bank							
	Read Array	Read Status Register	Read CFI Query	Read Electronic Signature	Program	Block Erase	Program/ Erase Suspend	Program/ Erase Resume
Idle	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Programming	Yes	Yes	Yes	Yes	–	–	Yes	–
Erasing	Yes	Yes	Yes	Yes	–	–	Yes	–
Program suspended	Yes	Yes	Yes	Yes	–	–	–	Yes
Erase suspended	Yes	Yes	Yes	Yes	Yes	–	–	Yes

Table 6-13. Dual Operations Allowed in Same Bank

Status of Bank	Commands Allowed in Same Bank							
	Read Array	Read Status Register	Read CFI Query	Read Electronic Signature	Program	Block Erase	Program/ Erase Suspend	Program/ Erase Resume
Idle	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Programming	–[1]	Yes	Yes	Yes	–	–	Yes	–
Erasing	–[1]	Yes	Yes	Yes	–	–	Yes	–
Program suspended	Yes[2]	Yes	Yes	Yes	–	–	–	Yes
Erase suspended	Yes[2]	Yes	Yes	Yes	Yes[2]	–	–	Yes

1 The read array command is accepted but the data output is no guaranteed until the program or erase has completed.
2 Not allowed in the block or word that is being erased or programmed.

Table 6-14. Dual Operation Limitations

Current Status		Commands Allowed			
		Read CFI/OTP/ Electronic Signature	Read Parameter Blocks	Read Main Blocks	
				Located in Parameter Bank	Not located in Parameter Bank
Programming/erasing parameter blocks		No	No	No	Yes
Programming/ erasing main blocks	Located in parameter bank	Yes	No	No	Yes
	Not located in parameter bank	Yes	Yes	Yes	In different bank only
Programming OTP		No	No	No	No

Block Locking

The flash device features an instant, individual block locking scheme that enables any block to be locked or unlocked with no latency. This locking scheme has three levels of protection.

- Lock/unlock – this first level allows software-only control of block locking.
- Lock-down – this second level requires hardware interaction before locking can be changed.
- $V_{PP} \leq V_{PPLK}$ (for example, clearing the FLASH_UNPROTECT bit in the FLASH_CONTROL register) – the third level offers a complete hardware protection against program and erase on all blocks.

The protection status of each block can be set to locked, unlocked, and lock-down. Table 6-15 on page 6-41 defines all of the possible protection states ($\overline{WP}$, D1, D0), and Figure 6-9 on page 6-64 shows a flowchart for the locking operations.

Reading a Block's Lock Status

The lock status of every block can be read in the read electronic signature mode of the device. To enter this mode write 0x90 to the device. Subsequent reads at the address specified in Table 6-6 on page 6-17 output the protection status of that block. The lock status is represented by `D0` and `D1`. `D0` indicates the block lock/unlock status and is set by the lock command and cleared by the unlock command. It is also automatically set when entering lock-down. `D1` indicates the lock-down status and is set by the lock-down command. It cannot be cleared by software, only by a hardware reset or power-down.

The following sections explain the operation of the locking system.

Locked State

The default status of all blocks on power-up or after a hardware reset is locked (states (0,0,1) or (1,0,1)). Locked blocks are fully protected from any program or erase. Any program or erase operations attempted on a locked block returns an error in the status register. The status of a locked block can be changed to unlocked or lock-down using the appropriate software commands. An unlocked block can be locked by issuing the lock command.

Unlocked State

Unlocked blocks (states (0,0,0), (1,0,0) (1,1,0)), can be programmed or erased. All unlocked blocks return to the locked state after a hardware reset or when the device is powered-down. The status of an unlocked block can be changed to locked or locked-down using the appropriate software commands. A locked block can be unlocked by issuing the unlock command.

Lock-Down State

Blocks that are locked-down (state (0,1,x)) are protected from program and erase operations (as for locked blocks) but their protection status cannot be changed using software commands alone. A locked or unlocked block can be locked-down by issuing the lock-down command. Locked-down blocks revert to the locked state when the device is reset or powered-down.

The lock-down function is dependent on the $\overline{WP}$ input pin. When $\overline{WP}=0$ (V_{IL}), the blocks in the lock-down state (0,1,x) are protected from program, erase and protection status changes. Device reset or power-down resets all blocks, including those in lock-down, to the locked state.

Because the internal flash memory on the ADSP-BF50xF Blackfin processors has its $\overline{WP}$ signal connected to logic low, the lock-down function is always enabled.

Locking Operations During Erase Suspend

Changes to block lock status can be performed during an erase suspend by using the standard locking command sequences to unlock, lock or lock down a block. This is useful in the case when another block needs to be updated while an erase operation is in progress.

To change block locking during an erase operation, first write the erase suspend command, then check the status register until it indicates that the erase operation has been suspended. Next, write the desired lock command sequence to a block and the lock status changes. After completing any desired lock, read, or program operations, resume the erase operation with the erase resume command.

If a block is locked or locked down during an erase suspend of the same block, the locking status bits change immediately. But when the erase is resumed, the erase operation completes. Locking operations cannot be performed during a program suspend.

Refer to "Command Interface State Tables" on page 6-68 for detailed information on which commands are valid during erase suspend.

Table 6-15. Lock Status

Current Protection Status[1] (D1, D0)		Next Protection Status[1] (D1, D0)		
Current State	**Program/Erase Allowed**	**After Block Lock Command**	**After Block Unlock Command**	**After Block Lock-Down Command**
0,0	yes	0,1	0,0	1,1
0,1[2]	no	0,1	0,0	1,1
1,1	no	1,1	1,1	1,1

1 The lock status is defined by the write protect pin and by D1 ('1' for a locked-down block) and D0 ('1' for a locked block) as read in the read electronic signature command with A1 = V_{IH} and A0 = V_{IL}.

2 All blocks are locked at power-up, so the default configuration is 01.

Block Address Table

Table 6-16 lists the top boot block addresses

Table 6-16. Top Boot Block Addresses

Bank[1]	#	Size (K word)	Address Range
Parameter Bank	0	4	0x203FE000 - 0x203FFFFE
	1	4	0x203FC000 - 0x203FDFFE
	2	4	0x203FA000 - 0x203FBFFE
	3	4	0x203F8000 - 0x203F9FFE
	4	4	0x203F6000 - 0x203F7FFE
	5	4	0x203F4000 - 0x203F5FFE
	6	4	0x203F2000 - 0x203F3FFE
	7	4	0x203F0000 - 0x203F1FFE
	8	32	0x203E0000 - 0x203EFFFE
	9	32	0x203D0000 - 0x203DFFFE
	10	32	0x203C0000 - 0x203CFFFE
	11	32	0x203B0000 - 0x203BFFFE
	12	32	0x203A0000 - 0x203AFFFE
	13	32	0x20390000 - 0x2039FFFE
	14	32	0x20380000 - 0x2038FFFE

Table 6-16. Top Boot Block Addresses (Cont'd)

Bank[1]	#	Size (K word)	Address Range
Bank 1	15	32	0x20370000 - 0x2037FFFE
	16	32	0x20360000 - 0x2036FFFE
	17	32	0x20350000 - 0x2035FFFE
	18	32	0x20340000 - 0x2034FFFE
	19	32	0x20330000 - 0x2033FFFE
	20	32	0x20320000 - 0x2032FFFE
	21	32	0x20310000 - 0x2031FFFE
	22	32	0x20300000 - 0x2030FFFE
Bank 2	23	32	0x202F0000 - 0x202FFFFE
	24	32	0x202E0000 - 0x202EFFFE
	25	32	0x202D0000 - 0x202DFFFE
	26	32	0x202C0000 - 0x202CFFFE
	27	32	0x202B0000 - 0x202BFFFE
	28	32	0x202A0000 - 0x202AFFFE
	29	32	0x20290000 - 0x2029FFFE
	30	32	0x20280000 - 0x2028FFFE
Bank 3	31	32	0x20270000 - 0x2027FFFE
	32	32	0x20260000 - 0x2026FFFE
	33	32	0x20250000 - 0x2025FFFE
	34	32	0x20240000 - 0x2024FFFE
	35	32	0x20230000 - 0x2023FFFE
	36	32	0x20220000 - 0x2022FFFE
	37	32	0x20210000 - 0x2021FFFE
	38	32	0x20200000 - 0x2020FFFE

Table 6-16. Top Boot Block Addresses (Cont'd)

Bank[1]	#	Size (K word)	Address Range
Bank 4	39	32	0x201F0000 - 0x201FFFFE
	40	32	0x201E0000 - 0x201EFFFE
	41	32	0x201D0000 - 0x201DFFFE
	42	32	0x201C0000 - 0x201CFFFE
	43	32	0x201B0000 - 0x201BFFFE
	44	32	0x201A0000 - 0x201AFFFE
	45	32	0x20190000 - 0x2019FFFE
	46	32	0x20180000 - 0x2018FFFE
Bank 5	47	32	0x20170000 - 0x2017FFFE
	48	32	0x20160000 - 0x2016FFFE
	49	32	0x20150000 - 0x2015FFFE
	50	32	0x20140000 - 0x2014FFFE
	51	32	0x20130000 - 0x2013FFFE
	52	32	0x20120000 - 0x2012FFFE
	53	32	0x20110000 - 0x2011FFFE
	54	32	0x20100000 - 0x2010FFFE
Bank 6	55	32	0x200F0000 - 0x200FFFFE
	56	32	0x200E0000 - 0x200EFFFE
	57	32	0x200D0000 - 0x200DFFFE
	58	32	0x200C0000 - 0x200CFFFE
	59	32	0x200B0000 - 0x200BFFFE
	60	32	0x200A0000 - 0x200AFFFE
	61	32	0x20090000 - 0x2009FFFE
	62	32	0x20080000 - 0x2008FFFE

Table 6-16. Top Boot Block Addresses (Cont'd)

Bank[1]	#	Size (K word)	Address Range
Bank 7	63	32	0x20070000 - 0x2007FFFE
	64	32	0x20060000 - 0x2006FFFE
	65	32	0x20050000 - 0x2005FFFE
	66	32	0x20040000 - 0x2004FFFE
	67	32	0x20030000 - 0x2003FFFE
	68	32	0x20020000 - 0x2002FFFE
	69	32	0x20010000 - 0x2001FFFE
	70	32	0x20000000 - 0x2000FFFE

1 There are two bank regions: bank region 1 contains all the banks that are made up of main blocks only; bank region 2 contains the banks that are made up of the parameter and main blocks (parameter bank).

Common Flash Interface

The common flash interface is a JEDEC approved, standardized data structure that can be read from the flash memory device. It allows a system software to query the device to determine various electrical and timing parameters, density information and functions supported by the memory. The system can interface easily with the device, enabling the software to upgrade itself when necessary.

When the read CFI query command is issued the device enters CFI query mode and the data structure is read from the memory. Table 6-17 through Table 6-26 show the addresses used to retrieve the data. The query data is always presented on the lowest order data outputs (`D0-D7`), the other outputs (`D8-D15`) are set to 0.

The CFI data structure also contains a security area where a 64-bit unique security number is written (see Figure 6-2 on page 6-18). This area can be

accessed only in read mode by the final user. It is impossible to change the security number after it has been written by the factory. Issue a read array command to return to read mode.

Table 6-17. Query Structure Overview[1]

Offset	Sub-Section Name	Description
0x00	Reserved	Reserved for algorithm-specific information
0x10	CFI Query Identification String	Command set ID and algorithm data offset
0x1B	System Interface Information	Device timing and voltage information
0x27	Device Geometry Definition	Flash device layout
P	Primary Algorithm-Specific Extended Query table	Additional information specific to the primary algorithm (optional)
A	Alternate Algorithm-Specific Extended Query Table	Additional information specific to the alternate algorithm (optional)
0x80	Security Code Area	Lock protection register unique device number and user programmable OTP

1 The flash memory display the CFI data structure when CFI query command is issued. In this table are listed the main sub-sections detailed in Table 6-18, Table 6-19, Table 6-20, and Table 6-21. Query data is always presented on the lowest order data outputs.

Table 6-18. CFI Query Identification String

Offset	Sub-Section Name	Description	Value
0x00	0x0020	Manufacturer code	
0x01	0x8866	Device code	
0x02	Reserved	Reserved	
0x03	Reserved	Reserved	
0x04-0x0F	Reserved	Reserved	
0x10 0x11 0x12	0x0051 0x0052 0x0059	Query unique ASCII string "QRY"	"Q" "R" "Y"

Table 6-18. CFI Query Identification String (Cont'd)

Offset	Sub-Section Name	Description	Value
0x13 0x14	0x0003 0x0000	Primary algorithm command set and control interface ID code 16 bit ID code defining a specific algorithm	
0x15 0x16	offset = P = 0x0039 0x0000	Address for primary algorithm extended query table (see Table 6-21 on page 6-49)	p = 0x39
0x17 0x18	0x0000 0x0000	Alternate vendor command set and control interface ID code second vendor-specified algorithm supported	N/A
0x19 0x1A	value = A = 0x0000 0x0000	Address for alternate algorithm extended query table	N/A

Table 6-19. CFI Query System Interface Information

Offset	Data	Description	Value
0x1B	0x0017	V_{DD} logic supply minimum program/erase or write voltage bit 7 to 4 BCD value in volts bit 3 to 0 BCD value in 100 millivolts	1.7 V
0x1C	0x0020	V_{DD} logic supply maximum program/erase or write voltage bit 7 to 4 BCD value in volts bit 3 to 0 BCD value in 100 millivolts	2 V
0x1D	0x0085	V_{PP} [programming] supply minimum program/erase voltage bit 7 to 4 HEX value in volts bit 3 to 0 BCD value in 100 millivolts	8.5 V
0x1E	0x0095	V_{PP} [programming] supply maximum program/erase voltage bit 7 to 4 HEX value in volts bit 3 to 0 BCD value in 100 millivolts	9.5 V
0x1F	0x0004	Typical time-out per single byte/word program = 2^n µs	16 µs
0x20	0x0000	Typical time-out for multi-byte programming = 2^n µs	N/A
0x21	0x000A	Typical time-out per individual block erase = 2^n ms	1 s
0x22	0x0000	Typical time-out for full chip erase = 2^n ms	N/A

Table 6-19. CFI Query System Interface Information (Cont'd)

Offset	Data	Description	Value
0x23	0x0003	Maximum time-out for word program = 2^n times typical	128 µs
0x24	0x0000	Maximum time-out for multi-byte programming = 2^n times typical	NA
0x25	0x0002	Maximum time-out per individual block erase = 2^n times typical	4 s
0x26	0x0000	Maximum time-out for chip erase = 2^n times typical	N/A

Table 6-20. Device Geometry Definition

Offset Word Mode		Data	Description	Value
0x27		0x0016	Device size = 2^n in number of bytes	4M bytes
0x28 0x29		0x0001 0x0000	Flash device interface code description	x16 Async.
0x2A 0x2B		0x0000 0x0000	Maximum number of bytes in multi-byte program or page = 2^n	N/A
0x2C		0x0002	Number of identical sized erase block regions within the device bit 7 to 0 = x = number of erase block regions	2
Top Devices	0x2D 0x2E	0x003E 0x0000	Internal flash region 1 information Number of identical-size erase blocks = 0x003E+1	63
		0x007E 0x0000	Internal flash region 1 information Number of identical-size erase blocks = 0x007E+1	127
	0x2F 0x30	0x0000 0x0001	Region 1 information Block size in region 1 = 0x0100 * 256 byte	64K byte
	0x31 0x32	0x0007 0x0000	Region 2 information Number of identical-size erase blocks = 0x0007+1	8
	0x33 0x34	0x0020 0x0000	Region 2 information Block size in region 2 = 0x0020 * 256 byte	8K byte
	0x35 0x38	Reserved for future erase block region information		N/A

Table 6-21. Primary Algorithm-Specific Extended Query Table[1]

Offset	Data	Description	Value
0x(P) = 0x39	0x0050 0x0052 0x0049	Primary algorithm extended query table unique ASCII string “PRI”	"P" "R" "I"
0x(P+3) = 0x3C	0x0031	Major version number, ASCII	"1"
0x(P+4) = 0x3D	0x0033	Minor version number, ASCII	"3"
0x(P+5) = 0x3E	0x00E6	Extended query table contents for primary algorithm. Address 0x(P+5) contains less significant byte.	
	0x0003		
0x(P+7) = 0x40	0x0000	Bit 0 chip erase supported (1 = Yes, 0 = No)	
0x(P+8) = 0x41	0x0000	Bit 1 erase suspend supported (1 = Yes, 0 = No)	No
		Bit 2 program suspend supported (1 = Yes, 0 = No)	Yes
		Bit 3 legacy lock/unlock supported (1 = Yes, 0 = No)	Yes
		Bit 4 queued erase supported (1 = Yes, 0 = No)	No
		Bit 5 instant individual block locking supported (1 = Yes, 0 = No)	No
		Bit 6 protection bits supported (1 = Yes, 0 = No)	Yes
		Bit 7 page mode read supported (1 = Yes, 0 = No)	Yes
		Bit 8 synchronous read supported (1 = Yes, 0 = No)	Yes
		Bit 9 simultaneous operation supported (1 = Yes, 0 = No)	Yes
		Bit 10 to 31 reserved; undefined bits are "0". If bit 31 is "1", then another 31-bit field of optional features follows at the end of the bit-30 field.	Yes
0x(P+9) = 0x42	0x0001	Supported functions after suspend Read array, read status register and CFI query	
		Bit 0 program supported after erase suspend (1 = Yes, 0 = No)	Yes
		Bit 7 to 1 reserved; undefined bits are "0"	

Table 6-21. Primary Algorithm-Specific Extended Query Table[1] (Cont'd)

Offset	Data	Description	Value
0x(P+A) = 0x43	0x0003	Block protect status Defines which bits in the block status register section of the query are implemented.	
0x(P+B) = 0x44	0x0000	Bit 0 block protect status register lock/unlock bit active (1 = Yes, 0 = No	
		Bit 1 block lock status register lock-down bit active (1 = Yes, 0 = No)	Yes
		Bit 15 to 2 reserved for future use; undefined bits are "0"	Yes
0x(P+C) = 0x45	0x0018	V_{DD} logic supply optimum program/erase voltage (highest performance) Bit 7 to 4 HEX value in volts Bit 3 to 0 BCD value in 100 mV	1.8 V
0x(P+D) = 0x46	0x0090	V_{PP} supply optimum program/erase voltage Bit 7 to 4 HEX value in volts Bit 3 to 0 BCD value in 100 mV	9 V

1 The variable P is a pointer that is defined at CFI offset 0x15.

Table 6-22. Protection Register Information[1]

Offset	Data	Description	Value
0x(P+E) = 0x47	0x0001	Number of protection register fields in JEDEC ID space. 0x0000 indicates that 256 fields are available.	1
0x(P+F) = 0x48	0x0080	Protection Field 1: protection description Bits 0-7 lower byte of protection register address Bits 8-15 upper byte of protection register address	0x0080
0x(P+10) = 0x49	0x0000		
0x(P+11) = 0x4A	0x0003	Bits 16-23 2^n bytes in factory pre-programmed region	8 bytes
0x(P+12) = 0x4B	0x0004	Bits 24-31 2^n bytes in user programmable region	16 bytes

1 The variable P is a pointer that is defined at CFI offset 0x15.

Table 6-23. Burst Read Information[1]

Offset	Data	Description	Value
0x(P+13) = 0x4C	0x0003	Page-mode read capability Bits 0-7 'n' such that 2^n HEX value represents the number of read-page bytes. See offset 0x28 for device word width to determine page-mode data output width.	8 bytes
0x(P+14) = 0x4D	0x0004	Number of synchronous mode read configuration fields that follow.	4
0x(P+15) = 0x4E	0x0001	Synchronous mode read capability configuration 1 Bit 3-7 Reserved Bit 0-2 'n' such that 2^n+1 HEX value represents the maximum number of continuous synchronous reads when the device is configured for its maximum word width. A value of 0x07 indicates that the device is capable of continuous linear bursts that will output data until the internal burst counter reaches the end of the device's burstable address space. This field's 3-bit value can be written directly to the read configuration register bit 0-2 if the device is configured for its maximum word width. See offset 0x28 for word width to determine the burst data output width.	4
0x(P+16) = 0x4F	0x0002	Synchronous mode read capability configuration 2	8
0x(P+17) = 0x50	0x0003	Synchronous mode read capability configuration 3	16
0x(P+18) = 0x51	0x0007	Synchronous mode read capability configuration 4	Cont.

1 The variable P is a pointer that is defined at CFI offset 0x15.

Table 6-24. Bank and Erase Block Region Information [1,2]

Internal Flash Memory		Description
Offset	Data	
0x(P+19) = 0x52	0x02	Number of bank regions within the device

1 The variable P is a pointer that is defined at CFI offset 0x15.
2 Bank regions. There are two bank regions, see Table 6-16 on page 6-42.

Table 6-25. Bank and Erase Block Region 1 Information[1]

Internal Flash Region 1		Description
Offset	**Data**	
0x(P+1A) = 0x53	0x07	Number of identical banks within bank region 1
0x(P+1B) = 0x54	0x00	
0x(P+1C) = 0x55	0x11	Number of program or erase operations allowed in bank region 1: Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations
0x(P+1D) = 0x56	0x00	Number of program or erase operations allowed in other banks while a bank in same region is programming Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations
0x(P+1E) = 0x57	0x00	Number of program or erase operations allowed in other banks while a bank in this region is erasing Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations
0x(P+1F) = 0x58	0x01	Types of erase block regions in bank region 1 n = number of erase block regions with contiguous same-size erase blocks. Symmetrically blocked banks have one blocking region.[2]
0x(P+20) = 0x59	0x07	Bank region 1 erase block type 1 information Bits 0-15: n + 1 = number of identical-sized erase blocks Bits 16-31: n × 256 = number of bytes in erase block region
0x(P+21) = 0x5A	0x00	
0x(P+22) = 0x5B	0x00	
0x(P+23) = 0x5C	0x01	
0x(P+24) = 0x5D	0x64	Bank region 1 (erase block type 1) Minimum block erase cycles × 1000
0x(P+25) = 0x5E	0x00	

Table 6-25. Bank and Erase Block Region 1 Information[1] (Cont'd)

Internal Flash Region 1		Description
Offset	Data	
0x(P+26) = 0x5F	0x01	Bank region 1 (erase block type 1): bits per cell, internal ECC Bits 0-3: bits per cell in erase region Bit 4: reserved for "internal ECC used" Bits 5-7: reserved 0x5E 01 0x5E 01
0x(P+27) = 0x60	0x03	Bank region 1 (erase block type 1): page mode and synchronous mode capabilities Bit 0: page-mode reads permitted Bit 1: synchronous reads permitted Bit 2: synchronous writes permitted Bits 3-7: reserved

1 The variable P is a pointer which is defined at CFI offset 0x15.
2 Bank regions. There are two bank regions, see Table 6-16 on page 6-42.

Table 6-26. Bank and Erase Block Region 2 Information[1]

Internal Flash Region 2		Description
Offset	Data	
0x(P+28) = 0x61	0x01	Number of identical banks within bank region 2
0x(P+29) = 0x62	0x00	
0x(P+2A) = 0x63	0x11	Number of program or erase operations allowed in bank region 2 Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations
0x(P+2B) = 0x64	0x00	Number of program or erase operations allowed in other banks while a bank in this region is programming Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations
0x(P+2C) = 0x65	0x00	Number of program or erase operations allowed in other banks while a bank in this region is erasing Bits 0-3: number of simultaneous program operations Bits 4-7: number of simultaneous erase operations

Table 6-26. Bank and Erase Block Region 2 Information[1] (Cont'd)

Internal Flash Region 2		Description
Offset	Data	
0x(P+2D) = 0x66	0x02	Types of erase block regions in bank region 2 n = number of erase block regions with contiguous same-size erase blocks Symmetrically blocked banks have one blocking region.[2]
0x(P+2E) = 0x67	0x06	Bank region 2 erase block type 1 information Bits 0-15: n + 1 = number of identical-sized erase blocks Bits 16-31: n × 256 = number of bytes in erase block region
0x(P+2F) = 0x68	0x00	
0x(P+30) = 0x69	0x00	
0x(P+31) = 0x6A	0x01	
0x(P+32) = 0x6B	0x64	Bank region 2 (erase block type 1) Minimum block erase cycles × 1000
0x(P+33) = 0x6C	0x00	
0x(P+34) = 0x6D	0x01	Bank region 2 (erase block type 1): bits per cell, internal ECC Bits 0-3: bits per cell in erase region Bit 4: reserved for "internal ECC used" Bits 5-7: reserved
0x(P+35) = 0x6E	0x03	Bank region 2 (erase block type 1): page mode and synchronous mode capabilities (defined in Table 6-23 on page 6-51) Bit 0: page-mode reads permitted Bit 1: synchronous reads permitted Bit 2: synchronous writes permitted Bits 3-7: reserved
0x(P+36) = 0x6F	0x07	Bank region 2 erase block type 2 information Bits 0-15: n + 1 = number of identical-sized erase blocks Bits 16-31: n × 256 = number of bytes in erase block region
0x(P+37) = 0x70	0x00	
0x(P+38) = 0x71	0x20	
0x(P+39) = 0x72	0x00	
0x(P+3A) = 0x73	0x64	Bank region 2 (erase block type 2) Minimum block erase cycles × 1000
0x(P+3B) = 0x74	0x00	

Table 6-26. Bank and Erase Block Region 2 Information[1] (Cont'd)

Internal Flash Region 2		Description
Offset	Data	
0x(P+3C) = 0x75	0x01	Bank region 2 (erase block type 2): bits per cell, internal ECC Bits 0-3: bits per cell in erase region Bit 4: reserved for "internal ECC used" Bits 5-7: reserved
0x(P+3D) = 0x76	0x03	Bank region 2 (erase block type 2): page mode and synchronous mode capabilities (defined in Table 6-23 on page 6-51) Bit 0: page-mode reads permitted Bit 1: synchronous reads permitted Bit 2: synchronous writes permitted Bits 3-7: reserved
0x(P+3E) = 0x77		Feature space definitions
0x(P+3F) = 0x78		Reserved

1 The variable P is a pointer which is defined at CFI offset 0x15.
2 Bank regions. There are two bank regions, see Table 6-16 on page 6-42.

Flowcharts and Pseudo Codes

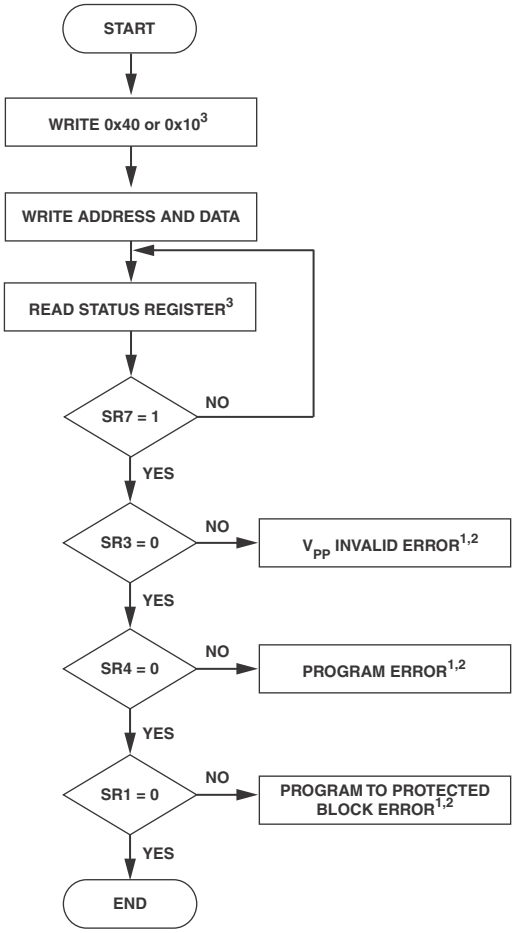


Figure 6-5. Program Flowchart[1,2,3]

1 Status check of SR1 (protected block), SR3 (V_{PP} invalid) and SR4 (program error) can be made after each program operation or after a sequence.
2 If an error is found, the status register must be cleared before further program/erase controller operations.
3 Any address within the bank can equally be used.

Listing 6-1. Program Pseudo Code

```
program_command (addressToProgram, dataToProgram) {:
   "
   writeToFlash (addressToProgram, 0x40);
   /*writeToFlash (addressToProgram, 0x10);*/
      /*see note (3)*/
   "
writeToFlash (addressToProgram, dataToProgram) ;
/*Memory enters read status state after
the Program Command*/

do {
   status_register=readFlash (addressToProgram);
      "see note (3)";
   /* E or G must be toggled*/

   } while (status_register.SR7== 0) ;

if (status_register.SR3==1) /*VPP invalid error */
   error_handler ( ) ;

if (status_register.SR4==1) /*program error */
   error_handler ( ) ;

if (status_register.SR1==1) /*program to protect block error */
   error_handler ( ) ;

}
```

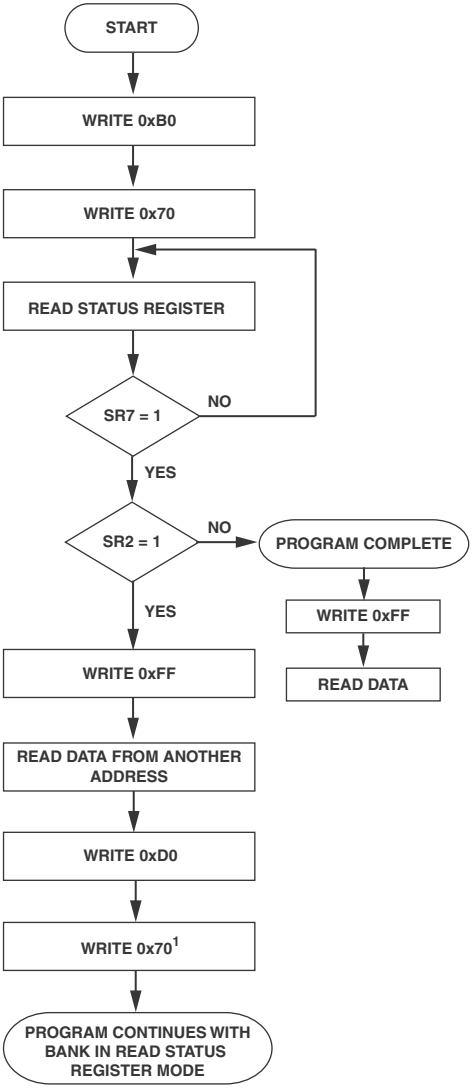


Figure 6-6. Program Suspend and Resume Flowchart[1]

1 The read status register command (write 0x70) can be issued just before or just after the program resume command.

Listing 6-2. Program Suspend and Resume Pseudo Code

```
program_suspend_command ( ) {
   writeToFlash (any_address, 0xB0) ;

   writeToFlash (bank_address, 0x70) ;
   /* read status register to check if
   program has already completed */

do {
   status_register=readFlash (bank_address) ;
   /* E or G must be toggled*/

} while (status_register.SR7== 0) ;

if (status_register.SR2==0) /*program completed */

   { writeToFlash (bank_address, 0xFF) ;
     read_data ( ) ;
     /*The device returns to Read Array
     (as if program/erase suspend was not issued).*/

   }
else
   { writeToFlash (bank_address, 0xFF) ;

     read_data ( ); /*read data from another address*/

     writeToFlash (any_address, 0xD0) ;
     /*write 0xD0 to resume program*/

     writeToFlash (bank_address, 0x70) ;
     /*read status register to check
     if program has completed */
```

```
        }
    }
```

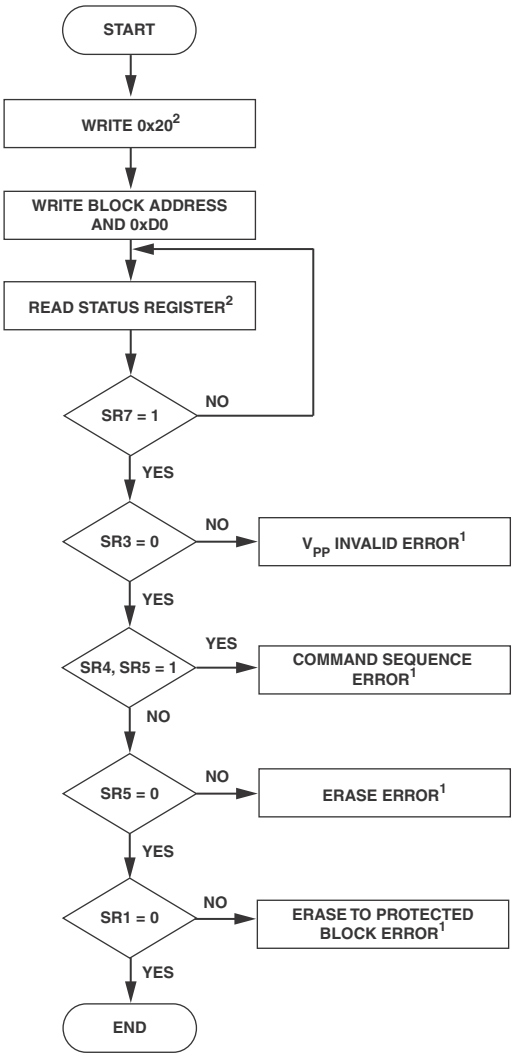


Figure 6-7. Block Erase Flowchart[1],[2]

1 If an error is found, the status register must be cleared before further program/erase operations.
2 Any address within the bank can be used also.

Listing 6-3. Block Erase Pseudo Code

```
erase_command ( blockToErase ) {
   writeToFlash (blockToErase, 0x20) ;
      /*see note (2) */

   writeToFlash (blockToErase, 0xD0) ;
   /* only A12-A20 are significant */
   /* Memory enters read status state after
   the Erase Command */

do {
   status_register=readFlash (blockToErase) ;
      /* see note (2) */
   /* E or G must be toggled*/

   } while (status_register.SR7== 0) ;

if (status_register.SR3==1) /*VPP invalid error */
   error_handler ( ) ;

if ( (status_register.SR4==1) && (status_register.SR5==1) )
/* command sequence error */
   error_handler ( ) ;

if ( (status_register.SR5==1) )
/* erase error */
   error_handler ( ) ;

if (status_register.SR1==1) /*program to protect block error */
   error_handler ( ) ;

}
```

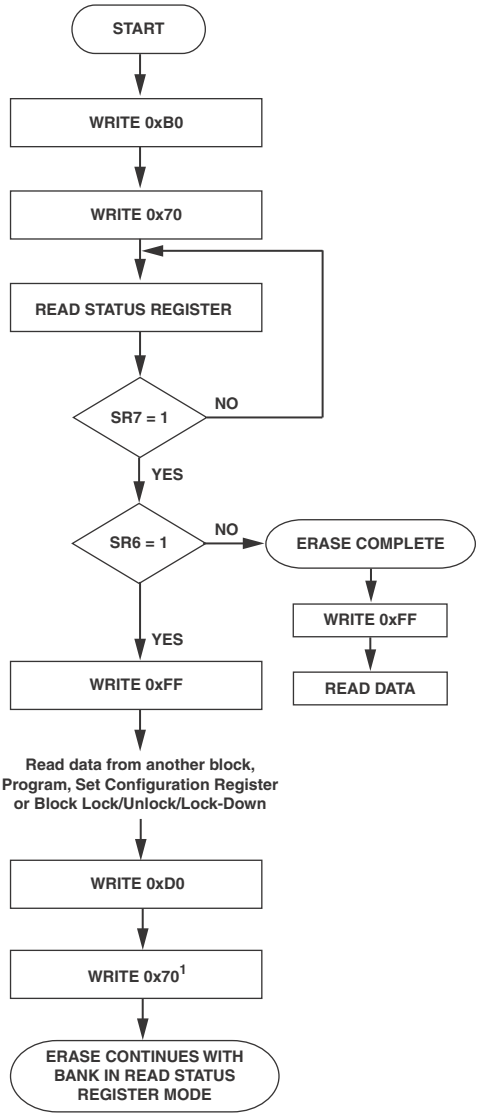


Figure 6-8. Erase Suspend and Resume Flowchart[1]

1 The read status register command (write 0x70) can be issued just before or just after the erase resume command.

Listing 6-4. Erase Suspend and Resume Pseudo Code

```
erase_suspend_command ( ) {
   writeToFlash (bank_address, 0xB0) ;

   writeToFlash (bank_address, 0x70) ;
   /* read status register to check if
   erase has already completed */

do {
   status_register=readFlash (bank_address) ;
   /* E or G must be toggled*/

   } while (status_register.SR7== 0) ;

if (status_register.SR6==0) /*erase completed */

   { writeToFlash (bank_address, 0xFF) ;

      read_data ( ) ;
      /*The device returns to Read Array
      (as if program/erase suspend was not issued).*/
   }
else
   { writeToFlash (bank_address, 0xFF) ;

      read_program_data ( );

      /*read or program data from another block*/

      writeToFlash (bank_address, 0xD0) ;
      /*write 0xD0 to resume erase*/

      writeToFlash (bank_address, 0x70) ;
```

```
      /*read status register to check if erase has completed */
      }
   }
```

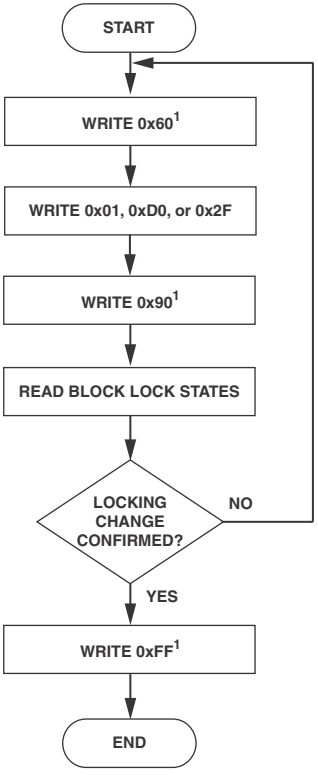


Figure 6-9. Locking Operations Flowchart[1]

1 Any address within the bank can equally be used.

Listing 6-5. Locking Operations Pseudo Code

```
locking_operation_command (address, lock_operation) {
   writeToFlash (address, 0x60) ; /*configuration setup*/
      /* see note (1) */
```

```
if (lock_operation==LOCK) /*to protect the block*/
   writeToFlash (address, 0x01) ;
else if (lock_operation==UNLOCK) /*to unprotect the block*/
   writeToFlash (address, 0xD0) ;
else if (lock_operation==LOCK-DOWN) /*to lock the block*/
   writeToFlash (address, 0x2F) ;

   writeToFlash (address, 0x90) ;
      /*see note (1) */

if (readFlash (address) ! = locking_state_expected)
      error_handler () ;
   /*Check the locking state
     (see Read Block Signature table )*/

   writeToFlash (address, 0xFF) ; /*Reset to Read Array mode*/
      /*see note (1) */

}
```

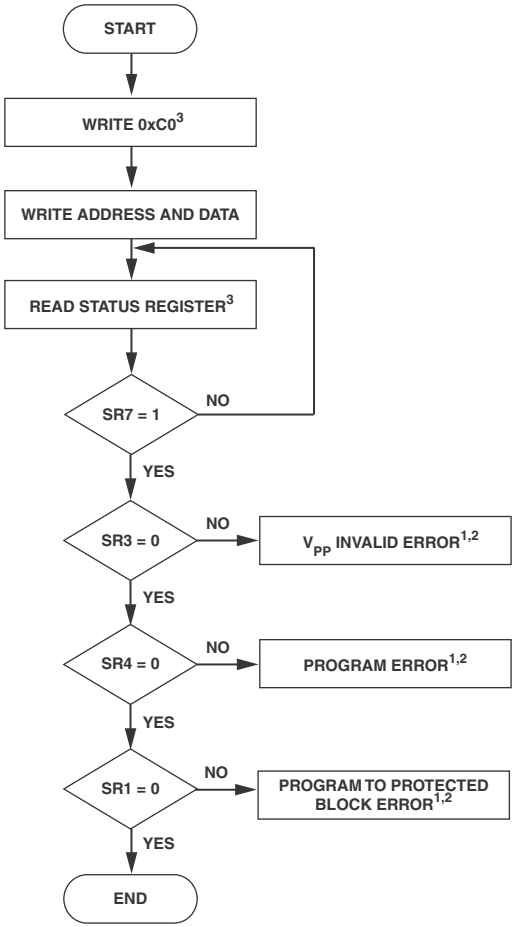


Figure 6-10. Protection Register Program Flowchart[1],[2],[3]

1 Status check of SR1 (protected block), SR3 (V_{PP} invalid) and SR4 (program error) can be made after each program operation or after a sequence.
2 If an error is found, the status register must be cleared before further program/erase controller operations.
3 Any address within the bank can equally be used.

Listing 6-6. Protection Register Program Pseudo Code

```
protection_register_program_command (addressToProgram,
                                     dataToProgram) {:
      writeToFlash (addressToProgram, 0xC0) ;
         /*see note (3) */

      writeToFlash (addressToProgram, dataToProgram) ;
      /*Memory enters read status state after
      the Program Command*/

   do {
      status_register=readFlash (addressToProgram) ;
         /* see note (3) */
      /* E or G must be toggled*/

      } while (status_register.SR7== 0) ;

   if (status_register.SR3==1) /*VPP invalid error */
      error_handler ( ) ;

   if (status_register.SR4==1) /*program error */
      error_handler ( ) ;

   if (status_register.SR1==1) /*program to protect block
      error */
      error_handler ( ) ;
```

Command Interface State Tables

Table 6-27. Command Interface States – Modify Table, Next State

Current CI State[1]		Command Input							
		Read Array[2] (0xFF)	WP setup[3,4], (10/0x40)	Block Erase Setup[3,4], (0x20)	Erase Confirm, P/E Resume, Block Unlock Confirm (0xD0)	Program / Erase Suspend (0xB0)	Read Status Register (0x70)	Clear Status Register[5] (0x50)	Read Electronic Signature, Read CFI Query (0x90, 0x98)
Ready		Ready	Program Setup	Erase Setup	Ready				
Lock/CR Setup		Ready (Lock Error)			Ready	Ready (Lock Error)			
OTP	Setup	OTP Busy							
	Busy	OTP Busy	IS in OTP Busy		OTP Busy				
	IS in OTP Busy	OTP Busy							
Prog.	Setup	Program Busy							
	Busy	Prog. Busy	IS in Program Busy		Prog. Busy	PS	Program Busy		
	IS in Prog. Busy	Program Busy							
	Suspend	PS	IS in Program Suspend		Prog. Busy	Program Suspend			
	IS in PS	Program Suspend							

Table 6-27. Command Interface States – Modify Table, Next State (Cont'd)

Current CI State[1]		Command Input							
		Read Array[2] (0xFF)	WP setup[3,4] (10/0x40)	Block Erase Setup[3,4] (0x20)	Erase Confirm, P/E Resume, Block Unlock Confirm (0xD0)	Program / Erase Suspend (0xB0)	Read Status Register (0x70)	Clear Status Register[5] (0x50)	Read Electronic Signature, Read CFI Query (0x90, 0x98)
Erase	Setup	Ready (Error)			Erase Busy	Ready (error)			
	Busy	Erase Busy	IS in Erase Busy		Erase Busy	ES	Erase Busy		
	IS in Erase Busy	Erase Busy							
	Suspend	ES	Prog. in ES	IS in Erase Suspend	Erase Busy	Erase Suspend			
	IS in ES	Erase Suspend							

Table 6-27. Command Interface States – Modify Table, Next State (Cont'd)

Current CI State[1]		Command Input							
		Read Array[2] (0xFF)	WP setup[3,4] (10/0x40)	Block Erase Setup[3,4] (0x20)	Erase Confirm, P/E Resume, Block Unlock Confirm (0xD0)	Program / Erase Suspend (0xB0)	Read Status Register (0x70)	Clear Status Register[5] (0x50)	Read Electronic Signature, Read CFI Query (0x90, 0x98)
Prog. in ES	Setup	Program Busy in Erase Suspend							
	Busy	Prog. Busy in ES	IS in Program Busy in Erase Suspend		Prog. Busy in ES	PS in ES	Program Busy in Erase Suspend		
	IS in Prog. Busy in ES	Program Busy in Erase Suspend							
	Suspend	PS in ES	IS in Program Suspend in ES		Prog. Busy in ES	Program Suspend in Erase Suspend			
	IS in PS in ES	Program Suspend in Erase Suspend							
Lock/CR Setup in ES		Erase Suspend (Lock Error)			ES	Erase Suspend (Lock Error)			

1 CI = command interface, CR = configuration register, P/E. C. = program/erase controller, PS = program suspend, ES = erase suspend, IS = illegal state.

2 At power-up, all banks are in read array mode. A read array command issued to a busy bank, results in undetermined data output.

3 The two cycle command should be issued to the same bank address.

4 If the P/EC is active, both cycles are ignored.

5 The clear status register command clears the status register error bits except when the P/EC is busy or suspended.

Table 6-28. Command Interface States – Modify Table, Next Output

Current CI State[1,2]	Command Input						
	Read Array[3] (0xFF)	Block Erase Setup[4,5] (0x20)	Erase Confirm P/E Resume, Block Unlock Confirm (0xD0)	Program / Erase Suspend (0xB0)	Read Status Register (0x70)	Clear Status Register[6] (0x50)	Read Electronic Signature, Read CFI Query (0x90, 0x98)
Program Setup	Status Register						
Erase Setup							
OTP Setup							
Program Setup in Erase Suspend							
Lock/CR Setup							
Lock/CR Setup in Erase Suspend							

Table 6-28. Command Interface States – Modify Table, Next Output (Cont'd)

Current CI State[1,2]	Command Input						
	Read Array[3] (0xFF)	Block Erase Setup[4,5] (0x20)	Erase Confirm P/E Resume, Block Unlock Confirm (0xD0)	Program / Erase Suspend (0xB0)	Read Status Register (0x70)	Clear Status Register[6] (0x50)	Read Electronic Signature, Read CFI Query (0x90, 0x98)
OTP Busy	Array	Status Register	Output Unchanged		Status Register	Output Unchanged	Status Register
Ready							Electronic Signature / CFI
Program Busy							
Erase Busy							
Program /Erase Suspend							
Program Busy in Erase Suspend							
Program Suspend in Erase Suspend							
Illegal State	Output Unchanged						

1 CI = command interface, CR = configuration register, P/E. C. = program/erase controller, IS = illegal state, ES = erase suspend, PS = program suspend.

2 The output state shows the type of data that appears at the outputs if the bank address is the same as the command address. A bank can be placed in read array, read status register, read electronic signature or read CFI query mode, depending on the command issued. Each bank remains in its last output state until a new command is issued. The next state does not depend on the bank's output state.

3 At power-up, all banks are in read array mode. A read array command issued to a busy bank, results in undetermined data output.

4 The two cycle command should be issued to the same bank address.

5 If the P/EC is active, both cycles are ignored.
6 The clear status register command clears the status register error bits except when the P/EC is busy or suspended.

Table 6-29. Command Interface States – Lock Table, Next State

Current CI State[1]		Command Input						
		Lock/CR Setup[2] (0x60)	OTP Setup[2] (0xC0)	Block Lock Confirm (0x01)	Block Lock-Down Confirm (0x2F)	Set CR Confirm (0x03)	Illegal Command[3]	P/E. C. Operation Completed
Ready		Lock/CR Setup	OTP Setup	Ready				N/A
Lock/CR Setup		Ready (Lock error)		Ready			Ready (Lock error)	N/A
OTP	Setup	OTP Busy						
	Busy	IS in OTP busy		OTP Busy				Ready
	IS in OTP busy	OTP Busy						IS Ready
Program	Setup	Program Busy						N/A
	Busy	IS in Program busy		Program Busy				Ready
	IS in Program busy	Program busy						IS Ready
	Suspend	IS in PS		Program Suspend				N/A
	IS in PS	Program Suspend						N/A
Erase	Setup	Ready (error)						N/A
	Busy	IS in Erase Busy		Erase Busy				Ready
	IS in Erase Busy	Erase Busy						IS Ready
	Suspend	Lock/CR Setup in ES	IS in Erase Suspend	Erase Suspend				N/A
	IS in ES	Erase Suspend						N/A

Table 6-29. Command Interface States – Lock Table, Next State (Cont'd)

Current CI State[1]		Command Input						
		Lock/CR Setup[2] (0x60)	OTP Setup[2] (0xC0)	Block Lock Confirm (0x01)	Block Lock-Down Confirm (0x2F)	Set CR Confirm (0x03)	Illegal Command[3]	P/E. C. Operation Completed
Program in Erase Suspend	Setup	Program Busy in Erase Suspend						
	Busy	IS in Program busy in ES		Program Busy in Erase Suspend				ES
	IS in Program busy in ES	Program busy in ES						IS in ES
	Suspend	IS in PS in ES		Program Suspend in Erase Suspend				N/A
	IS in PS in ES	Program Suspend in Erase Suspend						
Lock/CR Setup in ES		Erase Suspend (Lock error)		Erase Suspend			Erase Suspend (Lock error)	N/A

1 CI = command interface, CR = configuration register, enhanced factory program, P/E. C. = program/erase controller, IS = illegal state, ES = erase suspend, PS = program suspend
2 If the P/EC is active, both cycles are ignored.
3 Illegal commands are those not defined in the command set.

Table 6-30. Command Interface States – Lock Table, Next Output

Current CI State[1]	Command Input						
	Lock/CR Setup[2] (0x60)	OTP Setup[2] (0xC0)	Block Lock Confirm (0x01)	Block Lock-Down Confirm (0x2F)	Set CR Confirm (0x03)	Illegal Command[3]	P/E. C. Operation Completed
Program Setup	Status Register						Output Unchanged
Erase Setup							
OTP Setup							
Program Setup in Erase Suspend							
EFP Setup							
EFP Busy							
EFP Verify							
Quad EFP Setup							
Quad EFP Busy							
Lock/CR Setup	Status Register				Array	Status Register	
Lock/CR Setup in Erase Suspend							

Table 6-30. Command Interface States – Lock Table, Next Output (Cont'd)

Current CI State[1]	Command Input						
	Lock/CR Setup[2] (0x60)	OTP Setup[2] (0xC0)	Block Lock Confirm (0x01)	Block Lock-Down Confirm (0x2F)	Set CR Confirm (0x03)	Illegal Command[3]	P/E. C. Operation Completed
OTP Busy	Status Register		Output Unchanged				
Ready							
Program Busy							
Erase Busy							
Program/ Erase Suspend							
Program Busy in Erase Suspend							
Program Suspend in Erase Suspend							
Illegal State	Output Unchanged						

1 CI = command interface, CR = configuration register, P/E. C = program/erase controller
2 If the P/EC is active, both cycles are ignored.
3 Illegal commands are those not defined in the command set.

Internal Flash Memory Programming Guidelines

The following sections describe programming guidelines for the internal flash memory:

In these sections, references are made to the following parameters that describe the timing characteristics of the EBIU:

- Setup (ST): the time between the beginning of a memory cycle ($\overline{\text{AMSO}}$) and the read-enable assertion ($\overline{\text{ARE}}$) or write-enable assertion ($\overline{\text{AWE}}$)
- Read Access (RAT): the time between read-enable assertion ($\overline{\text{ARE}}$) and deassertion ($\overline{\text{ARE}}$)
- Write Access (WAT): the time between write-enable assertion ($\overline{\text{AWE}}$) and deassertion ($\overline{\text{AWE}}$)

- Hold (HT): the time between read-enable deassertion ($\overline{\text{ARE}}$) or write-enable deassertion ($\overline{\text{AWE}}$) and the end of the memory cycle ($\overline{\text{AMSO}}$)
- Transition (TT): the time between a read access in the current bank and a write access to the current bank or a read access to a different bank.

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

- ST ≥ 1 cycle
- RAT ≥ 1 cycle
- WAT ≥ 1 cycle
- HT ≥ 0 cycle
- TT ≥ 1 cycle

Bringing Internal Flash Memory Out of Reset

The $\overline{\text{RP}}$ pin of the internal flash memory device is controlled by bit 0 of the FLASH_CONTROL register. Setting bit 0 of the FLASH_CONTROL register to 1 enables the flash by bringing it out of reset.

Refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for the timing requirements needed to bring the internal flash memory out of reset. A minimum time (listed in the data sheet) should elapse before the signals $\overline{\text{W}}$, $\overline{\text{E}}$, $\overline{\text{G}}$, and $\overline{\text{L}}$ are asserted, and a minimum time should elapse before the the $\overline{\text{RP}}$ signal is toggled again. These timing requirements may be met by inserting an appropriate number of delay

instructions. For example, the code below uses a number of NOP assembly instructions in order to achieve the desired delay:

```
void flash_reset(void)
{
    /* Reset the flash */
    *pFLASH_CONTROL_CLEAR = FLASH_ENABLE;
    asm("ssync;nop;nop;nop;nop;nop;nop;nop;");
    asm("ssync;nop;nop;nop;nop;nop;nop;nop;");
    asm("ssync;nop;nop;nop;nop;nop;nop;nop;");
    asm("ssync;nop;nop;nop;nop;nop;nop;nop;");

    /* Release flash from reset state */
    *pFLASH_CONTROL_SET = FLASH_ENABLE;
    asm("ssync;nop;nop;nop;nop;nop;nop;nop;");
}
```

Timing Configurations for Setting the Internal Flash Memory in Asynchronous Read Mode

Once out of reset, the internal flash memory is configured in asynchronous mode. Therefore, the EBIU should be configured in asynchronous mode as well by programming the BOMODE field in the EBIU_MODECTL register to the value b#01.

The internal flash device's WAIT signal is not meaningful in asynchronous mode. Therefore, the BORDYEN field in the EBIU_AMBCTL register should be programmed with a value of 0 in asynchronous mode.

Based on the timing requirements of the internal flash memory device, which call for:

- ST > 10 ns
- ST + RAT > 70 ns

- RAT > 30 ns
- HT (for consecutive reads) = 0 ns

The recommended timing values to be programmed in the EBIU_AMBCTL register for asynchronous read accesses are:

- B0ST = ceiling (20 ns / t_{SCLK})
- B0RAT = ceiling (60 ns / t_{SCLK})
- B0TT = b#01
- B0HT = b#00 (see Note)

where ceiling (*x*) is the smallest integer not less than *x*.

There is no hold time requirement for read accesses. Therefore, if the flash access pattern is such that only read accesses are performed with no write accesses performed, then B0HT may be programmed to the value b#00. However, if there were any write accesses interspersed with the read accesses, then the B0HT field should be programmed according to the recommendation for write accesses. (See "Timing Configurations for Setting the Internal Flash Memory for Write Accesses" on page 6-80.)

Timing Configurations for Setting the Internal Flash Memory for Write Accesses

Based on the timing requirements of the internal flash memory device, which call for:

- ST > 10 ns
- WAT > 45 ns
- ST + HT (for consecutive writes) > 25 ns

The recommended timing values to be programmed in the EBIU_AMBCTL register for asynchronous write accesses are:

- B0ST = ceiling (20 ns / t_{SCLK})
- B0WAT = ceiling (45 ns / t_{SCLK})
- B0HT = ceiling (10 ns / t_{SCLK})

In addition to the above timing requirements, a minimum of 25 ns should elapse between completing a write access and starting a read access in the targeted bank or between reading following a Set Configuration Register command. System designers should take this into account and may insert software NOP instructions to delay the first read in the same bank after issuing any command and to delay the first read to any address after issuing a Set Configuration Register command. If the first read after the command is a read array operation in a different bank and no changes to the configuration register have been issued, then the 25 ns delay is not necessary.

Example calculation for asynchronous mode, assuming that f_{SCLK} = 100 MHz or t_{SCLK} = 10 ns:

- WAT ≥ ceiling (45 ns / 10 ns) = ceiling (4.5) = 5
- RAT ≥ ceiling (60 ns / 10 ns) = ceiling (6) = 6
- ST ≥ ceiling (20 ns / 10 ns) = ceiling (2) = 2
- HT = 1
- TT = 1

Figure 6-11 shows example asynchronous read and write waveforms for the internal flash. The signal names referenced in the figure are explained in Table 6-1 on page 6-2.

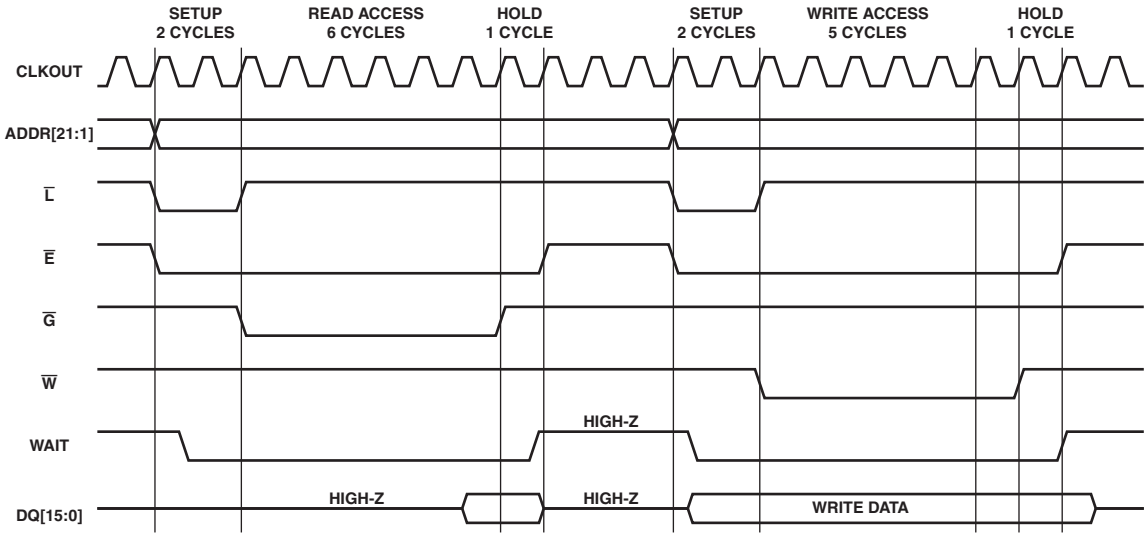


Figure 6-11. Example Asynchronous Read and Write Waveforms

Enabling the Program or Erasure of Internal Flash Memory Blocks

In order to enable program or erase operations in the internal flash memory device, the FLASH_UNPROTECT bit in the FLASH_CONTROL register has to be set to 1. Doing so disables the hardware flash protection mechanism by driving the flash's V_{PP} signal with logic high.

In addition, since the default status of all blocks on flash power-up or after a flash hardware reset is locked, further block unlock commands are necessary in order to allow for software programming or erasure of flash. Please refer to "Block Locking" on page 6-38 for further details on block locking and unlocking and to "Command Interface – Standard Commands" on

page 6-7 for information on flash commands including the block unlock command.

Configuring Internal Flash Memory for Synchronous Burst Read Mode

In order to operate the internal flash device in synchronous burst mode, both the internal flash device and the EBIU have to be set in synchronous mode. The internal flash device is set in synchronous burst mode through the use of the Set Configuration Register command. The EBIU is set in synchronous mode by programming the `BOMODE` field in the `EBIU_MODECTL` register to the value `b#11` and by programming the timing configurations in the EBIU registers as shown in section "Configuring the EBIU for Synchronous Read Mode" on page 6-85.

As shown in Table 6-5 (Standard Commands), two write cycles are required to issue the Set Configuration Register command.

The first cycle writes the setup command to the address corresponding to the value that is to be programmed into the configuration register. The second cycle writes the confirm command to the address corresponding to the value that is to be programmed into the configuration register.

For both cycles, the address corresponding to the value that is to be programmed into the configuration register is:

`FLASH_BASE_ADDRESS` + (`configuration_register_value` << 1)

where:

`FLASH_BASE_ADDRESS` is the base address of the flash device. On ADSP-BF50x devices, this address is 0x20000000 and `configuration_register_value` is the value to be programmed into the flash's configuration register.

Because the flash device is 2-bytes addressable while the Blackfin processor is 1-byte addressable, the value to be programmed into the flash's configuration register has to be shifted up by one so it will appear on the Blackfin processor's address bits [16:1] thus appearing on the flash device's address bits [15:0].

Supported Configuration Register Combinations in ADSP-BF50xF Processors

Since the internal flash device in ADSP-BF50xF processors can only be connected to the external bus interface unit (EBIU), some combinations of the flash configurations are not supported. Some of the restrictions on the values to be programmed in the flash configuration register are as follows:

- The programming of bit `CR10` determines the programming of bit 1 (`BORDYPOL`) of the `EBIU_AMBCTL0` register. When `CR10` is set to 1, `BORDYPOL` has to be programmed to 0. When `CR10` is programmed to 0, `BORDYPOL` has to be set to 1.
- `CR9` has to be programmed to 0.
- `CR8` has to be programmed to 1.
- `CR7` has to be programmed to 1.
- `CR6` has to be programmed to 1.
- `CR3` has to be programmed to 0.

- CR2 through CR0 have to be programmed to b#011.
- The value to be programmed in the X latency bit field (CR13 through CR11) depends on the NOR_CLK frequency, as shown in Table 6-31.

Table 6-31. X Latency Setting Depends on Frequency

NOR_CLK Frequency	X Latency (in Terms of NOR_CLK Cycles)
≤ 30 MHz	≥ 2
≤ 40 MHz	≥ 3
≤ 50 MHz	≥ 4

Configuring the EBIU for Synchronous Read Mode

In order to support internal flash operation in synchronous burst mode, the EBIU has to be configured in synchronous burst mode by programming the BOMODE field in the EBIU_MODECTL register to the value b#11, selecting the appropriate NOR_CLK frequency in the BCLK bit field of the EBIU_FCTL register, and configuring the EBIU_AMBCTL register as follows:

- BORDYEN must be programmed to 1 for synchronous burst read mode.
- BORDYPOL must be set to 1 if bit 10 of the flash configuration register (CR10) is programmed to 0 and programmed to 0 if bit 10 of the flash configuration register (CR10) is set to 1.
- BOTT must be set to b#11.

- B0ST must be programmed depending on the NOR_CLK frequency selected in the EBIU_FCTL register as shown in the following table:

SCLK:NOR_CLK	Min Setup Time	B0ST Values Supported	B0ST Value Recommended
2 : 1	2 SCLK cycles	10,11,00	10
3 : 1	3 SCLK cycles	11,00	11
4 : 1	4 SCLK cycles	00	00

- B0HT may be programmed to any supported value, but should be programmed to the recommended value of b#00.
- B0RAT must be programmed, depending on the NOR_CLK frequency selected in the EBIU_FCTL register and on the X latency setting selected in the flash configuration register (CR13-CR11) according to the following table:

SCLK:NOR_CLK	B0RAT Value
2 : 1	2 * X latency
3 : 1	(3 * X latency) -1
4 : 1	(4 * X latency) -1

Example synchronous read and write waveforms using the internal flash memory pins from Table 6-1 on page 6-2 appear in Figure 6-12.

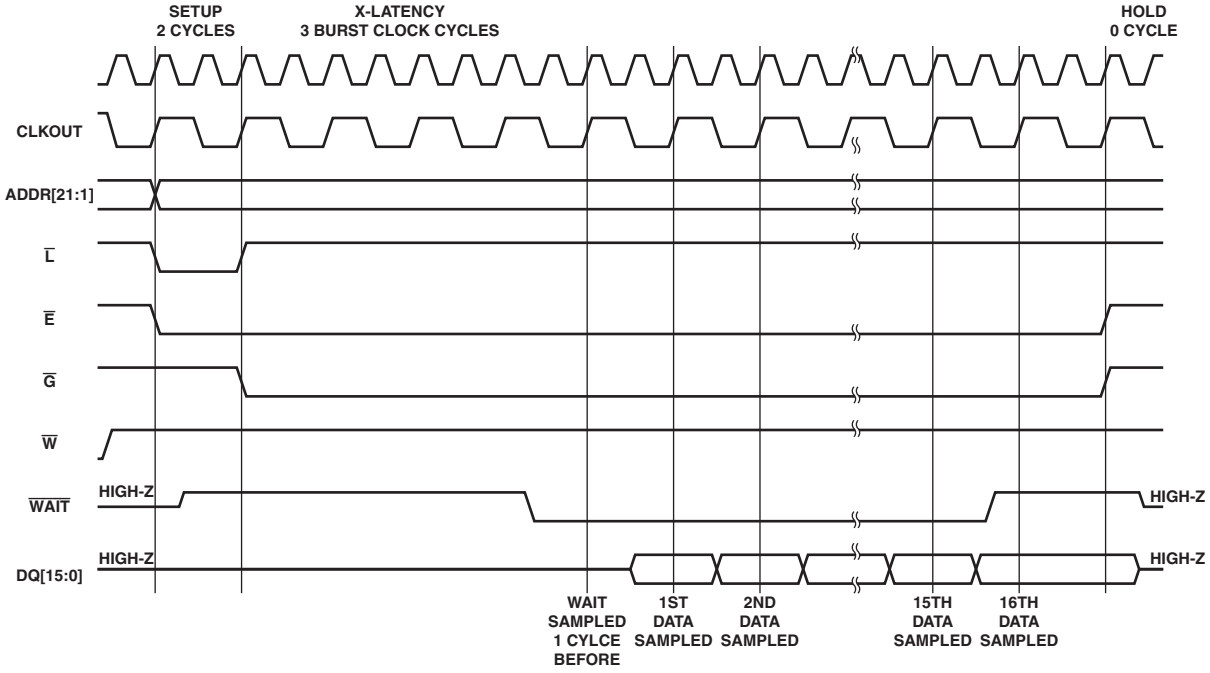


Figure 6-12. Example Sync Read and Write Waveforms

Unsupported Programming Practices in Flash

The following programming practices are unsupported by the internal flash memory:

- Writes to flash are not supported when the flash is configured in synchronous burst mode. Writes to flash can only be performed when the flash is configured in asynchronous mode.
- The flash is only addressable and programmed at 2-bytes granularity. Therefore, any byte write instructions with the destination address residing in flash shall be avoided.

- The TESTSET assembly instruction of the Blackfin processor cannot be used with a variable that resides in the internal flash memory.
- DMA writes to internal flash memory shall be avoided.
- While the internal flash memory is supported by the Blackfin processor cache, no writes from cache to flash are supported. Therefore, cache may only be used to support code and/or read-only data in flash. For the same reason, the FLUSH and FLUSHINV data cache flush instructions cannot be used when the flash memory is the destination of the data in cache.

Internal Flash Memory Control Registers

In addition to the EBIU registers (see "EBIU Registers" on page 5-9), the internal flash memory usage is controlled by the following registers:

- "Internal Flash Memory Control (FLASH_CONTROL) Register"
- "Internal Flash Memory Control Set (FLASH_CONTROL_SET) Register" on page 6-91
- "Internal Flash Memory Control Clear (FLASH_CONTROL_CLEAR) Register" on page 6-91

Internal Flash Memory Control (FLASH_CONTROL) Register

The FLASH_CONTROL register address and reset value are:

Address	Register Name	Size	Reset Value
0xFFC0 328C	FLASH_CONTROL	16	0x8000

Using the bits in the FLASH_CONTROL register (see Table 6-32) permits control of the internal flash memory.

Table 6-32. Internal Flash Memory Control Register (FLASH_CONTROL)

Register Field Name	Offset	Access	Description
FLASH_ENABLE	0	RO	Enable internal flash memory for read/write 0 – internal flash memory is in reset state 1 – internal flash memory is out of reset state
RESERVED	7:1	RO	
FLASH_UNPROTECT	8	RO	0 – Protect the entire internal flash memory from any program/erase by controlling V_{PP} 1 – Unprotect the internal flash memory – internal flash memory is available for program/erase
RESERVED	14:9	RO	
UNLOCK_HIGHBYTE	15	RO	0 – Bits 15:8 of the FLASH_CONTROL register cannot be set to 1. They can only be cleared (programmed to 0). 1 – No restrictions on the programming of bits 15:8 of the FLASH_CONTROL register. They can both be set to 1 or cleared to 0.

Bits 14 to 0 of the FLASH_CONTROL register reset at hardware reset ($\overline{\text{RESET}}$ pin), system reset, watchdog reset, or core double-fault reset. Bit 15 only resets to a 1 at hardware reset ($\overline{\text{RESET}}$ pin) or when going into hibernate.

The FLASH_ENABLE bit controls the $\overline{\text{RP}}$ pin of the internal flash memory device. Since the reset value of the FLASH_ENABLE bit is 0, the default state of the internal flash memory device is the reset state. However, upon Blackfin processor boot, the boot firmware brings the flash out of reset state by programming the FLASH_ENABLE bit to 1.

When the Blackfin processor is in hibernate state, the internal flash memory device is placed in reset state by driving the $\overline{\text{RP}}$ signal with a logic-low.

Please refer to "Bringing Internal Flash Memory Out of Reset" on page 6-78 for more information on using the FLASH_ENABLE bit to reset the internal flash memory device.

Since the reset value of the UNLOCK_HIBYTE bit is 1, bit [15:8] of the FLASH_CONTROL register can be programmed after reset to either 1 or 0. However, once the UNLOCK_HIBYTE bit is programmed to 0, bits [15:8] in the FLASH_CONTROL register can no longer be set to 1; they can only be cleared (programmed to 0).

This UNLOCK_HIBYTE feature may be used to provide a level of protection against inadvertent programming/erasure of the flash. Since the reset value of the FLASH_UNPROTECT bit (bit 8 in the FLASH_CONTROL register) is 0, the internal flash memory device is protected by default against programming and erasure. If the user chose to ensure the continued protection of flash, then he/she can program the UNLOCK_HIBYTE and the FLASH_UNPROTECT bits to 0 to prevent the ability to further set the FLASH_UNPROTECT bit to 1, thus effectively locking the current flash image.

When the Blackfin processor is in hibernate state, all internal flash memory signals, except $\overline{\text{RP}}$, are three-stated. This is the lowest power consumption mode. After the Blackfin processor exits hibernate state, the FLASH_CONTROL register is programmed again to its reset value.

Internal Flash Memory Control Set (FLASH_CONTROL_SET) Register

Writing to a bit in the FLASH_CONTROL_SET register sets the corresponding bit in the internal flash memory control register. Reads return the internal flash memory control register value.

Address	Register Name	Size	Reset Value
0xFFC0 3290	FLASH_CONTROL_SET	16	0x8000

Internal Flash Memory Control Clear (FLASH_CONTROL_CLEAR) Register

Writing to a bit in the FLASH_CONTROL_CLEAR register clears the corresponding bit in the internal flash memory control register. Reads return the internal flash memory control register value.

Address	Register Name	Size	Reset Value
0xFFC0 3294	FLASH_CONTROL_CLEAR	16	0x8000

7 DIRECT MEMORY ACCESS

This chapter describes the direct memory access (DMA) controller. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in Chapter 3, "Chip Bus Hierarchy".

Specific Information for the ADSP-BF50x

For details regarding the number of DMA controllers for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For DMA interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the DMAs is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Chapter A, "System MMR Assignments".

DMA controller behavior for the ADSP-BF50x that differs from the general information in this chapter can be found in the section "Unique Information for the ADSP-BF50x Processor" on page 7-105.

Overview and Features

The processor uses DMA to transfer data between memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity.

The DMA controller can perform several types of data transfers:

- Peripheral DMA transfers data between memory and on-chip peripherals.
- Memory DMA (MDMA) transfers data between memory and memory. The processor has two MDMA modules, each consisting of independent memory read and memory write channels.
- Handshaking memory DMA (HMDMA) transfers data between off-chip peripherals and memory. This enhancement of the MDMA channels enables external hardware to control the timing of individual data transfers or block transfers.

The HMDMA feature is not available for all products. Refer to "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine whether it applies to this product.

All DMAs can transport data to and from on-chip and off-chip memories, including L1 and SDRAM. The L1 scratchpad memory cannot be accessed by DMA.

SDRAM and SRAM are not available on all products. Refer to "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine whether it applies to this product.

DMA transfers on the processor can be descriptor-based or register-based.

Register-based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes.

Examples of DMA styles supported by flex descriptors include:

- A single linear buffer that stops on completion (`FLOW` = stop mode)
- A linear buffer with byte strides of any integer value, including negative values (`DMAx_X_MODIFY` register)
- A circular, auto-refreshing buffer that interrupts on each full buffer
- A similar buffer that interrupts on fractional buffers (for example, ½, ¼) (2-D DMA)
- 1-D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing a link pointer and a 32-bit address
- 1-D DMA, using a linked list of 5-word descriptors containing a link pointer, a 32-bit address, the buffer length, and a configuration

- 2-D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page
- 2-D DMA, using a linked list of 9-word descriptors specifying everything

DMA Controller Overview

A block diagram of the DMA controller can be found in the "Unique Information for the ADSP-BF50x Processor" on page 7-105.

External Interfaces

The DMA does not connect external memories and devices directly. Rather, data is passed through the EBIU port. Any kind of device that is supported by the EBIU can also be accessed by peripheral DMA or memory DMA operation. This is typically flash memory, SRAM, SDRAM, FIFOs, or memory-mapped peripheral devices.

For products with handshaking MDMA (HMDMA), the operation is supported by two MDMA request input pins, `DMAR0` and `DMAR1`. The `DMAR0` pin controls transfer timing on the `MDMA0` destination channel. The `DMAR1` pin controls the destination channel of `MDMA1`. With these pins, external FIFO devices, ADC or DAC converters, or other streaming or block-processing devices can use the MDMA channels to exchange their data or data buffers with the Blackfin processor memory.

Internal Interfaces

Figure 3-1 on page 3-3 shows the dedicated DMA buses used by the DMA controller to interconnect L1 memory, the on-chip peripherals, and the EBIU port.

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

The 16-bit DMA access bus (DAB) connects the DMA controller to the on-chip peripherals. This bus operates at SCLK frequency.

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

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

Memory DMA can pass data every SCLK cycle between L1 memory and the EBIU. Transfers from L1 memory to L1 memory require two cycles, as the DCB bus is used for both source and destination transfers. Similarly, transfers between two off-chip devices require EBIU and DEB resources twice. Peripheral DMA transfers can be performed every other SCLK cycle.

For more details on DMA performance see "DMA Performance" on page 7-41.

Peripheral DMA

The DMA controller features 12 channels that perform transfers between peripherals and on-chip or off-chip memories. The user has full control over the mapping of DMA channels and peripherals. The default DMA channel priority and mapping, shown in Table 7-7 on page 7-107, can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

The default configuration should suffice in most cases, but there are some cases where remapping the assignment can be helpful because of the DMA channel priorities. When competing for any of the system buses, DMA0 has higher priority than DMA1, and so on. DMA11 has the lowest priority of the peripheral DMA channels.

A 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

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

Every DMA channel features its own 4-deep FIFO that decouples DAB activity from DCB and DEB availability. DMA interrupt and descriptor fetch timing is aligned with the memory side (DCB/DEB side) of the FIFO. The user does, however, have an option to align interrupts with the peripheral side (DAB side) of the FIFO for transmit operations. Refer to the SYNC bit in the DMAx_CONFIG register for details.

Memory DMA

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

Each MDMA channel contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the DCB and DEB

buses. Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from the boot ROM on the DEB bus. The FIFO can be used to hold DMA data transferred between two L1 memory locations or between two external memory locations.

Each page of MDMA channels consists of:

- A source channel (for reading from memory)
- A destination channel (for writing to memory)

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

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

MDMA0 takes precedence over MDMA1, unless round-robin scheduling is used or priorities become urgent, as programmed by the `DRQ` bit field in the `HMDMA_CONTROL` register.

It is illegal to program a source channel for memory write or a destination channel for memory read.

The channels support 8-, 16-, and 32-bit memory DMA transfers, but both ends of the MDMA connect to 16-bit buses. Source and destination channels must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of the MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word deep 16-bit FIFO. The source DMA engine fills the FIFO, while the

destination DMA engine empties it. The FIFO depth allows the burst transfers of the external access bus (EAB) and DMA access bus (DAB) to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total transfer count is the same.

To start a MDMA transfer operation, the MMRs for the source and destination channels are written, each in a manner similar to peripheral DMA.

The `DMAx_CONFIG` register for the source channel must be written before the `DMAx_CONFIG` register for the destination channel.

Handshaked Memory DMA (HMDMA) Mode

This feature is not available for all products. Refer to "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine whether it applies to this product.

Handshaked operation applies only to memory DMA channels.

Normally, memory DMA transfers are performed at maximum speed. Once started, data is transferred in a continuous manner until either the data count expires or the MDMA is stopped. In handshake mode, the MDMA does not transfer data automatically when enabled; it waits for an external trigger on the MDMA request input signals. The `DMAR0` input is associated with MDMA0 and the `DMAR1` input with MDMA1. Once a trigger event is detected, a programmable portion of data is transferred and then the MDMA stalls again and waits for the next trigger.

Handshake operation is not only useful for controlling the timing of memory-to-memory transfers, it also enables the MDMA to operate with

asynchronous FIFO-style devices connected to the EBIU port. The Blackfin processor acknowledges a DMA request by a proper number of read or write operations. It is up to the device connected to any of the AMSx strobes to deassert or pulse the request signal and to decrement the number of pending requests accordingly.

Depending on HMDMA operating mode, an external DMA request may trigger individual data word transfers or block transfers. A block can consist of up to 65535 data words. For best throughput, DMA requests can be pipelined. The HMDMA controllers feature a request counter to decouple request timing from the data transfers.

See "Handshaked Memory DMA Operation" on page 7-37 for a functional description.

Modes of Operation

The following sections describe the DMA operation.

Register-Based DMA Operation

Register-based DMA is the traditional kind of DMA operation. Software configures the source or destination address and the length of the data to be transferred to memory-mapped registers and then starts DMA operation.

For basic operation, the software performs these steps:

- Write the source or destination address to the 32-bit DMAx_START_ADDR register.
- Write the number of data words to be transferred to the 16-bit DMAx_X_COUNT register.

- Write the address modifier to the 16-bit DMAx_X_MODIFY register. This is the two's-complement value added to the address pointer after every transfer. This value must always be initialized as there is no default value. Typically, this register is set to 0x0004 for 32-bit DMA transfers, to 0x0002 for 16-bit transfers, and to 0x0001 for byte transfers.
- Write the operation mode to the DMAx_CONFIG register. These bits in particular need to be changed as needed:
    - The DMAEN bit enables the DMA channel.
    - The WNR bit controls the DMA direction. DMAs that read from memory (peripheral transmit DMAs and source channel MDMAs) keep this bit cleared. Peripheral receive DMAs and destination channel MDMAs set this bit because they write to memory.
    - The WDSIZE bit controls the data word width for the transfer. It can be 8-, 16-, or 32-bits wide.
    - The DI_EN bit enables an interrupt when the DMA operation has finished.
    - Set the FLOW field to 0x0 for stop mode or 0x1 for autobuffer mode.

Once the DMAEN bit is set, the DMA channel starts its operation. While running, the DMAx_CURR_ADDR and the DMAx_CURR_X_COUNT registers can be monitored to determine the current progress of the DMA operation. However they should not be used to synchronize software and hardware.

The DMAx_IRQ_STATUS register signals whether the DMA has finished (DMA_DONE bit), whether a DMA error has occurred (DMA_ERR bit), and whether the DMA is currently running (DMA_RUN bit). The DMA_DONE and the DMA_ERR bits also function as interrupt latch bits. They must be cleared by write-one-to-clear (W1C) operations by the interrupt service routine.

Stop Mode

In stop mode, the DMA operation is executed only once. When started, the DMA channel transfers the desired number of data words and stops itself when the transfer is complete. If the DMA channel is no longer used, software should clear the DMAEN enable bit to disable the otherwise paused channel. Stop mode is entered if the FLOW bit field in the DMA channel's DMAx_CONFIG register is 0. The NDSIZE field must always be 0 in this mode.

For receive (memory write) operation, the DMA_RUN bit functions almost the same as the inverted DMA_DONE bit. For transmit (memory read) operation, however, the two bits have different timing. Refer to the description of the SYNC bit in the DMAx_CONFIG register for details.

Autobuffer Mode

In autobuffer mode, the DMA operates repeatedly in a circular manner. If all data words have been transferred, the address pointer DMAx_CURR_ADDR is reloaded automatically by the DMAx_START_ADDR value. An interrupt may also be generated.

Autobuffer mode is entered if the FLOW field in the DMAx_CONFIG register is 1. The NDSIZE bit must be 0 in autobuffer mode.

Two-Dimensional DMA Operation

Register-based and descriptor-based DMA can operate in one-dimensional mode or two-dimensional mode.

In two-dimensional (2-D) mode, the DMAx_X_COUNT register is accompanied by the DMAx_Y_COUNT register, supporting arbitrary row and column sizes up to 64K × 64K elements, as well as arbitrary DMAx_X_MODIFY and DMAx_Y_MODIFY values up to ±32K bytes. Furthermore, DMAx_Y_MODIFY can be negative, allowing implementation of interleaved datastreams. The DMAx_X_COUNT and

DMAx_Y_COUNT values specify the row and column sizes, where DMAx_X_COUNT must be 2 or greater.

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

The DMAx_X_MODIFY value is the byte-address increment that is applied after each transfer that decrements the DMAx_CURR_X_COUNT register. The DMAx_X_MODIFY value is not applied when the inner loop count is ended by decrementing DMAx_CURR_X_COUNT from 1 to 0, except that it is applied on the final transfer when DMAx_CURR_Y_COUNT is 1 and DMAx_CURR_X_COUNT decrements from 1 to 0.

The DMAx_Y_MODIFY value is the byte-address increment that is applied after each decrement of the DMAx_CURR_Y_COUNT register. However, the DMAx_Y_MODIFY value is not applied to the last item in the array on which the outer loop count (DMAx_CURR_Y_COUNT) also expires by decrementing from 1 to 0.

After the last transfer completes, DMAx_CURR_Y_COUNT = 1, DMAx_CURR_X_COUNT = 0, and DMAx_CURR_ADDR is equal to the last item's address plus DMAx_X_MODIFY.

If the DMA channel is programmed to refresh automatically (auto-buffer mode), then these registers will be loaded from DMAx_X_COUNT, DMAx_Y_COUNT, and DMAx_START_ADDR upon the first data transfer.

The DI_SEL configuration bit enables DMA interrupt requests every time the inner loop rolls over. If DI_SEL is cleared, but DI_EN is still set, only one interrupt is generated after the outer loop completes.

Examples of Two-Dimensional DMA

Example 1: Retrieve a 16 × 8 block of bytes from a video frame buffer of size (N × M) pixels:

```
DMAx_X_MODIFY = 1
DMAx_X_COUNT = 16
DMAx_Y_MODIFY = N-15 (offset from the end of one row to the start of another)
DMAx_Y_COUNT = 8
```

This produces the following address offsets from the start address:

```
0,1,2,...15,
N,N + 1, ... N + 15,
2N, 2N + 1,... 2N + 15, ...
7N, 7N + 1,... 7N + 15,
```

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) × (N × M image size):

```
DMAx_X_MODIFY = (N * M)
DMAx_X_COUNT = 3
DMAx_Y_MODIFY = 1 - 2(N * M) (negative)
DMAx_Y_COUNT = (N * M)
```

This produces the following address offsets from the start address:

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

Descriptor-based DMA Operation

In descriptor-based DMA operation, software does not set up DMA sequences by writing directly into DMA controller registers. Rather, software keeps DMA configurations, called descriptors, in memory. On demand, the DMA controller loads the descriptor from memory and overwrites the affected DMA registers by its own control. Descriptors can be fetched from L1 memory using the DCB bus or from external memory using the DEB bus.

A descriptor describes what kind of operation should be performed next by the DMA channel. This includes the DMA configuration word as well as data source/destination address, transfer count, and address modify values. A DMA sequence controlled by one descriptor is called a work unit.

Optionally, an interrupt can be requested at the end of any work unit by setting the DI_EN bit in the configuration word of the respective descriptor.

A DMA channel is started in descriptor-based mode by first writing the 32-bit address of the first descriptor into the DMAx_NEXT_DESC_PTR register (or the DMAx_CURR_DESC_PTR in case of descriptor array mode) and then performing a write to the DMAx_CONFIG register that sets the FLOW field to either 0x4, 0x6, or 0x7 and enables the DMAEN bit. This causes the DMA controller to immediately fetch the descriptor from the address pointed to by the DMAx_NEXT_DESC_PTR register. The fetch overwrites the DMAx_CONFIG register again. If the DMAEN bit is still set, the channel starts DMA processing.

The DFETCH bit in the DMAx_IRQ_STATUS register tells whether a descriptor fetch is ongoing on the respective DMA channel. The DMAx_CURR_DESC_PTR points to the descriptor value that is to be fetched next.

Descriptor List Mode

Descriptor list mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to either 0x6 (small descriptor mode) or 0x7 (large descriptor mode). In either of these modes multiple descriptors form a chained list. Every descriptor contains a pointer to the next descriptor. When the descriptor is fetched, this pointer value is loaded into the DMAx_NEXT_DESC_PTR register of the DMA channel. In large descriptor mode this pointer is 32 bits wide. Therefore, the next descriptor may reside in any address space accessible through the DCB and DEB buses. In small descriptor mode this pointer is just 16 bits wide. For this reason, the next descriptor must reside in the same 64K byte address space as the first one because the upper 16 bits of the DMAx_NEXT_DESC_PTR register are not updated.

Descriptor list modes are started by writing first to the `DMAx_NEXT_DESC_PTR` register and then to the `DMAx_CONFIG` register.

Descriptor Array Mode

Descriptor array mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to 0x4. In this mode, the descriptors do not contain further descriptor pointers. The initial DMAx_CURR_DESC_PTR value is written by software. It points to an array of descriptors. The individual descriptors are assumed to reside next to each other and, therefore, their addresses are known.

Variable Descriptor Size

In any descriptor-based mode the `NDSIZE` field in the configuration word specifies how many 16-bit words of the next descriptor need to be loaded on the next fetch. In descriptor-based operation, `NDSIZE` must be non-zero. The descriptor size can be any value from one entry (the lower 16 bits of `DMAx_START_ADDR` only) to nine entries (all the DMA parameters). Table 7-1 illustrates how a descriptor must be structured in

memory. The values have the same order as the corresponding MMR addresses.

If, for example, a descriptor is fetched in array mode with NDSIZE = 0x5, the DMA controller fetches the 32-bit start address, the DMA configuration word, and the XCNT and XMOD values. However, it does not load YCNT and YMOD. This might be the case if the DMA operates in one-dimensional mode or if the DMA is in two-dimensional mode, but the YCNT and YMOD values do not need to change.

All the other registers not loaded from the descriptor retain their prior values, although the DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, and DMAx_CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.

Table 7-1 shows the offsets for descriptor elements in the three modes described above. Note the names in the table describe the descriptor elements in memory, not the actual MMRs into which they are eventually loaded. For more information regarding descriptor element acronyms, see Table 7-4 on page 7-64.

Table 7-1. Parameter Registers and Descriptor Offsets

Descriptor Offset	Descriptor Array Mode	Small Descriptor List Mode	Large Descriptor List Mode
0x0	SAL	NDPL	NDPL
0x2	SAH	SAL	NDPH
0x4	DMACFG	SAH	SAL
0x6	XCNT	DMACFG	SAH
0x8	XMOD	XCNT	DMACFG
0xA	YCNT	XMOD	XCNT
0xC	YMOD	YCNT	XMOD
0xE		YMOD	YCNT
0x10			YMOD

Note that every descriptor fetch consumes bandwidth from either the DCB bus or the DEB bus and the external memory interface, so it is best to keep the size of descriptors as small as possible.

Mixing Flow Modes

The FLOW mode of a DMA is not a global setting. If the DMA configuration word is reloaded with a descriptor fetch, the FLOW and NDSIZE bit fields can also be altered. A small descriptor might be used to loop back to the first descriptor if a descriptor array is used in an endless manner. If the descriptor chain is not endless and the DMA is required to stop after a certain descriptor has been processed, the last descriptor is typically processed in stop mode. That is, its FLOW and NDSIZE fields are 0, but its DMAEN bit is still set.

Functional Description

The following sections provide a functional description of DMA.

DMA Operation Flow

Figure 7-1 and Figure 7-2 describe the DMA flow.

DMA Startup

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

Before initiating DMA for the first time on a given channel, all parameter registers must be initialized. Be sure to initialize the upper 16 bits of the DMAx_NEXT_DESC_PTR (or DMAx_CURR_DESC_PTR register in FLOW = 4 mode) and DMAx_START_ADDR registers, because they might not otherwise be accessed, depending upon the flow mode. Also note that the

DMAx_X_MODIFY and DMAx_Y_MODIFY registers are not preset to a default value at reset.

The user may wish to write other DMA registers that might be static during DMA activity (for example, DMAx_X_MODIFY, DMAx_Y_MODIFY). The contents of NDSIZE and FLOW in DMAx_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMAx_CONFIG with DMAEN = 1.

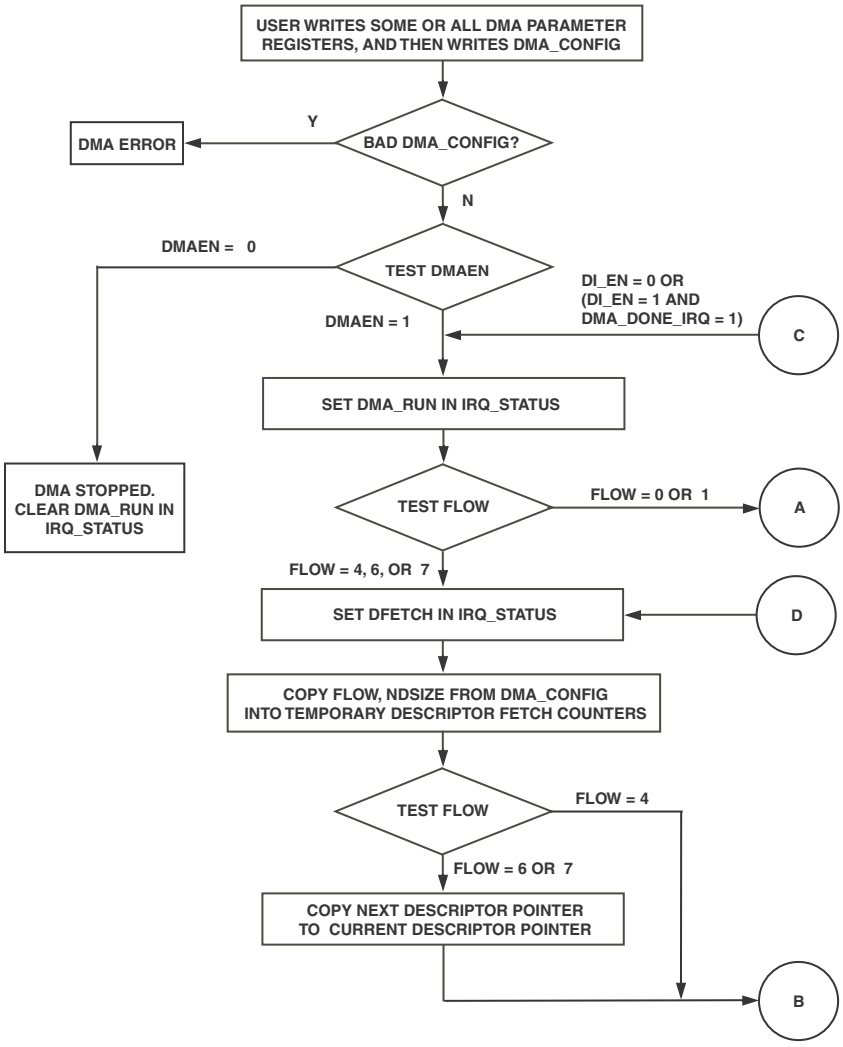


Figure 7-1. DMA Flow, From DMA Controller's Point of View (1 of 2)

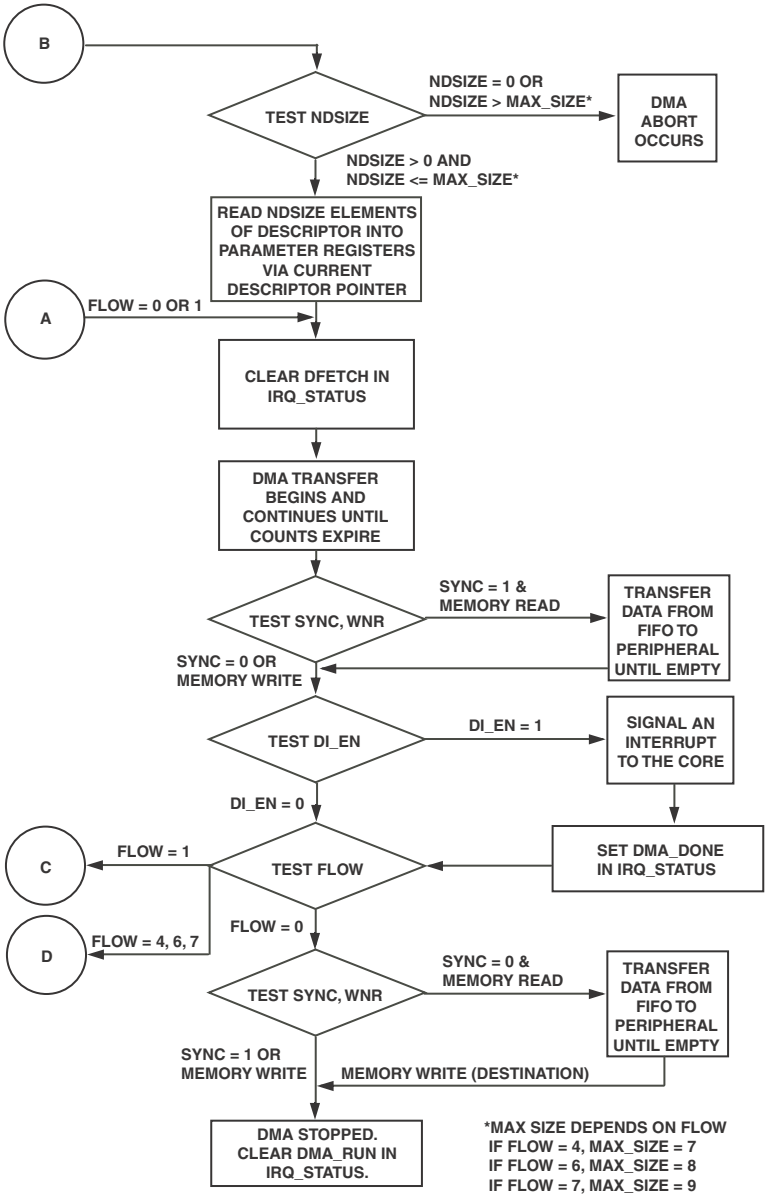


Figure 7-2. DMA Flow, From DMA Controller's Point of View (2 of 2)

When DMAx_CONFIG is written directly by software, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMAEN = 1, the DMACFG descriptor element that was read into DMAx_CONFIG assumes control. Before this point, the direct write to DMAx_CONFIG had control. In other words, the WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMACFG value in the descriptor read from memory, while these field values initially written to the DMAx_CONFIG register are ignored.

As Figure 7-1 and Figure 7-2 show, at startup the FLOW and NDSIZE bits in DMAx_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values.

If the FLOW value specifies small or large descriptor list modes, the DMAx_NEXT_DESC_PTR is copied into DMAx_CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by DMAx_CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into DMAx_NEXT_DESC_PTR, but the fetch of the current descriptor continues using DMAx_CURR_DESC_PTR. After completion of the descriptor fetch, DMAx_CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

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

If DMACFG is not part of the descriptor, the previous DMAx_CONFIG settings (as written by MMR access at startup) control the work unit operation. If

DMACFG is part of the descriptor, then the DMAx_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor's DMACFG and by the parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the DMAx_IRQ_STATUS register indicate the state of the DMA channel. After a write to DMAx_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, DMA data transfer operation begins. The DMA channel immediately attempts to fill its FIFO, subject to channel priority—a memory write (RX) DMA channel begins accepting data from its peripheral, and a memory read (TX) DMA channel begins memory reads, provided the channel wins the grant for bus access.

When the DMA channel performs its first data memory access, its address and count computations take their input operands from the start registers (DMAx_START_ADDR, DMAx_X_COUNT, DMAx_Y_COUNT), and write results back to the current registers (DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, DMAx_CURR_Y_COUNT). Note also that the current registers are not valid until the first memory access is performed, which may be some time after the channel is started by the write to the DMA_CONFIG register. The current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- DMAx_START_ADDR is copied to DMAx_CURR_ADDR
- DMAx_X_COUNT is copied to DMAx_CURR_X_COUNT
- DMAx_Y_COUNT is copied to DMAx_CURR_Y_COUNT

Then DMA data transfer operation begins, as shown in Figure 7-2 on page 7-20.

DMA Refresh

On completion of a work unit:

- The DMA controller completes the transfer of all data between memory and the DMA unit.
- If SYNC = 1 and WNR = 0 (memory read), the DMA controller selects a synchronized transition and transfers all data to the peripheral before continuing.
- If enabled by DI_EN, the DMA controller signals an interrupt to the core and sets the DMA_DONE bit in the channel's DMAx_IRQ_STATUS register.
- If FLOW = 0 the DMA controller stops operation by clearing the DMA_RUN bit in DMAx_IRQ_STATUS register after all data in the channel's DMA FIFO has been transferred to the peripheral.
- During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in DMAx_IRQ_STATUS register to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

  If FLOW = 4 (descriptor array) the DMA controller loads a new descriptor from memory into the DMA registers using the contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMAx_CONFIG register prior to the beginning of the fetch.

  If FLOW = 6 (small descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 16-bit value for the lower 16 bits of DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor

elements. The high 16 bits of DMAx_NEXT_DESC_PTR will retain their former value. This supports a shorter, more efficient descriptor than the large descriptor list model, which is suitable whenever the application can place the channel's descriptors in the same 64K byte range of memory.

If FLOW = 7 (large descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 32-bit value for the full DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory or external memory.

- If it is necessary to link from a descriptor chain whose descriptors are in one 64K byte area to another chain whose descriptors are outside that area, only the descriptor containing the link to the new 64K byte range needs to use FLOW = 7. All descriptors that reference the same 64K byte area may use FLOW = 6.
- If FLOW = 4, 6, or 7 (descriptor array, small descriptor list, or large descriptor list, respectively), the DMA controller clears the DFETCH bit in the DMAx_IRQ_STATUS register.

- If FLOW = any value but 0 (Stop), the DMA controller begins the next work unit for that channel, which must contend with other channels for priority on the memory buses. On the first memory transfer of the new work unit, the DMA controller updates the current registers from the start registers:

  DMAx_CURR_ADDR loaded from DMAx_START_ADDR
  DMAx_CURR_X_COUNT loaded from DMAx_X_COUNT
  DMAx_CURR_Y_COUNT loaded from DMAx_Y_COUNT

  The DFETCH bit in the DMAx_IRQ_STATUS register is then cleared, after which the DMA transfer begins again, as shown in Figure 7-2 on page 7-20.

Work Unit Transitions

Transitions from one work unit to the next are controlled by the SYNC bit in the DMAx_CONFIG register of the work units. In general, continuous transitions have lower latency at the cost of restrictions on changes of data format or addressed memory space in the two work units. These latency gains and data restrictions arise from the way the DMA FIFO pipeline is handled while the next descriptor is fetched. In continuous transitions (SYNC = 0), the DMA FIFO pipeline continues to transfer data to and from the peripheral or destination memory during the descriptor fetch and/or when the DMA channel is paused between descriptor chains.

Synchronized transitions (SYNC = 1), on the other hand, provide better real-time synchronization of interrupts with peripheral state and greater flexibility in the data formats and memory spaces of the two work units, at the cost of higher latency in the transition. In synchronized transitions, the DMA FIFO pipeline is drained to the destination or flushed (RX data discarded) between work units.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be

0. In transmit (memory read) channels, the SYNC bit of the last descriptor prior to the transition controls the transition behavior. In contrast, in receive channels, the SYNC bit of the first descriptor of the next descriptor chain controls the transition.

DMA Transmit and MDMA Source

In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work units.

If SYNC = 0, a continuous transition is selected. In a continuous transition, just after the last data item is read from memory, the following operations start in parallel:

- The interrupt (if any) is signalled.
- The DMA_DONE bit in the DMAx_IRQ_STATUS register is set.
- The next descriptor begins to be fetched.
- The final data items are delivered from the DMA FIFO to the destination memory or peripheral.

This allows the DMA to provide data from the FIFO to the peripheral "continuously" during the descriptor fetch latency period.

When SYNC = 0, the final interrupt (if enabled) occurs when the last data is read from memory. This interrupt is at the earliest time that the output memory buffer may safely be modified without affecting the previous data transmission. Up to four data items may still be in the DMA FIFO, however, and not yet at the peripheral, so the DMA interrupt should not be used as the sole means of synchronizing the shutdown or reconfiguration of the peripheral following a transmission.

If SYNC = 0 (continuous transition) on a transmit (memory read) descriptor, the next descriptor must have the same data word size, read/write direction, and source memory (internal vs. external) as the current descriptor.

SYNC = 0 selects continuous transition on a work unit in FLOW = 0 mode with interrupt enabled. The interrupt service routine may begin execution while the final data is still draining from the FIFO to the peripheral. This is indicated by the DMA_RUN bit in the DMAx_IRQ_STATUS register; if it is 1, the FIFO is not empty yet. Do not start a new work unit with different word size or direction while DMA_RUN = 1. Further, if the channel is disabled (by writing DMAEN = 0), the data in the FIFO is lost.

SYNC = 1 selects a synchronized transition in which the DMA FIFO is first drained to the destination memory or peripheral before any interrupt is signalled and before any subsequent descriptor or data is fetched. This incurs greater latency, but provides direct synchronization between the DMA interrupt and the state of the data at the peripheral.

For example, if SYNC = 1 and DI_EN = 1 on the last descriptor in a work unit, the interrupt occurs when the final data has been transferred to the peripheral, allowing the service routine to properly switch to non-DMA transmit operation. When the interrupt service routine is invoked, the DMA_DONE bit is set and the DMA_RUN bit is cleared.

A synchronized transition also allows greater flexibility in the format of the DMA descriptor chain. If SYNC = 1, the next descriptor may have any word size or read/write direction supported by the peripheral and may come from either memory space (internal or external). This can be useful in managing MDMA work unit queues, since it is no longer necessary to interrupt the queue between dissimilar work units.

DMA Receive

In DMA receive (memory write) channels, the SYNC bit controls the handling of the DMA FIFO between descriptor chains (not individual

descriptors), when the DMA channel is paused. The DMA channel pauses after descriptors with FLOW = 0 mode, and may be restarted (for example, after an interrupt) by writing the channel's DMAx_CONFIG register with DMAEN = 1.

If the SYNC bit is 0 in the new work unit's DMAx_CONFIG value, a continuous transition is selected. In this mode, any data items received into the DMA FIFO while the channel was paused are retained, and they are the first items written to memory in the new work unit. This mode of operation provides lower latency at work unit transitions and ensures that no data items are dropped during a DMA pause, at the cost of certain restrictions on the DMA descriptors.

If the SYNC bit is 0 on the first descriptor of a descriptor chain after a DMA pause, the DMA word size of the new chain must not change from the word size of the previous descriptor chain active before the pause, unless the DMA channel is reset between chains by writing the DMAEN bit to 0 and then to 1 again.

If the SYNC bit is 1 in the new work unit's DMAx_CONFIG value, a synchronized transition is selected. In this mode, only the data received from the peripheral by the DMA channel after the write to the DMAx_CONFIG register are delivered to memory. Any prior data items transferred from the peripheral to the DMA FIFO before this register write are discarded. This provides direct synchronization between the data stream received from the peripheral and the timing of the channel restart (when the DMAx_CONFIG register is written).

For receive DMAs, the SYNC bit has no effect in transitions between work units in the same descriptor chain (that is, when the previous descriptor's FLOW mode was not 0, so that DMA channel did not pause.)

If a descriptor chain begins with a SYNC bit of 1, there is no restriction on DMA word size of the new chain in comparison to the previous chain.

The DMA word size must not change between one descriptor and the next in any DMA receive (memory write) channel within a single descriptor chain, regardless of the SYNC bit setting. In other words, if a descriptor has WNR = 1 and FLOW = 4, 6, or 7, then the next descriptor must have the same word size. For any DMA receive (memory write) channel, there is no restriction on changes of memory space (internal vs. external) between descriptors or descriptor chains. DMA transmit (memory read) channels may have such restrictions (see "DMA Transmit and MDMA Source" on page 7-26).

Stopping DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMAEN bit is 0. A write of 0 to the entire register will always terminate DMA gracefully (without DMA abort).

If a channel has been stopped abruptly by writing DMAx_CONFIG to 0 (or any value with DMAEN = 0), the user must ensure that any memory read or write accesses in the pipelines have completed before enabling the channel again. If the channel is enabled again before an "orphan" access from a previous work unit completes, the state of the DMA interrupt and FIFO is unspecified. This can generally be handled by ensuring that the core allocates several consecutive idle cycles in its usage of the relevant memory space to allow up to three pending DMA accesses to issue, plus allowing enough memory access time for the accesses themselves to complete.

DMA Errors (Aborts)

The DMA controller flags conditions that cause the DMA process to end abnormally (abort). This functionality is provided as a tool for system development and debug to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

There is only one DMA_ERROR interrupt for the whole DMA controller, which is asserted whenever any of the channels has detected an error condition.

The DMA_ERROR interrupt handler must:

- Read each channel's DMAx_IRQ_STATUS register to look for a channel with the DMA_ERR bit set (bit 1).
- Clear the problem with that channel (for example, fix register values).
- Clear the DMA_ERR bit (write DMAx_IRQ_STATUS with bit 1 set).

The following error conditions are detected by the DMA hardware and result in a DMA abort interrupt.

- The configuration register contains invalid values:
  - Incorrect WDSIZE value (WDSIZE = b#11)
  - Bit 15 not set to 0
  - Incorrect FLOW value (FLOW = 2, 3, or 5)
  - NDSIZE value does not agree with FLOW. See Table 7-2 on page 7-32.

- A disallowed register write occurred while the channel was running. Only the DMAx_CONFIG and DMAx_IRQ_STATUS registers can be written when DMA_RUN = 1.
- An address alignment error occurred during any memory access. For example, when DMAx_CONFIG register WDSIZE = 1 (16-bit) but the least significant bit (LSB) of the address is not equal to b#0, or when WDSIZE = 2 (32-bit) but the two LSBs of the address are not equal to b#00.
- A memory space transition was attempted (internal-to-external or vice versa). For example, the value in the DMAx_CURR_ADDR register or DMAx_CURR_DESC_PTR register crossed a memory boundary.
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signaled by the external memory interface).

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signaled for these situations:

- DMAx_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMAx_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMAx_CONFIG word size (WDSIZE) is not supported by the mapped peripheral. See Table 7-2.
- DMAx_CONFIG word size in source and destination of the MDMA stream are not equal.
- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2-D DMA, X_COUNT = 1

Table 7-2. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	0 < NDSIZE ≤ 7	Descriptor array, no descriptor pointer fetched
6	0 < NDSIZE ≤ 8	Descriptor list, small descriptor pointer fetched
7	0 < NDSIZE ≤ 9	Descriptor list, large descriptor pointer fetched

DMA Control Commands

Advanced peripherals, such as an Ethernet MAC module, are capable of managing some of their own DMA operations, thus dramatically improving real-time performance and relieving control and interrupt demands on the Blackfin processor core. These peripherals may communicate to the DMA controller using DMA control commands, which are transmitted from the peripheral to the associated DMA channel over internal DMA request buses. Refer to "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine if DMA control commands are applicable to a particular product.

The request buses consist of three wires per DMA-management-capable peripheral. The DMA control commands extend the set of operations available to the peripheral beyond the simple "request data" command used by peripherals in general.

While these DMA control commands are not visible to or controllable by the user, their use by a peripheral has implications for the structure of the DMA transfers which that peripheral can support. It is important that application software be written to comply with certain restrictions regarding work units and descriptor chains (described later in this section) so that the peripheral operates properly whenever it issues DMA control commands.

MDMA channels do not service peripherals and therefore do not support DMA control commands. The DMA control commands are shown in Table 7-3.

Table 7-3. DMA Control Commands

Code	Name	Description
000	NOP	No operation
001	Restart	Restarts the current work unit from the beginning
010	Finish	Finishes the current work unit and starts the next
011	-	Reserved
100	Req Data	Typical DMA data request
101	Req Data Urgent	Urgent DMA data request
110	-	Reserved
111	-	Reserved

Additional information for the control commands includes:

- **Restart**

  The Restart command causes the current work unit to interrupt processing and start over, using the addresses and counts from DMAx_START_ADDR, DMAx_X_COUNT, and DMAx_Y_COUNT. No interrupt is signalled.

  If a channel programmed for transmit (memory read) receives a Restart command, the channel momentarily pauses while any pending memory reads initiated prior to the Restart command are completed.

  During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel resets its counters and FIFO and starts prefetch reads from memory. DMA data requests from the

peripheral are granted as soon as new prefetched data is available in the DMA FIFO. The peripheral can thus use the Restart command to re-attempt a failed transmission of a work unit.

If a channel programmed for receive (memory write) receives a Restart command, the channel stops writing to memory, discards any data held in its DMA FIFO, and resets its counters and FIFO. As soon as this initialization is complete, the channel again grants DMA write requests from the peripheral. The peripheral can thus use the Restart command to abort transfer of received data into a work unit and re-use the memory buffer for a later data transfer.

- **Finish**

  The Finish command causes the current work unit to terminate and move on to the next work unit. An interrupt is signalled as usual, if selected by the DI_EN bit. The peripheral can thus use the Finish command to partition the DMA stream into work units on its own, perhaps as a result of parsing the data currently passing though its supported communication channel, without direct real-time control by the processor.

  If a channel programmed for transmit (memory read) receives a Finish command, the channel momentarily pauses while any pending memory reads initiated prior to the Finish command are completed. During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel signals an interrupt (if enabled), and begins fetching the next descriptor (if any). DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO.

  If a channel programmed for receive (memory write) receives a Finish command, the channel stops granting new DMA requests while it drains its FIFO. Any DMA data received by the DMA controller prior to the Finish command is written to memory. When the

FIFO reaches an empty state, the channel signals an interrupt (if enabled) and begins fetching the next descriptor (if any). Once the next descriptor has been fetched, the channel initializes its FIFO and then resumes granting DMA requests from the peripheral.

- **Request Data**

  The `Request Data` command is identical to the DMA request operation of peripherals that are not DMA-management-capable.

- **Request Data Urgent**

  The `Request Data Urgent` command behaves identically to the DMA `Request` command, except that the DMA channel performs its memory accesses with urgent priority while it is asserted. This includes both data and descriptor-fetch memory accesses. A DMA-management-capable peripheral might use this command if an internal FIFO is approaching a critical condition.

Restrictions

The proper operation of the 4-location DMA channel FIFO leads to certain restrictions in the sequence of DMA control commands.

Transmit Restart or Finish

No `Restart` or `Finish` command may be issued by a peripheral to a channel configured for memory read unless the peripheral has already performed at least one DMA transfer in the current work unit and the current work unit has more than four items remaining in `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT` (thus not yet read from memory). Otherwise, the current work unit may already have completed memory operations and can no longer be restarted or finished properly.

If the `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT` value of the current work unit is sufficiently large that it is always at least five more than the maximum data count prior to any `Restart` or `Finish` command, the above

restriction is satisfied. This implies that any work unit which might be managed by Restart or Finish commands must have DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing at least five data items.

Particularly if the DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers are programmed to 0 (representing 65,536 transfers, the maximum value) the channel will operate properly for 1-D work units up to 65,531 data items or 2-D work units up to 4,294,967,291 data items.

Receive Restart or Finish

No Restart or Finish command may be issued by a peripheral to a channel configured for memory write unless either the peripheral has already performed at least five DMA transfers in the current work unit or the previous work unit was terminated by a Finish command and the peripheral has performed at least one DMA transfer in the current work unit. If five data transfers have been performed, then at least one data item has been written to memory in the current work unit, which implies that the current work unit's descriptor fetch completed before the data grant of the fifth item. Otherwise, if less than five data items have been transferred, it is possible that all of them are still in the DMA FIFO and the previous work unit is still in the process of completion and transition between work units.

Similarly, if a Finish command ended the previous work unit and at least one subsequent DMA data transfer has occurred, then the fact that the DMA channel issued the grant guarantees that the previous work unit has already completed the process of draining its data to memory and transitioning to the new work unit.

If a peripheral terminates all work units with the Finish opcode (effectively assuming responsibility for all work unit boundaries for the DMA channel), then the peripheral need only ensure that it performs a single transfer in each work unit before any restart or finish. This requires, however, that the user programs the descriptors for all work units managed by the channel with DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values repre-

senting more data items than the maximum work unit size that the peripheral will encounter. For example, DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values of 0 allow the channel to operate properly on 1-D work units up to 65,535 data items and 2-D work units up to 4,294,967,295 data items.

Handshaked Memory DMA Operation

Handshaked memory DMA operation is not available for all products. Refer to the "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine whether this feature applies to this product.

Each DMARx input has its own set of control and status registers. Handshake operation for MDMA0 is enabled by the HMDMAEN bit in the HMDMA0_CONTROL register. Similarly, the HMDMAEN bit in the HMDMA1_CONTROL register enables handshake mode for MDMA1.

It is important to understand that the handshake hardware works completely independently from the descriptor and autobuffer capabilities of the MDMA, allowing most flexible combinations of logical data organization vs. data portioning as required by FIFO depths, for example. If, however, the connected device requires certain behavior of the address lines, these must be controlled by traditional DMA setup.

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

The HMDMAx_BCINIT registers control how many data transfers are performed upon every DMA request. If set to one, the peripheral can time every individual data transfer. If greater than one, the peripheral must have sufficient buffer size to provide or consume the number of words programmed. Once the transfer has been requested, no further handshake can hold off the DMA from transferring the entire block, except by stalling the EBIU accesses by the ARDY signal. Nevertheless, the peripheral

may request a block transfer before the entire buffer is available by simply taking the minimum transfer time based on wait-state settings into consideration.

The block count defines how many data transfers are performed by the MDMA engine. A single DMA transfer can cause two read or write operations on the EBIU port if the transfer word size is set to 32-bit in the MDMA_yy_CONFIG register (WDSIZE = b#10).

Since the block count registers are 16 bits wide, blocks can group up to 65,535 transfers.

Once a block transfer has been started, the HMDMAx_BCOUNT registers return the remaining number of transfers to complete the current block. When the complete block has been processed, the HMDMAx_BCOUNT register returns zero. Software can force a reload of the HMDMAx_BCOUNT from the HMDMAx_BCINIT register even during normal operation by setting the RBC bit in the HMDMAx_CONTROL register. Set RBC when the HMDMA module is already active, but only when the MDMA is not enabled.

Pipelining DMA Requests

The device mastering the DMA request lines is allowed to request additional transfers even before the former transfer has completed. As long as the device can provide or consume sufficient data it is permitted to pulse the DMARx inputs multiple times.

The HMDMAx_ECOUNT registers are incremented every time a significant edge is detected on the respective DMARx input, and they are decremented when the MDMA completes the block transfer. These read-only registers use a 16-bit twos-complement data representation: if they return zero, all requested block transfers have been performed. A positive value signals up to 32767 requests that haven't been served yet and indicates that the MDMA is currently processing. Negative values indicate the number of DMA requests that will be ignored by the engine. This feature restrains initial pulses on the DMARx inputs at startup.

The HMDMAx_ECINIT registers reload the HMDMAx_ECOUNT registers every time the handshake mode is enabled (when the HMDMAEN bit changes from 0 to 1). If the initial edge count value is 0, the handshake operation starts with a settled request budget. If positive, the engine starts immediately transferring the programmed number (up to 32767) of blocks once enabled, even without detecting any activity on the DMARx pins. If negative, the engine will disregard the programmed number (up to 32768) significant edges on the DMARx inputs before starting normal operation.

Figure 7-3 illustrates how an asynchronous FIFO could be connected. In such a scenario the REP bit should be cleared to let the DMARx request pin listen to falling edges. The Blackfin processor does not evaluate the full flag such FIFOs usually provide because asynchronous polling of that signal would reduce the system throughput drastically. Moreover, the processor first fills the FIFO by initializing the HMDMAx_ECINIT register to 1024, which equals the depth of the FIFO. Once enabled, the MDMA automatically transmits 1024 data words. Afterward it continues to transmit only if the FIFO is emptied by its read strobe again. Most likely, the HMDMAx_BCINIT register is programmed to 1 in this case.

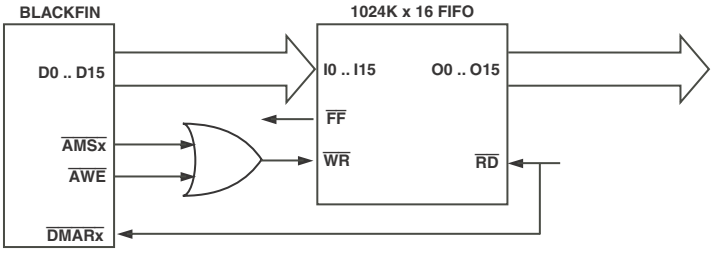


Figure 7-3. Transmit DMA Example Connection

In the receive example shown in Figure 7-4, the Blackfin processor again does not use the FIFO's internal control mechanism. Rather than testing the empty flag, the processor counts the number of data words available in the FIFO in its own HMDMAx_ECOUNT register. Theoretically, the MDMA could immediately process data as soon as it is written into the FIFO by

the write strobe, but the fast MDMA engine would read out the FIFO quickly and stall soon if the FIFO was not promptly filled with new data. Streaming applications can balance the FIFO so that the producer is never held off by a full FIFO and the consumer is never held by an empty FIFO. This is accomplished by filling the FIFO halfway and then letting both consumer and producer run at the same speed. In this case the HMDMAx_ECINIT register can be written with a negative value, which corresponds to half the FIFO depth. Then, the MDMA does not start consuming data as long as the FIFO is not half-filled.

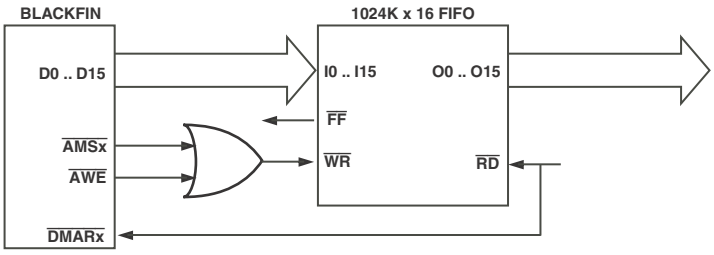


Figure 7-4. Receive DMA Example Connection

On internal system buses, memory DMA channels have lower priority than other DMAs. In busy systems, the memory DMAs may tend to starve. As this is not acceptable when transferring data through high-speed FIFOs, the handshake mode provides a high-water functionality to increase the MDMA's priority. With the UTE bit in the HMDMAx_CONTROL register set, the MDMA gets higher priority as soon as a (positive) value in the HMDMAx_ECOUNT register becomes higher than the threshold held by the HMDMAx_ECURGENT register.

HMDMA Interrupts

In addition to the normal MDMA interrupt channels, the handshake hardware provides two new interrupt sources for each DMARx input. The HMDMAx_CONTROL registers provide interrupt enable and status bits. The

interrupt status bits require a write-1-to-clear operation to cancel the interrupt request.

The `block done` interrupt signals that a complete MDMA block, as defined by the `HMDMAx_BCINIT` register, has been transferred (when the `HMDMAx_BCOUNT` register decrements to zero). While the `BDIE` bit enables this interrupt, the `MBDI` bit can gate it until the edge count also becomes zero, meaning that all requested MDMA transfers have been completed.

The overflow interrupt is generated when the HMDMA_ECOUNT register overflows. Since it can count up to 32767, which is much more than most peripheral devices can support, the Blackfin processor has another threshold register called HMDMA_ECOVERFLOW. It resets to 0xFFFF and should be written with any positive value by the user before enabling the function by the OIE bit. Then, the overflow interrupt is issued when the value of the HMDMA_ECOUNT register exceeds the threshold in the HMDMA_ECOVERFLOW register.

DMA Performance

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

The Blackfin architecture features several mechanisms to customize system behavior for best performance. This includes DMA channel prioritization, traffic control, and priority treatment of bursted transfers. Nevertheless, the resulting performance of a DMA transfer often depends on application-level circumstances. For best performance consider the following system software architecture questions:

- What is the required DMA bandwidth?
- Which DMA transfers have real-time requirements and which do not?

- How heavily is the DMA controller competing with the core for on-chip and off-chip resources?
- How often do competing DMA channels require the bus systems to alter direction?
- How often do competing DMA or core accesses cause the SDRAM to open different pages?
- Is there a way to distribute DMA requests nicely over time?

A key feature of the DMA architecture is the separation of the activity on the DMA access bus (DAB) used by the peripherals from the activity on the buses between the DMA and memory. For DMA to/from on-chip memory the DMA core bus (DCB) is used, and the DMA external bus (DEB) is used for DMA transfers with off-chip memory. Chapter 3, "Chip Bus Hierarchy" explains the bus architecture.

Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

DMA Throughput

Each peripheral DMA channel has a maximum transfer rate of one 16-bit word per two system clocks in either direction. Like the DAB and DEB buses, the DMA controller resides in the SCLK domain. The controller synchronizes accesses to and from the DCB bus, which runs at the CCLK rate.

Each memory DMA channel has a maximum transfer rate of one 16-bit word per system clock (SCLK) cycle.

When the traffic on all DMA channels is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock.

Some considerations which limit the actual performance include:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example when both the core and the DMA access the same L1 bank, when SDRAM pages need to be opened/closed, or when cache lines are filled.
- Direction changes from `RX` to `TX` on the DAB bus impose a one `SCLK` cycle delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to DMA registers other than `DMAx_CONFIG`, `DMAx_IRQ_STATUS`, or `DMAx_PERIPHERAL_MAP` stall all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the control/status registers do not cause stalls or wait states.
- Reads from DMA registers other than control/status registers use one PAB bus wait state, delaying the core for several core clocks.

- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMAEN changes from 0 to 1 or when the SYNC bit is set in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor's traffic control features, described in the next section.

The MDMA channels are clocked by SCLK. If the source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. For example DMA typically runs at 2/3 of the system clock rate when the core-to-system clock ratio is 2:1. At higher clock ratios, full bandwidth is maintained.

If the source and destination are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to three plus the memory latency at the

DMA in SCLKs (which is typically seven for internal transfers and six for external transfers).

Memory DMA Timing Details

When the destination DMAx_CONFIG register is written, MDMA operation starts after a latency of three SCLK cycles.

If either MDMA channel has been selected to use descriptors, the descriptors are fetched from memory. The destination channel descriptors are fetched first. Then the source MDMA channel begins fetching data from the source buffer, after a latency of four SCLK cycles after the last descriptor word is returned from memory. Due to memory pipelining, this is typically eight SCLK cycles after the fetch of the last descriptor word. The resulting data is deposited in the MDMA channel's 8-location FIFO. After a latency of two SCLK cycles, the destination MDMA channel begins writing data to the destination memory buffer.

Static Channel Prioritization

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the PMAP field in the DMAx_PERIPHERAL_MAP registers. The memory DMA streams are always lower static priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers.

Temporary DMA Urgency

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals

is not too large a fraction of the total, then all peripherals' requests should be granted as required.

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

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

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

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

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

DMA requests from an MDMA channel become urgent when handshaked operation is enabled and the DMARx edge count exceeds the value stored in the HMDMAx_ECURGENT register. If handshaked operation is disabled, software can control urgency of requests directly by altering the DRQ bit field in the HMDMAx_CONTROL register.

When one or more DMA channels express an urgent memory request, two events occur:

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external). All prior incomplete memory transfers ahead of it in that memory system are also marked for expedited processing. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

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

Memory DMA Priority and Scheduling

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

By default, when more than one MDMA stream is enabled and ready, only the highest priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, the `MDMA_ROUND_ROBIN_PERIOD` may be programmed to select each stream in turn for a fixed number of transfers.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the `MDMA_ROUND_ROBIN_PERIOD` field in the `DMA_TC_PER` register (see "Static Channel Prioritization" on page 7-45).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA stream 0 takes precedence over MDMA stream 1 whenever stream 0 is ready to perform transfers. Since an MDMA stream is typically capable of transferring data on every available cycle, this could cause MDMA stream 1 traffic to be delayed for an indefinite time until any and all MDMA stream 0 operations are completed. This scheme could be appropriate in systems where low duration but latency-sensitive data buffers need to be moved immediately, interrupting long duration, low priority background transfers.

If the MDMA_ROUND_ROBIN_PERIOD field is set to some nonzero value in the range $1 \le P \le 31$, then a round-robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to P data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for external-to-internal moves, and each would be allocated approximately equal data bandwidth.

In round-robin operation, the MDMA stream selection at any time is either “free” or “locked.” Initially, the selection is free. On any free cycle available to MDMA (when no peripheral DMA accesses take precedence), if either or both MDMA streams request access, the higher precedence stream will be granted (stream 0 in case of conflict), and that stream’s selection is then “locked.” The MDMA_ROUND_ROBIN_COUNT counter field in the DMA_TC_CNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of one, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not

ready to perform a transfer, then no transfer is performed, and the stream selection unlocks and becomes free again on the next cycle.

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

Traffic Control

In the Blackfin DMA architecture, there are two completely separate but simultaneous prioritization processes—the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (`SCLK`). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see “Temporary DMA Urgency” on page 7-45.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same direction transfers together. The DMA block provides a traffic control

mechanism controlled by the DMA_TC_PER and DMA_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMA_TC_CNT register. See "Memory DMA Priority and Scheduling" on page 7-47.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out or traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower priority channel 5 is going with traffic and higher priority channel 3 is going against traffic, then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If, on the next cycle, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both against traffic, then their effective priorities would become 19 and 22, respectively. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required for the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the DMA_TC_PER register to 0x0000.

Programming Model

Several synchronization and control methods are available for use in development of software tasks which manage peripheral DMA and memory DMA (see also "Memory DMA" on page 7-6). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each peripheral DMA and memory DMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the `DMAx_IRQ_STATUS` register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of `DMAx_IRQ_STATUS`, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note, since every interrupt channel

has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Due to DMA FIFOs and DMA/memory pipelining, polling of the DMAx_CURR_ADDR, DMAx_CURR_DESC_PTR, or DMAx_CURR_X_COUNT/DMAx_CURR_Y_COUNT registers is not recommended for precisely synchronizing DMA with data processing. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the SDRAM to perform a page open operation which takes many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but will be stalled behind the slow operation of channel A. Software monitoring of channel B, based on examination of the DMAx_CURR_ADDR register contents, would not safely conclude whether the memory location pointed to by channel B's DMAx_CURR_ADDR register has or has not been written.

If allowances are made for the lengths of the DMA/memory pipeline, polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software. The depth of the DMA FIFO is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) for a peripheral DMA channel, and eight locations (four 32-bit data elements) for an MDMA FIFO. The DMA will not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and external bus interface unit (EBIU) pipelines is approximately three data elements, when measured from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO

length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. This value is a maximum because the DMA/memory pipeline may include traffic from other DMA channels.

For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its DMAx_CURR_X_COUNT register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. Since the total pipeline length is no greater than the sum of four (for the peripheral DMA FIFO) plus six (for the DMA/memory pipeline) or ten data elements, it is safe to conclude that the DMA transfer of the first 30 (40-10) data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's DMAx_IRQ_STATUS register to confirm completion of DMA, rather than polling current address/pointer/count registers. When the DMA system issues an interrupt or changes a DMAx_IRQ_STATUS bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to processor code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO. For memory write DMA, the DMA unit will have received an acknowledgement from L1 memory, or the EBIU, that the data has been written.

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if a peripheral's DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel's control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the DMAx_CONFIG and the DMAx_NEXT_DESC_PTR registers. Alternatively, the

user may choose to write all the MMR registers directly from software, ending with the write to the DMAx_CONFIG register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the DI_EN bit in the DMAx_CONFIG register, and by the necessary setup of the system interrupt controller. If no interrupt is desired, the software can poll for completion by reading the DMAx_IRQ_STATUS register and testing the DMA_RUN bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering (FLOW = 1) may be an effective option. Here, DMA is transferred from or to a memory buffer with a circular addressing scheme, using either one- or two-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

- 1-D interrupt-driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
- 2-D interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set DI_SEL = 1 in DMAx_CONFIG) to be signaled at the completion of each DMA inner loop. In this way, a traditional double buffer or "ping-pong" scheme can be implemented.

For example, two 512-word sub-buffers inside a 1K-word buffer could be used to receive 16-bit peripheral data with these settings:

DMAx_START_ADDR = buffer base address

DMAx_CONFIG = 0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1, DMA2D = 1, WDSIZE = b#01, WNR = 1, DMAEN = 1)

DMAx_X_COUNT = 512

DMAx_X_MODIFY = 2 for 16-bit data

DMAx_Y_COUNT = 2 for two sub-buffers

DMAx_Y_MODIFY = 2 same as DMAx_X_MODIFY for contiguous sub-buffers

- 2-D polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2-D multibuffer synchronization scheme may be used. For example, assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2-D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

  DMAx_START_ADDR = buffer base address

  DMAx_CONFIG = 0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1, WDSIZE = b#10, WNR = 1, DMAEN = 1)

  DMAx_X_COUNT = 16

  DMAx_X_MODIFY = 4 for 32-bit data

  DMAx_Y_COUNT = 4 for four sub-buffers

  DMAx_Y_MODIFY = 4 same as DMAx_X_MODIFY for contiguous sub-buffers

The synchronization core might read DMAx_Y_COUNT to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of DMAx_Y_COUNT shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.

- 1-D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1-D autobuffer mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1-D or 2-D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list by selecting the appropriate FLOW setting in DMAx_CONFIG.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

It is important to remember the meaning of the various fields in the DMAx_CONFIG descriptor elements when building a list or array of DMA descriptors. In particular:

- The lower byte of DMAx_CONFIG specifies the DMA transfer to be performed by the *current* descriptor (for example 2-D interrupt-enable mode)
- The upper byte of DMAx_CONFIG specifies the format of the *next* descriptor in the chain. The NDSIZE and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the DMAx_CONFIG value written to the DMA channel's DMAx_CONFIG register should correspond to the current descriptor. At a minimum, the FLOW, NDSIZE, WNR, and DMAEN fields must all agree with the current descriptor. The WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMAx_CONFIG value in the descriptor read from memory. The field values initially written to the register are ignored. See "Initializing Descriptors in Memory" on page 7-97 in the "Programming Examples" section for information on how descriptors can be set up.

Descriptor Queue Management

A system designer might want to write a DMA manager facility which accepts DMA requests from other software. The DMA manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's NDPH and NDPL members point to the next descriptor, and the last descriptor points back to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (Ix, Lx, Mx, and Bx registers), so that it does not need to use comparison and conditional instructions to manage the

circular structure. In this case, the NDPH and NDPL members of each descriptor could even be written once at startup and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

- Interrupt on every descriptor
- Interrupt minimally – only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

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

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a DMAx_CONFIG value with a FLOW value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts are equal), the software increments its count and then starts the DMA unit by writing the new descriptor's DMAx_CONFIG value to the DMA channel's DMAx_CONFIG register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's DMAx_CONFIG value so that its upper half (FLOW and NDSIZE) now describes the newly queued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMAx_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's DMAx_IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was queued too late (the modification of the next-to-last descriptor's DMAx_CONFIG element occurred after that element was read into the DMA unit). In this case, the interrupt handler should write the DMAx_CONFIG value appropriate for the last descriptor to the DMA channel's DMAx_CONFIG register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts would need to be complex, performing several MMR accesses to ensure robust operation. In such a system environment, a minimal interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's DMAx_CONFIG word should have a FLOW value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code should queue later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the queue being processed by the DMA unit. In other words, all but the last active descriptors contain FLOW values ≥ 4 and have no interrupt enable set, while the last active descriptor contains a FLOW of 0 and an interrupt enable bit DI_EN set to 1. Also, all but the last waiting descriptors contain FLOW values ≥ 4 and no interrupt enables set, while the last waiting descriptor contains a FLOW of 0 and an interrupt enable bit set. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler with a single DMAx_CONFIG register write.

After queuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location containing the desired DMAx_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting).

Once processing by the DMA unit has started, it is critical that the software not directly modify the contents of the active descriptor queue unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

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

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

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped (the mailbox from the interrupt software is zero), the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMAx_CONFIG value to the channel's DMAx_CONFIG register). If the queue is not stopped, the non-interrupt software must not write to the DMAx_CONFIG register (which would cause a DMA error). Instead the descriptor should queue to the waiting queue, and update its mailbox directed to the interrupt handler.

Software-Triggered Descriptor Fetches

If a DMA has been stopped in FLOW = 0 mode, the DMA_RUN bit in the DMAx_IRQ_STATUS register remains set until the content of the internal DMA FIFOs has been completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMAx_CONFIG register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that has been stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the next descriptor pointer, regardless of whether operating in descriptor array mode or descriptor list mode.

The next descriptor pointer remains valid if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA

and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the configuration register immediately.

One possible procedure is:

1. Write to DMAx_NEXT_DESC_PTR
2. Write to DMAx_CONFIG with

   FLOW = 0x8

   NDSIZE ≥ 0xA

   DI_EN = 0

   DMAEN = 1
3. Automatically fetched DMACFG has

   FLOW = 0x0

   NDSIZE = 0x0

   SYNC = 1 (for transmitting DMAs only)

   DI_EN = 1

   DMAEN = 1
4. In the interrupt routine, repeat step 2. The DMAx_NEXT_DESC_PTR is updated by the descriptor fetch.

To avoid polling of the DMA_RUN bit, set the SYNC bit in case of memory read DMAs (DMA transmit or MDMA source).

If all DMACFG fields in a descriptor chain have the FLOW and NDSIZE fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

Especially when applied to MDMA channels, such scenarios play an important role. Usually, the timing of MDMAs cannot be controlled (see "Handshaked Memory DMA Operation" on page 7-37). By halting descriptor chains or rings this way, the whole DMA transaction can be broken into pieces that are individually triggered by software.

Source and destination channels of a MDMA may differ in descriptor structure. However, the total work count must match when the DMA stops. Whenever a MDMA is stopped, destination and source channels should both provide the same FLOW = 0 mode after exactly the same number of words. Accordingly, both channels need to be started afterward.

Software-triggered descriptor fetches are illustrated in Listing 7-7 on page 7-100. MDMA channels can be paused by software at any time by writing a 0 to the DRQ bit field in the HMDMAx_CONTROL register. This simply disables the self-generated DMA requests, whether or not the HMDMA is enabled.

DMA Registers

DMA registers fall into three categories:

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

DMA Channel Registers

A processor features up to twelve peripheral DMA channels and two channel pairs for memory DMA. All channels have an identical set of registers as summarized in Table 7-4.

Table 7-4 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, the register category, and where applicable, the corresponding name for the data element in a DMA descriptor.

Table 7-4. Generic Names of the DMA Memory-Mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter	NDPH (upper 16 bits), NDPL (lower 16 bits)
0x04	START_ADDR	Start address of current buffer	Parameter	SAH (upper 16 bits), SAL (lower 16 bits)
0x08	CONFIG	DMA Configuration register, including enable bit	Parameter	DMACFG
0x0C	Reserved	Reserved		
0x10	X_COUNT	Inner loop count	Parameter	XCNT
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter	XMOD
0x18	Y_COUNT	Outer loop count (2-D only)	Parameter	YCNT
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter	YMOD
0x20	CURR_DESC_PTR	Current descriptor pointer	Current	N/A
0x24	CURR_ADDR	Current DMA address	Current	N/A

Table 7-4. Generic Names of the DMA Memory-Mapped Registers (Cont'd)

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x28	IRQ_STATUS	Interrupt status register contains completion and DMA error interrupt status and channel state (run/fetch/paused)	Control/ Status	N/A
0x2C	PERIPHERAL_MAP	Peripheral to DMA channel mapping contains a 4-bit value specifying the peripheral associated with this DMA channel (read-only for MDMA channels)	Control/ Status	N/A
0x30	CURR_X_COUNT	Current count (1-D) or intra-row X count (2-D); counts down from X_COUNT	Current	N/A
0x34	Reserved	Reserved		
0x38	CURR_Y_COUNT	Current row count (2-D only); counts down from Y_COUNT	Current	N/A
0x3C	Reserved	Reserved		

Channel-specific register names are composed of a prefix and the generic MMR name shown in Table 7-4. For peripheral DMA channels the prefix "DMAx_" is used, where "x" stands for a channel number between 0 and 11. For memory DMA channels, the prefix is "MDMA_yy_", where "yy" stands for either "D0", "S0", "D1", or "S1" to indicate destination and source channel registers of MDMA0 and MDMA1. For example the peripheral DMA channel 6 configuration register is called DMA6_CONFIG. The register for the MDMA1 source channel is called MDMA_S1_CONFIG.

The generic MMR names shown in Table 7-4 are not actually mapped to resources in the processor.

For convenience, discussions in this chapter use generic (non-peripheral specific) DMA and memory DMA register names.

DMA channel registers fall into three categories.

- Parameter registers such as `DMAx_CONFIG` and `DMAx_X_COUNT` that can be loaded directly from descriptor elements as shown in Table 7-4
- Current registers such as `DMAx_CURR_ADDR` and `DMAx_CURR_X_COUNT`
- Control/status registers such as `DMAx_IRQ_STATUS` and `DMAx_PERIPHERAL_MAP`

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers.

- `DMAx_NEXT_DESC_PTR`
- `DMAx_START_ADDR`
- `DMAx_CURR_DESC_PTR`
- `DMAx_CURR_ADDR`

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses different naming conventions for physical registers and their corresponding elements in descriptors that reside in memory. Table 7-4 shows the relation.

DMA Peripheral Map Registers (DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP)

Each DMA channel's DMAx_PERIPHERAL_MAP register contains bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel

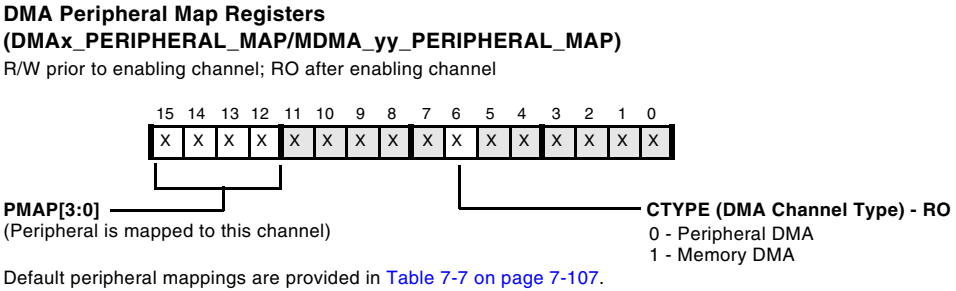


Figure 7-5. DMA Peripheral Map Registers

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7.
2. Write `DMA6_PERIPHERAL_MAP` with 0x7000 and `DMA7_PERIPHERAL_MAP` with 0x6000.
3. Enable DMA on channels 6 and/or 7.

DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)

The DMAx_CONFIG register, shown in Figure 7-6, is used to set up DMA parameters and operating modes. Writing the DMAx_CONFIG register while DMA is already running will cause a DMA error unless writing with the DMAEN bit set to 0.

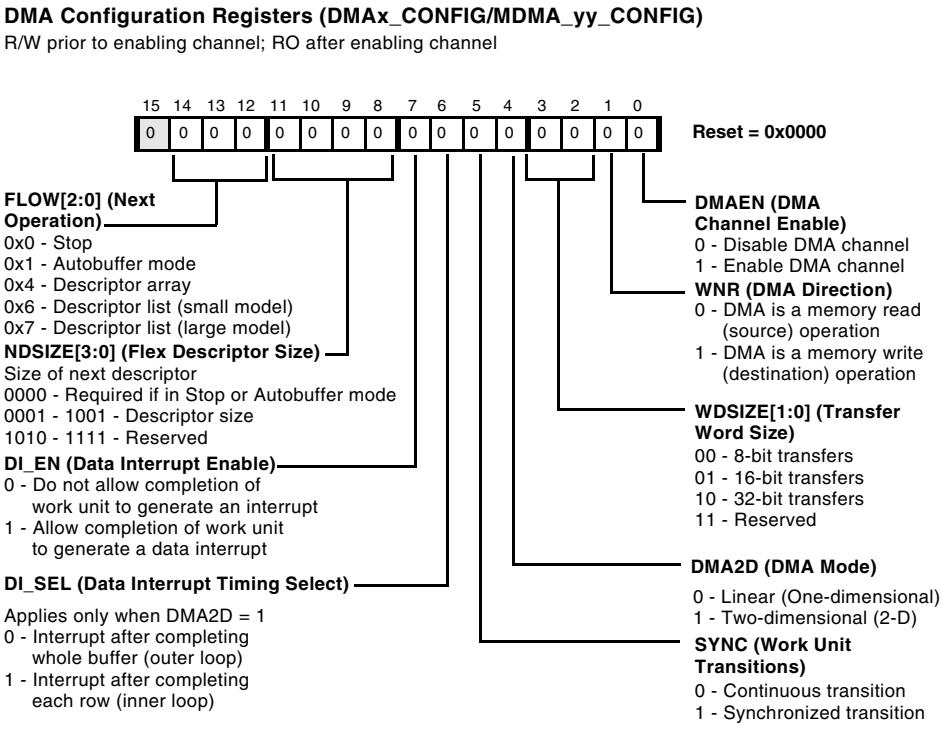


Figure 7-6. DMA Configuration Registers

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

- FLOW[2:0] (next operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:

  0x0 - stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The DMA_RUN status bit in the DMAx_IRQ_STATUS register changes from 1 to 0, while the DMAEN bit in the DMAx_CONFIG register is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to the DMAx_CONFIG register specifying the next work unit, in which the DMAEN bit is set to 1.

  0x1 - autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular buffer fashion based on user-programmed DMA MMR settings. Upon completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMAEN bit in the DMAx_CONFIG register.

  0x4 - descriptor array mode. This mode fetches a descriptor from memory that does not include the NDPH or NDPL elements. Because the descriptor does not contain a next descriptor pointer entry, the DMA engine defaults to using the DMAx_CURR_DESC_PTR register to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.

  0x6 - descriptor list (small model) mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the next descriptor pointer field are taken from the upper 16 bits of the DMAx_NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

0x7 - descriptor list (large model) mode. This mode fetches a descriptor from memory that includes NDPH and NDPL, thus allowing maximum flexibility in locating descriptors in memory.

- NDSIZE[3:0] (flex descriptor size). This field specifies the number of descriptor elements in memory to load. This field must be 0 if in stop or autobuffer mode. If NDSIZE and FLOW specify a descriptor that extends beyond YMOD, a DMA error results.
- DI_EN (data interrupt enable). This bit specifies whether to allow completion of a work unit to generate a data interrupt.
- DI_SEL (data interrupt timing select). This bit specifies the timing of a data interrupt—after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2-D DMA operation.
- SYNC (work unit transitions). This bit specifies whether the DMA channel performs a continuous transition (SYNC = 0) or a synchronized transition (SYNC = 1) between work units. For more information, see "Work Unit Transitions" on page 7-25.

  In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work unit.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be 0.

- DMA2D (DMA mode). This bit specifies whether DMA mode involves only DMAx_X_COUNT and DMAx_X_MODIFY (one-dimensional DMA) or also involves DMAx_Y_COUNT and DMAx_Y_MODIFY (two-dimensional DMA).
- WDSIZE[1:0] (transfer word size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port or through the 16-bit DMA access bus). The increment sizes (strides) of the DMA address pointer registers must be a multiple of the transfer unit size—one for 8-bit, two for 16-bit, four for 32-bit.

  Only SPORT DMA and Memory DMA can operate with a transfer size of 32 bits. All other peripherals have a maximum DMA transfer size of 16 bits.
- WNR (DMA direction). This bit specifies DMA direction—memory read (0) or memory write (1).
- DMAEN (DMA channel enable). This bit specifies whether to enable a given DMA channel.

When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

The DMAx_IRQ_STATUS register, shown in Figure 7-7, contains bits that record whether the DMA channel:

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

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

For a memory transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event has been signaled. It does not indicate the status of the DMA FIFO.

For MDMA transfers where an interrupt is not desired to notify when the DMA operation has ended, software should poll the DMA_DONE bit, rather than the DMA_RUN bit to determine when the transaction has completed.

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

- Data driven interrupts (see Table 7-5)
- Peripheral error interrupts
- DMA error interrupts (for example, bad descriptor or bus error)

Separate interrupt request (IRQ) levels are allocated for data, peripheral error, and DMA error interrupts.

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

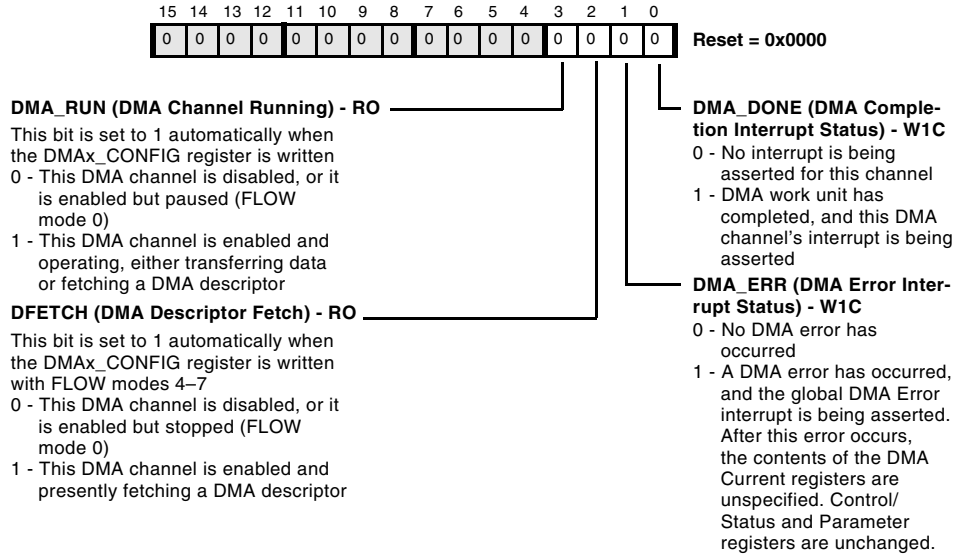


Figure 7-7. DMA Interrupt Status Registers

Table 7-5. Data Driven Interrupts

Interrupt Name	Description
No Interrupt	Interrupts can be disabled for a given work unit.
Peripheral Interrupt	These are peripheral (non-DMA) interrupts.
Row Completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer Completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

The DMA error conditions for all DMA channels are OR'd together into one system-level DMA error interrupt. The individual IRQ_STATUS words

of each channel can be read to identify the channel that caused the DMA error interrupt.

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

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

DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR)

The DMAx_START_ADDR register, shown in Figure 7-8, contains the start address of the data buffer currently targeted for DMA.

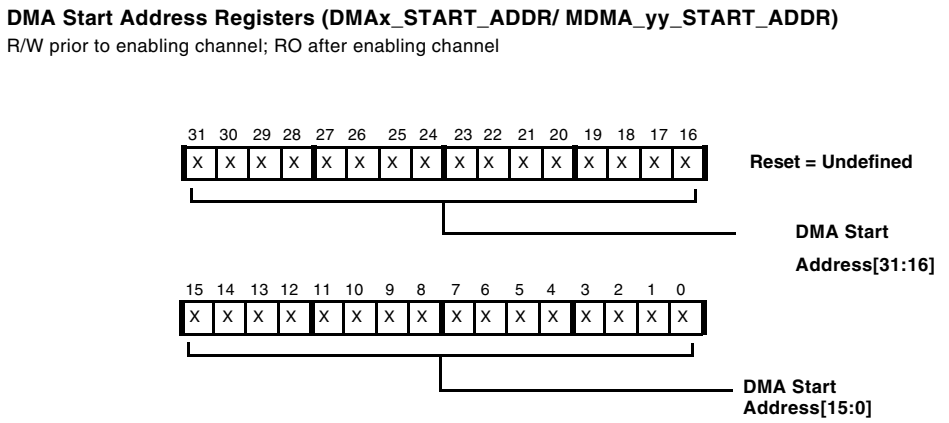


Figure 7-8. DMA Start Address Registers

DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)

The 32-bit DMAx_CURR_ADDR register shown in Figure 7-9, contains the present DMA transfer address for a given DMA session. On the first memory transfer of a DMA work unit, the DMAx_CURR_ADDR register is loaded from the DMAx_START_ADDR register, and it is incremented as each transfer occurs.

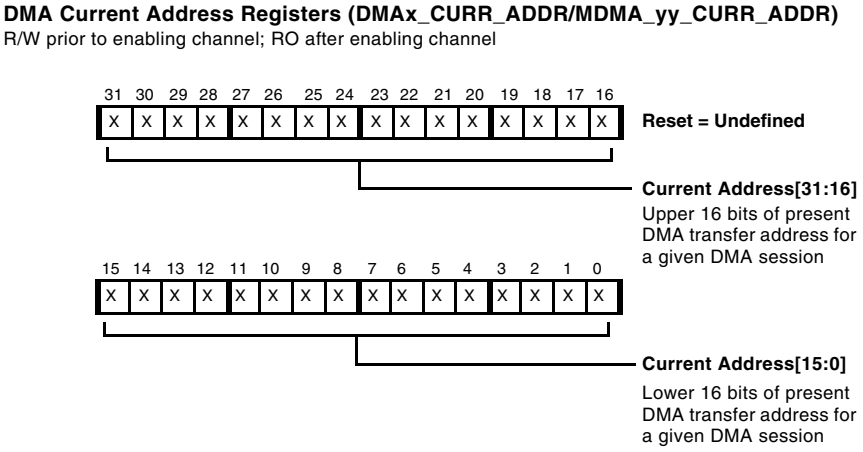


Figure 7-9. DMA Current Address Registers

DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)

For 2-D DMA, the DMAx_X_COUNT register, shown in Figure 7-10, contains the inner loop count. For 1-D DMA, it specifies the number of elements to transfer. For details, see "Two-Dimensional DMA Operation" on page 7-11. A value of 0 in DMAx_X_COUNT corresponds to 65,536 elements.

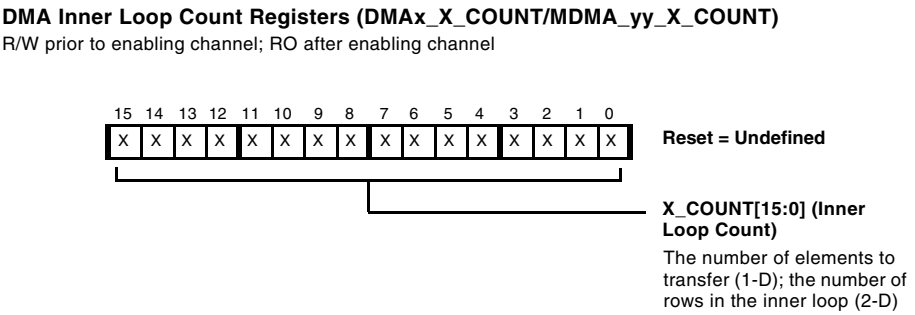


Figure 7-10. DMA Inner Loop Count Registers

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT)

The DMAx_CURR_X_COUNT register, shown in Figure 7-11, holds the number of transfers remaining in the current DMA row (inner loop). On the first memory transfer of each DMA work unit, it is loaded with the value in the DMAx_X_COUNT register and then decremented. For 2-D DMA, on the last memory transfer in each row except the last row, it is reloaded with the value in the DMAx_X_COUNT register; this occurs at the same time that the value in the DMAx_CURR_Y_COUNT register is decremented. Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2-D DMA, the DMAx_CURR_X_COUNT register value is 0 only when the entire transfer is complete. Between rows it is equal to the value of the DMAx_X_COUNT register.

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT/ MDMA_yy_CURR_X_COUNT)

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

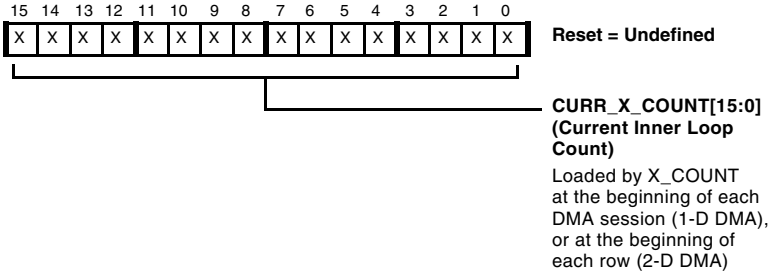


Figure 7-11. DMA Current Inner Loop Count Registers

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)

The DMAx_X_MODIFY register, shown in Figure 7-12, contains a signed, two's-complement byte-address increment. In 1-D DMA, this increment is the stride that is applied after transferring each element.

DMAx_X_MODIFY is specified in bytes, regardless of the DMA transfer size.

In 2-D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, the DMAx_Y_MODIFY register is applied instead, except on the very last transfer of each work unit. The DMAx_X_MODIFY register is always applied to the last transfer of a work unit.

The DMAx_X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)
R/W prior to enabling channel; RO after enabling channel

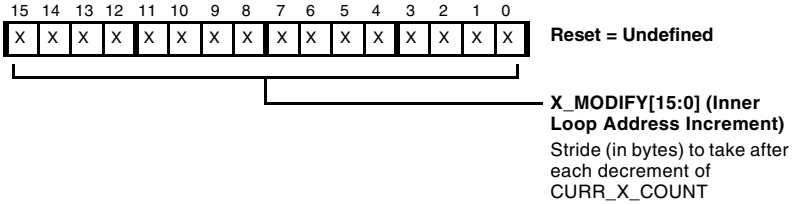


Figure 7-12. DMA Inner Loop Address Increment Registers

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)

For 2-D DMA, the DMAx_Y_COUNT register, shown in Figure 7-13, contains the outer loop count. It is not used in 1-D DMA mode. This register contains the number of rows in the outer loop of a 2-D DMA sequence. For details, see "Two-Dimensional DMA Operation" on page 7-11.

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)
R/W prior to enabling channel; RO after enabling channel

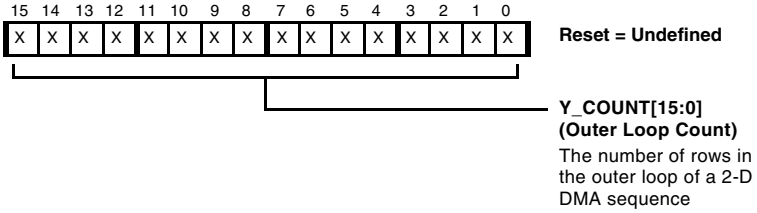


Figure 7-13. DMA Outer Loop Count Registers

DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT)

The DMAx_CURR_Y_COUNT register, used only in 2-D mode, holds the number of full or partial rows (outer loops) remaining in the current work unit. See Figure 7-14. On the first memory transfer of each DMA work unit, it is loaded with the value of the DMAx_Y_COUNT register. The register is decremented each time the DMAx_CURR_X_COUNT register expires during 2-D DMA operation (1 to DMAx_X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2-D DMA session is complete, DMAx_CURR_Y_COUNT = 1 and DMAx_CURR_X_COUNT = 0.

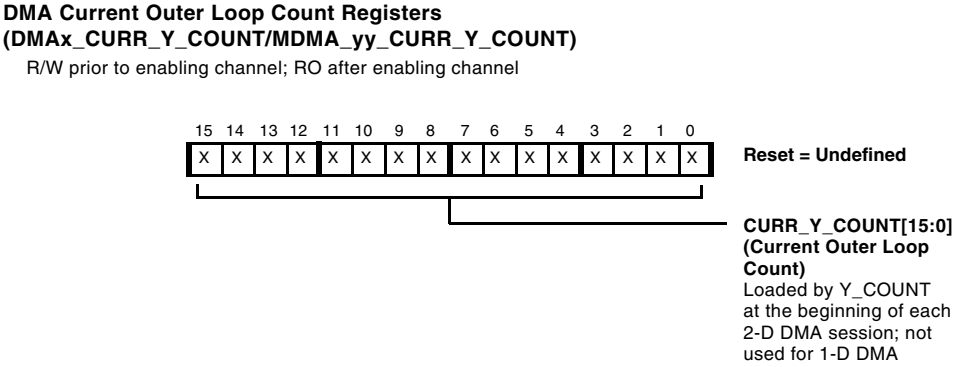


Figure 7-14. DMA Current Outer Loop Count Registers

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY)

The DMAx_Y_MODIFY register contains a signed, two's-complement value. See Figure 7-15. This byte-address increment is applied after each decrement of the DMAx_CURR_Y_COUNT register except for the last item in the 2-D array where the DMAx_CURR_Y_COUNT also expires. The value is the offset between the last word of one row and the first word

of the next row. For details, see "Two-Dimensional DMA Operation" on page 7-11.

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

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/ MDMA_yy_Y_MODIFY)
R/W prior to enabling channel; RO after enabling channel

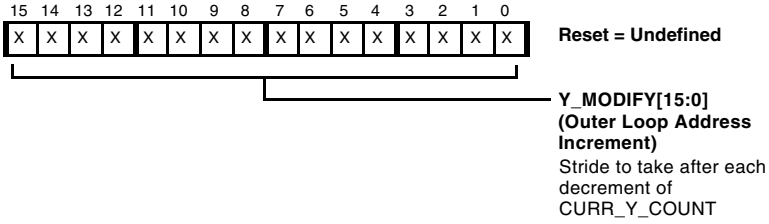


Figure 7-15. DMA Outer Loop Address Increment Registers

DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR)

The 32-bit DMAx_NEXT_DESC_PTR register, shown in Figure 7-16, specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, this register is copied into the DMAx_CURR_DESC_PTR register. Then, during the descriptor fetch, the DMAx_CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, the DMAx_NEXT_DESC_PTR register, and not the DMAx_CURR_DESC_PTR register, must be programmed directly via MMR access before starting DMA operation.

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

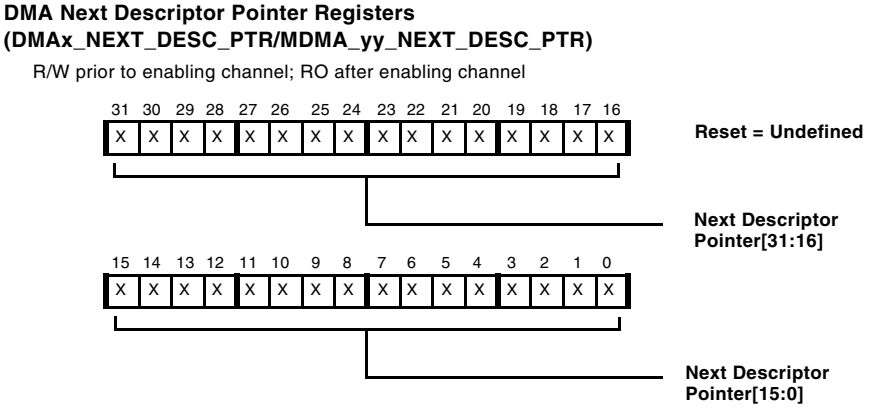


Figure 7-16. DMA Next Descriptor Pointer Registers

DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

The 32-bit DMAx_CURR_DESC_PTR register, shown in Figure 7-17, contains the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW = 4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (FLOW = 6 or 7), this register is initialized from the DMAx_NEXT_DESC_PTR register before loading each descriptor. Then, the address in the DMAx_CURR_DESC_PTR register increments as each descriptor element is read in.

When the entire descriptor has been read, the `DMAx_CURR_DESC_PTR` register contains this value:

Descriptor Start Address + (2 × Descriptor Size) (# of elements)

For descriptor array mode (FLOW = 4), this register, and not the DMAx_NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

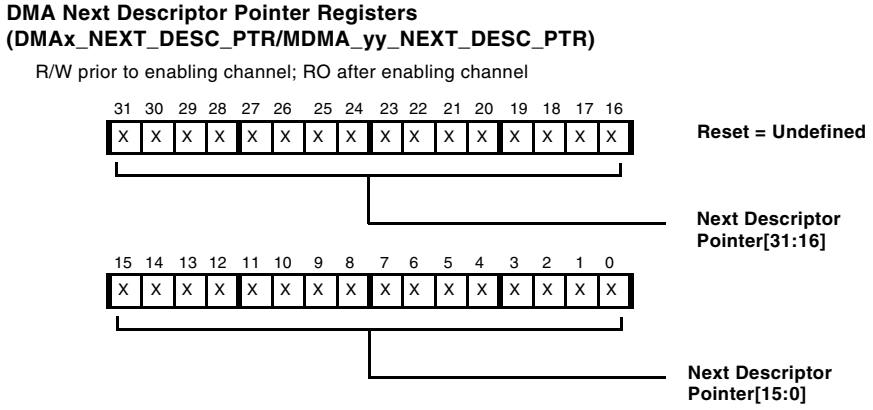


Figure 7-17. DMA Current Descriptor Pointer Registers

HMDMA Registers

Some processors have two HMDMA blocks, while others have none. See the "Unique Information for the ADSP-BF50x Processor" on page 7-105 to determine whether this feature is applicable to your product. HMDMA0 is associated with MDMA0, and HMDMA1 is associated with MDMA1.

Handshake MDMA Control Registers (HMDMAx_CONTROL)

The HMDMAx_CONTROL register, shown in Figure 7-18, is used to set up HMDMA parameters and operating modes.

The `DRQ[1:0]` field is used to control the priority of the MDMA channel when the HMDMA is disabled, that is, when handshake control is not being used (see Table 7-6).

Table 7-6. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
00	Disabled	The MDMA request is disabled.
01	Enabled/S	Normal MDMA channel priority. The channel in this mode is limited to single memory transfers separated by one idle system clock. Request single transfer from MDMA channel.
10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
11	Urgent	The MDMA channel priority is elevated to urgent. In this state, it has higher priority for memory access than non-urgent channels. If two channels are both urgent, the lower-numbered channel has priority.

The RBC bit forces the BCOUNT register to be reloaded with the BCINIT value while the module is already active. Do not set this bit in the same write that sets the HMDMAEN bit to active.

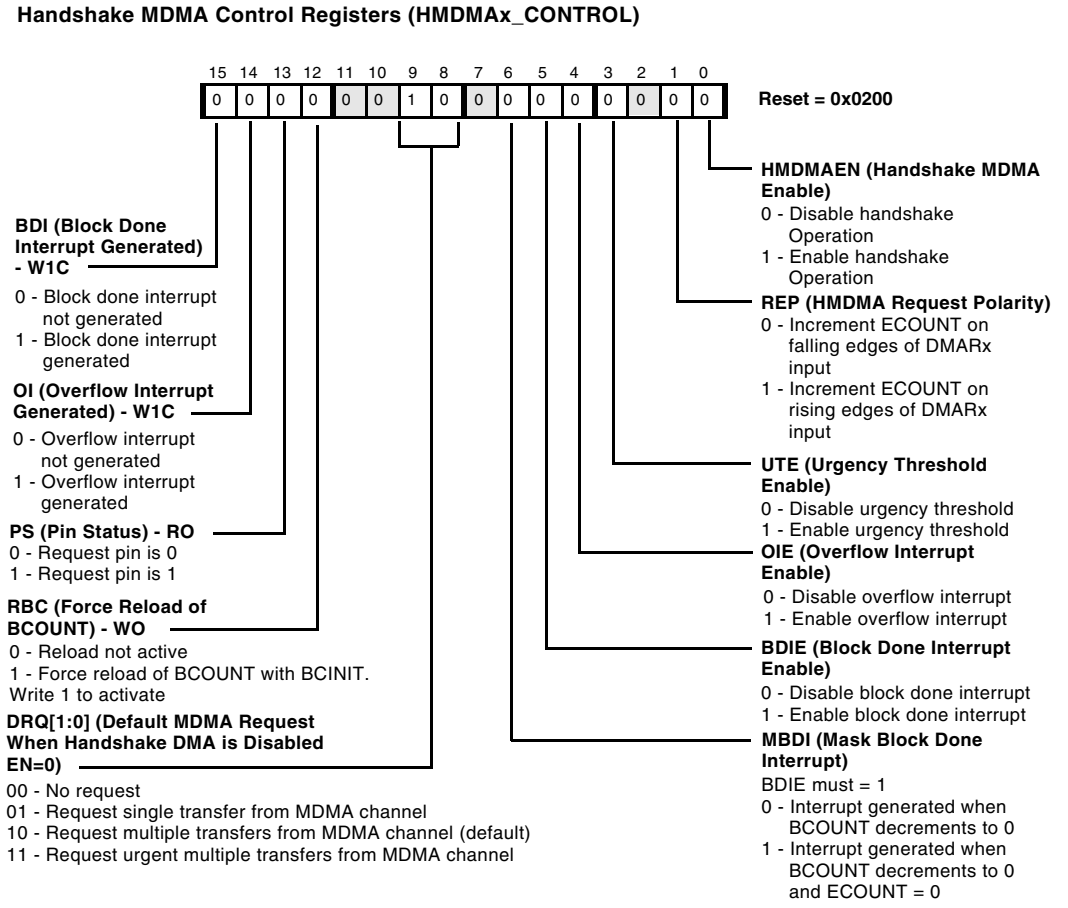


Figure 7-18. Handshake MDMA Control Registers

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)

The HMDMAx_BCINIT register, shown in Figure 7-19, holds the number of transfers to do per edge of the DMARx control signal.

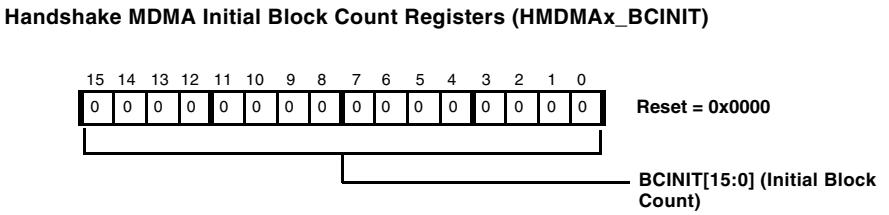


Figure 7-19. Handshake MDMA Initial Block Count Registers

Handshake MDMA Current Block Count Registers (HMDMAx_BCOUNT)

The HMDMAx_BCOUNT register, shown in Figure 7-20, holds the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0000 = 0 transfers remaining
- FFFF = 65535 transfers remaining

The BCOUNT field is loaded with BCINIT when ECOUNT is greater than 0 and BCOUNT is expired (0). Also, if the RBC bit in the HMDMAx_CONTROL register is written to 1, BCOUNT is loaded with BCINIT. The BCOUNT field is decremented with each MDMA grant. It is cleared when HMDMA is disabled.

A block done interrupt is generated when BCOUNT decrements to 0. If the MBDI bit in the HMDMAx_CONTROL register is set, the interrupt is suppressed until ECOUNT is 0. If BCINIT is 0, no block done interrupt is generated, since no DMA requests were generated or grants received.

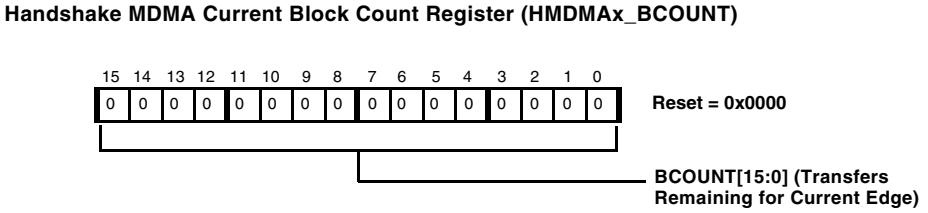


Figure 7-20. Handshake MDMA Current Block Count Registers

Handshake MDMA Current Edge Count Registers (HMDMAx_ECOUNT)

The HMDMAx_ECOUNT register, shown in Figure 7-21, holds a signed number of edges remaining to be serviced. This number is in a signed two's complement representation. When an edge is detected on the respective DMARx input, requests occur if this count is greater than or equal to 0 and BCOUNT is greater than 0.

When the handshake mode is enabled, ECOUNT is loaded and the resulting number of requests is:

Number of edges + N,

where N is the number loaded from ECINIT. The number N can be positive or negative. Examples:

- 0x7FFF = 32,767 edges remaining
- 0x0000 = 0 edges remaining
- 0x8000 = –32,768: ignore the next 32,768 edges

Each time that BCOUNT expires, ECOUNT is decremented and BCOUNT is reloaded from BCINIT. When a handshake request edge is detected, ECOUNT is incremented. The ECOUNT field is cleared when HMDMA is disabled.

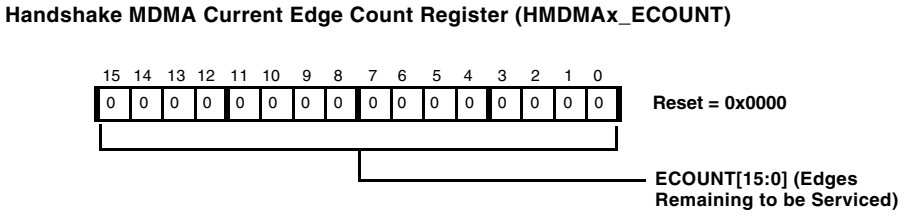


Figure 7-21. Handshake MDMA Current Edge Count Registers

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)

The HMDMAx_ECINIT register, shown in Figure 7-22, holds a signed number that is loaded into HMDMAx_ECOUNT when handshake DMA is enabled. This number is in a signed two's complement representation.

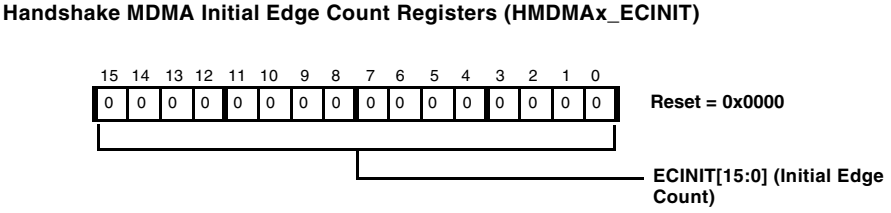


Figure 7-22. Handshake MDMA Initial Edge Count Registers

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)

The HMDMAx_ECURGENT register, shown in Figure 7-23, holds the urgent threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than this threshold, the MDMA request is urgent and might get higher priority.

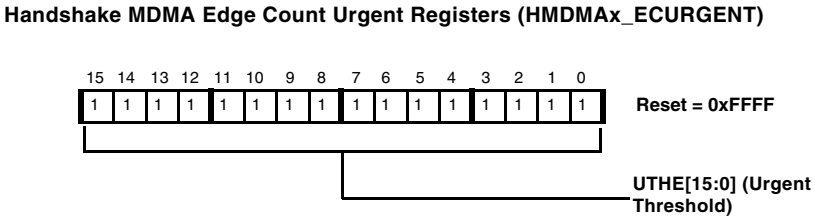


Figure 7-23. Handshake MDMA Edge Count Urgent Registers

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)

The HMDMAx_ECOVERFLOW register, shown in Figure 7-24, holds the interrupt threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than this threshold, an overflow interrupt is generated.

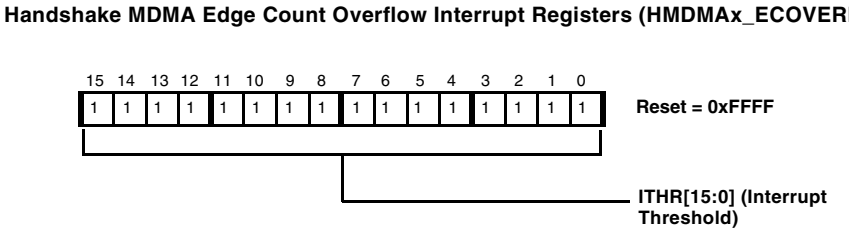


Figure 7-24. Handshake MDMA Edge Count Overflow Interrupt Registers

DMA Traffic Control Registers (DMA_TC_PER and DMA_TC_CNT)

The DMA_TC_PER register (see Figure 7-25) and the DMA_TC_CNT register (see Figure 7-26) work with other DMA registers to define traffic control.

DMA_TC_PER Register

DMA Traffic Control Counter Period Register (DMA_TC_PER)

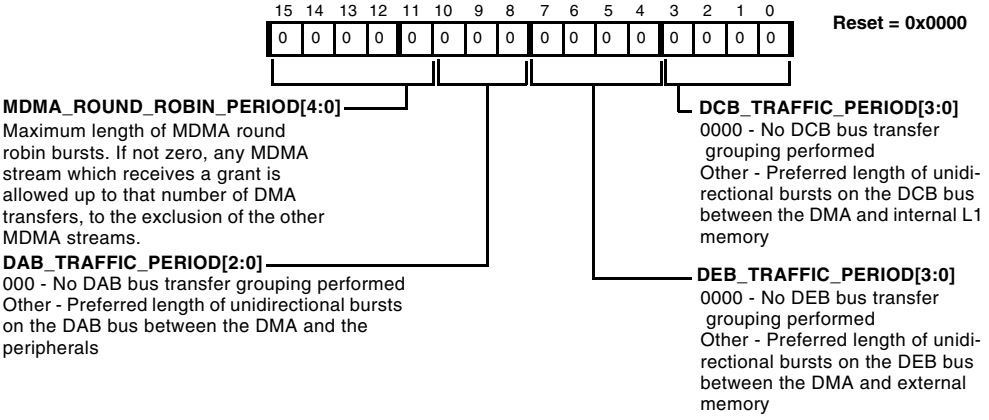


Figure 7-25. DMA Traffic Control Counter Period Register

DMA_TC_CNT Register

DMA Traffic Control Counter Register (DMA_TC_CNT)
RO

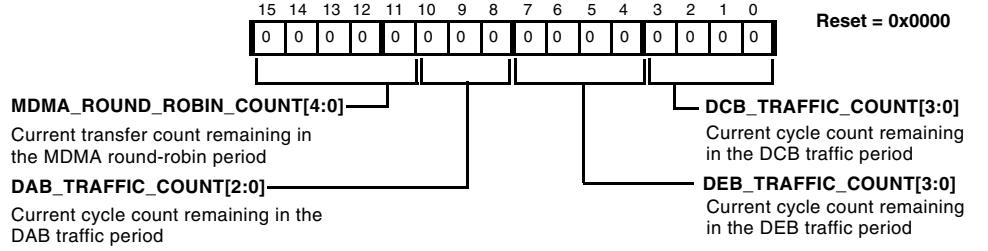


Figure 7-26. DMA Traffic Control Counter Register

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round-robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMA_TC_PER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever DMA_TC_PER is written or whenever the DEB bus changes direction or becomes idle. It then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMA_TC_PER is written or whenever the DCB bus changes direction or becomes idle. It then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same

direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from `DCB_TRAFFIC_PERIOD` to begin a new burst.

Programming Examples

The following examples illustrate memory DMA and handshaked memory DMA basics. Examples for peripheral DMAs can be found in the respective peripheral chapters.

Register-Based 2-D Memory DMA

Listing 7-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel re-sorts elements by transposing the two-dimensional data array. See Figure 7-27.

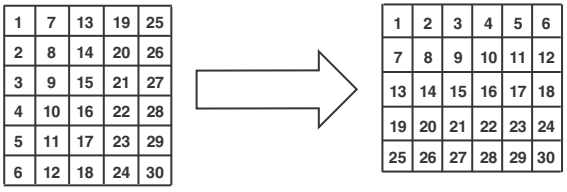


Figure 7-27. DMA Example, 2-D Array

The two arrays reside in two different L1 data memory blocks. However, the arrays could reside in any internal or external memory, including L1 instruction memory and SDRAM. For the case where the destination array resided in SDRAM, it is a good idea to let the source channel re-sort elements and to let the destination buffer store linearly.

Listing 7-1. Register-Based 2-D Memory DMA

```
#include <defBF527.h>/*For ADSP-BF527 product, as an example.*/
#define X 5
#define Y 6

.section L1_data_a;
.byte2 aSource[X*Y] =
    1,  7, 13, 19, 25,
    2,  8, 14, 20, 26,
    3,  9, 15, 21, 27,
    4, 10, 16, 22, 28,
    5, 11, 17, 23, 29,
    6, 12, 18, 24, 30;

.section L1_data_b;
.byte2 aDestination[X*Y];

.section L1_code;
.global _main;
_main:
    p0.l = lo(MDMA_S0_CONFIG);
    p0.h = hi(MDMA_S0_CONFIG);
    call memdma_setup;
    call memdma_wait;
_main.forever:
    jump _main.forever;
_main.end:
```

The setup routine shown in Listing 7-2 initializes either MDMA0 or MDMA1, depending on whether the MMR address of `MDMA_S0_CONFIG` or `MDMA_S1_CONFIG` is passed in the `P0` register. Note that the source channel is enabled before the destination channel. Also, it is common to synchronize interrupts with the destination channel because only those interrupts indicate completion of both DMA read and write operations.

Listing 7-2. Two-Dimensional Memory DMA Setup Example

```
memdma_setup:
    [--sp] = r7;
/* setup 1D source DMA for 16-bit transfers */
    r7.l = lo(aSource);
    r7.h = hi(aSource);
    [p0 + MDMA_S0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2;
    w[p0 + MDMA_S0_X_MODIFY - MDMA_S0_CONFIG] = r7;
     r7.l = X * Y;
    w[p0 + MDMA_S0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = WDSIZE_16 | DMAEN;
    w[p0] = r7;
/* setup 2D destination DMA for 16-bit transfers */
    r7.l = lo(aDestination);
    r7.h = hi(aDestination);
    [p0 + MDMA_D0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2*Y;
    w[p0 + MDMA_D0_X_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = Y;
    w[p0 + MDMA_D0_Y_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = X;
    w[p0 + MDMA_D0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = -2 * (Y * (X-1) - 1);
    w[p0 + MDMA_D0_Y_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = DMA2D | DI_EN | WDSIZE_16 | WNR | DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_setup.end:
```

For simplicity the example shown in Listing 7-3 polls the DMA status rather than using interrupts, which is the normal case in a real application.

Listing 7-3. Polling DMA Status

```
memdma_wait:
    [--sp] = r7;
memdma_wait.test:
    r7 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r7, bitpos(DMA_DONE));
    if !CC jump memdma_wait.test;
    r7 = DMA_DONE (z);
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_wait.end:
```

Initializing Descriptors in Memory

Descriptor-based DMAs expect the descriptor data to be available in memory by the time the DMA is enabled. Often, the descriptors are programmed by software at run-time. Many times, however, the descriptors—or at least large portions of them—can be static and therefore initialized at boot time. How to set up descriptors in global memory depends heavily on the programming language and the tool set used. The following examples show how this is best performed in the VisualDSP++ tools' assembly language.

Listing 7-4 uses multiple variables of either 16-bit or 32-bit size to describe DMA descriptors. This example has two descriptors in small list flow mode that point to each other. At the end of the second work unit, an interrupt is generated without discontinuing the DMA processing. The trailing `.end` label is required to let the linker know that a descriptor forms a logical unit. It prevents the linker from removing variables when optimizing.

Listing 7-4. Two Descriptors in Small List Flow Mode

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

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

.byte2 descBlock2 = lo(descBlock1);
.var   descBlock2.addr = arrBlock2;
.byte2 descBlock2.cfg =
FLOW_SMALL|NDSIZE_5|DI_EN|WDSIZE_16|DMAEN;
.byte2 descBlock2.len = length(arrBlock2);
       descBlock2.end:
```

Another method featured by the VisualDSP++ tools takes advantage of C-style structures in global header files. The header file `descriptors.h` could look like Listing 7-5.

Listing 7-5. Header File to Define Descriptor Structures

```
#ifndef __INCLUDE_DESCRIPTORS__
#define __INCLUDE_DESCRIPTORS__
#ifdef _LANGUAGE_C
typedef struct {
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
```

```
    short dYModify;
} dma_desc_arr;

typedef struct {
    void  *pNext;
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
    short dYModify;
} dma_desc_list;

#endif // _LANGUAGE_C
#endif // __INCLUDE_DESCRIPTORS__
```

Note that near pointers are not natively supported by the C language and, thus, pointers are always 32 bits wide. Therefore, the scheme above cannot be used directly for small list mode without giving up pointer syntax. The variable definition file is required to import the C-style header file and can finally take advantage of the structures. See Listing 7-6.

Listing 7-6. Using Descriptor Structures

```
#include "descriptors.h"
.import "descriptors.h";

.section L1_data_a;
.align 4;
.var arrBlock3[N];
.var arrBlock4[N];

.struct dma_desc_list descBlock3 = {
    descBlock4, arrBlock3,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_32 | DMAEN,
```

```
    length(arrBlock3), 4,
    0, 0        /* unused values */
};

.struct dma_desc_list descBlock4 = {
    descBlock3, arrBlock4,
    FLOW_LARGE | NDSIZE_7 | DI_EN | WDSIZE_32 | DMAEN,
    length(arrBlock4), 4,
    0, 0        /* unused values */
};
```

Software-Triggered Descriptor Fetch Example

Listing 7-7 demonstrates a large list of descriptors that provide `FLOW` = 0 (stop mode) configuration. Consequently, the DMA stops by itself as soon as the work unit has finished. Software triggers the next work unit by simply writing the proper value into the DMA configuration registers. Since these values instruct the DMA controller to fetch descriptors in large list mode, the DMA immediately fetches the descriptor, thus overwriting the configuration value again with the new settings when it is started.

Note the requirement that source and destination channels stop after the same number of transfers. Between stops, the two channels can have completely individual structures.

Listing 7-7. Software-Triggered Descriptor Fetch

```
.import "descriptors.h";

#define N 4
.section L1_data_a;
.byte2 arrSource1[N] = { 0x1001, 0x1002, 0x1003, 0x1004 };
.byte2 arrSource2[N] = { 0x2001, 0x2002, 0x2003, 0x2004 };
.byte2 arrSource3[N] = { 0x3001, 0x3002, 0x3003, 0x3004 };
```

```
.byte2 arrDest1[N];
.byte2 arrDest2[2*N];

.struct dma_desc_list descSource1 = {
    descSource2, arrSource1,
    WDSIZE_16 | DMAEN,
    length(arrSource1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource2 = {
    descSource3, arrSource2,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_16 | DMAEN,
    length(arrSource2), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource3 = {
    descSource1, arrSource3,
    WDSIZE_16 | DMAEN,
    length(arrSource3), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest1 = {
    descDest2, arrDest1,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest2 = {
    descDest1, arrDest2,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest2), 2,
    0, 0        /* unused values */
};
```

```
.section L1_code;
_main:
/* write descriptor address to next descriptor pointer */
    p0.h = hi(MDMA_S0_CONFIG);
    p0.l = lo(MDMA_S0_CONFIG);
    r0.h = hi(descDest1);
    r0.l = lo(descDest1);
    [p0 + MDMA_D0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;
    r0.h = hi(descSource1);
    r0.l = lo(descSource1);
    [p0 + MDMA_S0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;

/* start first work unit */
    r6.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|DMAEN;
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    r7.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|WNR|DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;

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

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

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

Handshaked Memory DMA Example

The functional block for the handshaked MDMA operation can be considered completely separately from the MDMA channels themselves. Therefore the following HMDMA setup routine can be combined with any of the MDMA examples discussed above. Be sure that the HMDMA module is enabled before the MDMA channels.

Listing 7-8 enables the HMDMA1 block, which is controlled by the DMAR1 pin and is associated with the MDMA1 channel pair.

Listing 7-8. HMDMA1 Block Enable

```
/* optionally, enable all four bank select strobes */
    p1.l = lo(EBIU_AMGCTL);
    p1.h = hi(EBIU_AMGCTL);
    r0.l = 0x0009;
    w[p1] = r0;

/* function enable for DMAR1 */
    p1.l = lo(PORTG_FER);
    r0.l = PG12;
    w[p1] = r0;
    p1.l = lo(PORTG_MUX);
    r0.l = 0x0000;
    w[p1] = r0;

/* every single transfer requires one DMAR1 event */
    p1.l = lo(HMDMA1_BCINIT);
    r0.l = 1;
    w[p1] = r0;

/* start with balanced request counter */
    p1.l = lo(HMDMA1_ECINIT);
    r0.l = 0;
```

```
    w[p1] = r0;

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

If the HMDMA is intended to copy from internal memory to external devices, the above setup is sufficient. If, however, the data flow is from outside the processor to internal memory, then this small issue must be considered—the HMDMA only controls the destination channel of the memory DMA. It does not gate requests to the source channel at all. Thus, as soon as the source channel is enabled, it starts filling the DMA FIFO immediately. In 16-bit DMA mode, this results in eight read strobes on the EBIU even before the first DMAR1 event has been detected. In other words, the transferred data and the DMAR1 strobes are eight positions off. The example in Listing 7-9 delays processing until eight DMAR1 requests have been received. By doing so, the transmitter is required to add eight trailing dummy writes after all data words have been sent. This is because the transmit channel still has to drain the DMA FIFO.

Listing 7-9. HMDMA With Delayed Processing

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

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

```
w[p1] = r0;
w[p1] = r2;
```

If the polling operation shown in Listing 7-9 is too expensive, an interrupt version of it can be implemented by using the HMDMA overflow feature. Temporarily set the HMDMAx_OVERFLOW register to eight.

Unique Information for the ADSP-BF50x Processor

Figure 7-28 provides a block diagram of the ADSP-BF50x DMA controller.

The ADSP-BF50x processors do not contain an SDRAM interface or an HMDMA controller. Therefore, any discussion or examples above regarding SDRAM and HMDMA do not apply to the ADSP-BF50x processors.

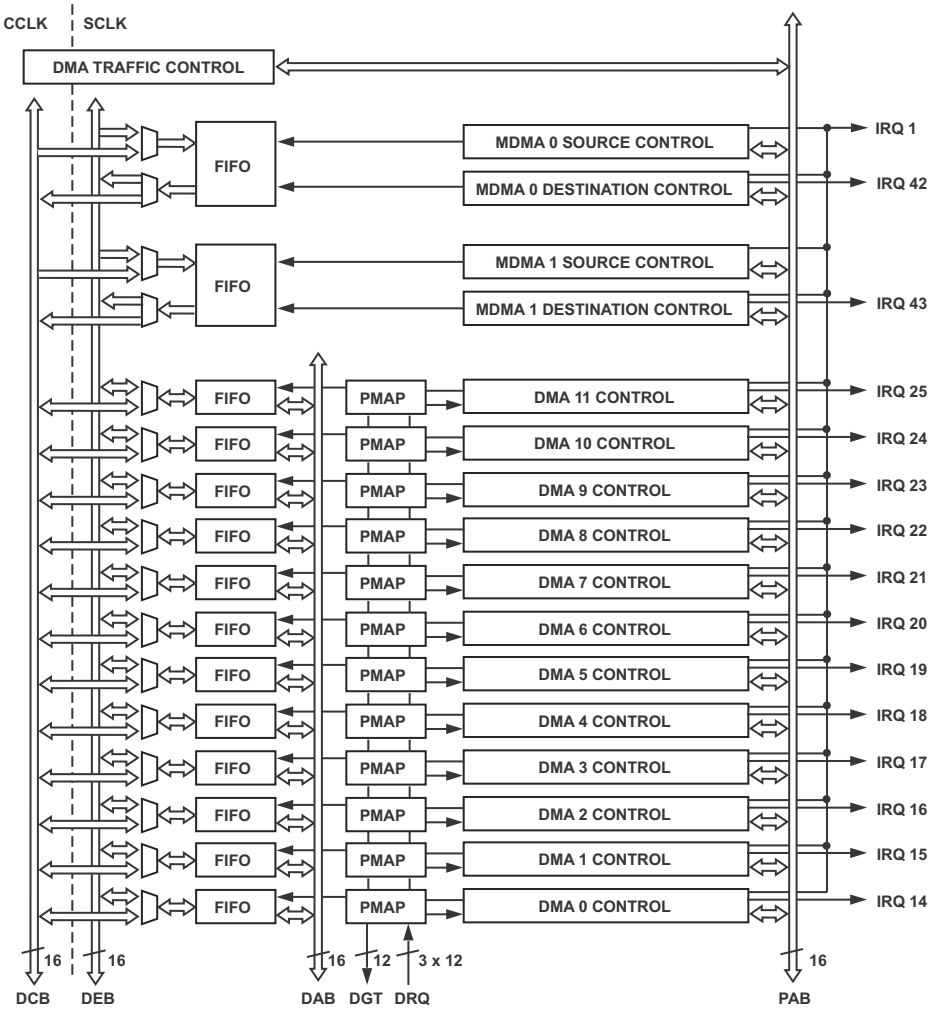


Figure 7-28. ADSP-BF50x DMA Controller Block Diagram

Static Channel Prioritization

The default DMA channel priority and mapping shown in Table 7-7 can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

Table 7-7. Priority and Default Mapping of Peripheral to DMA

Priority	DMA Channel	PMAP Default Value	Peripheral Mapped by Default
Highest	DMA 0	0x0	PPI receive or transmit
	DMA 1	0x1	RSI receive or transmit
	DMA 2	0x2	SPORT0 receive
	DMA 3	0x3	SPORT0 transmit
	DMA 4	0x4	SPORT1 receive
	DMA 5	0x5	SPORT1 transmit
	DMA 6	0x6	SPI0 receive or transmit
	DMA 7	0x7	SPI1 receive or transmit
	DMA 8	0x8	UART0 receive
	DMA 9	0x9	UART0 transmit
	DMA 10	0xA	UART1 receive
	DMA 11	0xB	UART1 transmit
	MDMA D0	None	Mem DMA has no peripheral mapping.
	MDMA S0	None	Mem DMA has no peripheral mapping.
	MDMA D1	None	Mem DMA has no peripheral mapping.
Lowest	MDMA S1	None	Mem DMA has no peripheral mapping.

8 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the Blackfin processor and includes the following sections:

Phase Locked Loop and Clock Control

The input clock into the processor, CLKIN, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip PLL module. During normal operation, the user programs the PLL with a multiplication factor for CLKIN. The resulting, multiplied signal is the voltage controlled oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (CCLK).

A user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the Peripheral Access Bus (PAB),

DMA Access Bus (DAB), External Access Bus (EAB), and the external bus interface unit (EBIU).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in the processor data sheet.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a "coarse adjustment." For a "fine adjustment," the PLL clock frequency can also be varied.

PLL Overview

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the Dynamic Power Management Controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see "Dynamic Power Management Controller" on page 8-7.

Subject to the maximum VCO frequency specified in the processor data sheet, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

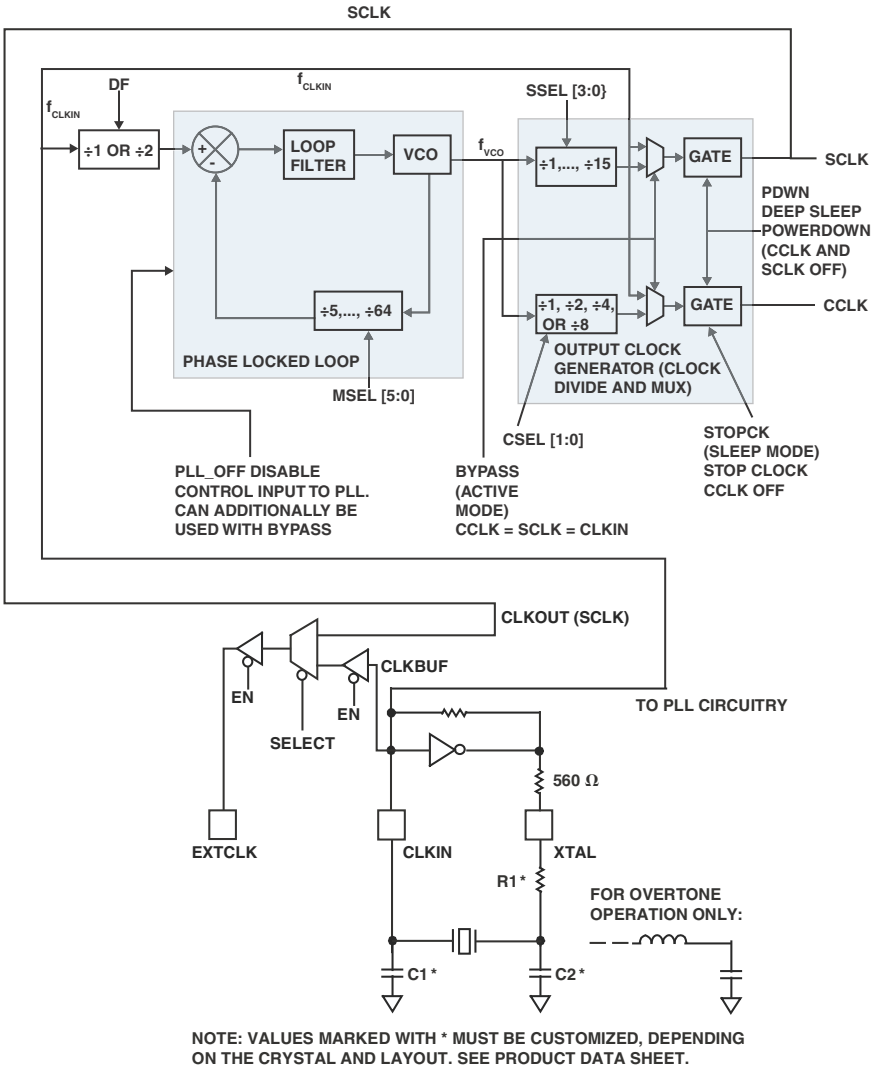


Figure 8-1. PLL Block Diagram

Figure 8-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an

intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL_CTL Register" on page 8-22.

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers

The reset value of MSEL is 0x6. This value can be reprogrammed at startup in the boot code.

Table 8-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See the processor data sheet for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 8-1. MSEL Encodings

Signal name	VCO Frequency	
MSEL[5:0]	DF = 0	DF = 1
5	5x	1.5x
6	6x	3x
N = 7–62	Nx	0.5Nx

Table 8-1. MSEL Encodings (Cont'd)

Signal name	VCO Frequency	
MSEL[5:0]	DF = 0	DF = 1
63	63x	31.5x
0	64x	32x

The PLL control (PLL_CTL) register controls operation of the PLL (see Figure 8-4 on page 8-22). Note that changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with a new value, and then a specific PLL programming sequence must be executed to implement the changes. This is handled automatically by the system control ROM function (bfrom_SysControl()) as described in "System Control ROM Function" on page 8-24.

Core Clock/System Clock Ratio Control

Table 8-2 describes the programmable relationship between the VCO frequency and the core clock. Table 8-3 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to limit the SCLK to a frequency specified in the processor data sheet. The SCLK drives all synchronous, system-level logic.

The divider ratio control bits, CSEL and SSEL, are in the PLL divide (PLL_DIV) register. For information about this register, see "PLL_DIV Register" on page 8-21.

The reset value of CSEL[1:0] is 0x0, and the reset value of SSEL[3:0] is 0x4. These values can be reprogrammed at startup by the boot code.

By updating PLL_DIV with an appropriate value, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

Unlike writing the PLL_CTL register, the PLL_DIV register can be programmed at any time to change the CCLK and SCLK divide values without entering the PLL programing sequence.

Table 8-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
00	1	300	300
01	2	300	150
10	4	400	100
11	8	400	50

As long as the MSEL and DF control bits in the PLL control (PLL_CTL) register remain constant, the PLL is locked.

Table 8-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	50	50
0010	2:1	150	75
0011	3:1	150	50
0100	4:1	200	50
0101	5:1	300	60
0110	6:1	360	60
N = 7–15	N:1	400	400/N

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency. The PLL lock count (PLL_LOCKCNT) register defines the number of CLKIN cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each CLKIN cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See the processor data sheet for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see "Operating Modes" on page 8-8.

Dynamic Power Management Controller

The Dynamic Power Management Controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor's performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See "Operating Modes" on page 8-8.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.
- Voltage control – The V_{DDINT} domain must be powered by an external voltage regulator. For more information see "Voltage Regulation Interface" on page 25-11.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 8-4 summarizes the operational characteristics of each mode.

Table 8-4. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The active and full-on states of the DPMC/PLL can be determined by reading the PLL status register (see "PLL_STAT Register" on page 8-22). In these modes, the core can either execute instructions or be in the `IDLE` core state. If the core is in the `IDLE` state, it can be awakened by several sources (see Chapter 4, "System Interrupts" for details).

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full-On Mode

Full-on mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full-on mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. The system clock (`SCLK`) frequency is determined by the `SSEL` specified ratio to VCO. DMA access is available to L1 and external memories. From full-on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in Figure 8-2 on page 8-13.

Active Mode

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

In active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to full-on or sleep modes.

From active mode, the processor can transition directly to full-on, sleep, or deep sleep modes.

In this mode or in the transition phase to other modes, changes to MSEL are not latched by the PLL.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the processor core. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. Since CCLK is disabled, DMA access is available only to external memory in sleep mode. From sleep mode, a wakeup event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full-on mode if the BYPASS bit is cleared

When sleep mode is exited, the processor resumes execution from the program counter value present immediately prior to entering sleep mode.

Deep Sleep Mode

Deep sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals (except

those enabled as wakeup sources) are disabled. DMA is not supported in this mode.

Deep sleep mode can be exited only by a hardware reset event, by a wakeup event on a programmable flag pin (including PH0, PF8, or PF9), or by a wakeup event on the programmable flag pin associated with the CAN_RX signal (PG1). A hardware reset begins the hardware reset sequence. For more information about hardware reset, see Chapter 4, "System Interrupts". A programmable flag event causes the processor to transition to active mode, and execution resumes at the program counter value at which the processor entered deep sleep mode. If an interrupt is also enabled in SIC_IMASK, the interrupt is vectored immediately after exit of deep sleep, and the related ISR executed.

Note that a programmable flag event in deep sleep mode automatically resets some fields in the PLL control (PLL_CTL) register. See Table 8-5.

Table 8-5. PLL_CTL Values after Programmable Flag Event

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down by the external regulator, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 8-2. This feature is discussed in detail in "Powering Down the Core (Hibernate State)" on page 8-19.

Operating Mode Transitions

Figure 8-2 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes and rectangles represent processor states. Arrows show the allowed transitions into and out of each mode or state.

For mode transitions, the text next to each transition arrow shows the fields in the PLL control (PLL_CTL) register that must be changed for the transition to occur. For example, the transition from full-on mode to sleep mode indicates that the STOPCK bit must be set to 1 and the PDWN bit must be set to 0.

For transitions to processor states, the text next to each transition arrow shows either a processor event (hardware reset or wakeup event) or the fields in the voltage regulator control register (VR_CTL) that must be changed for the transition to occur.

For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 8-15.

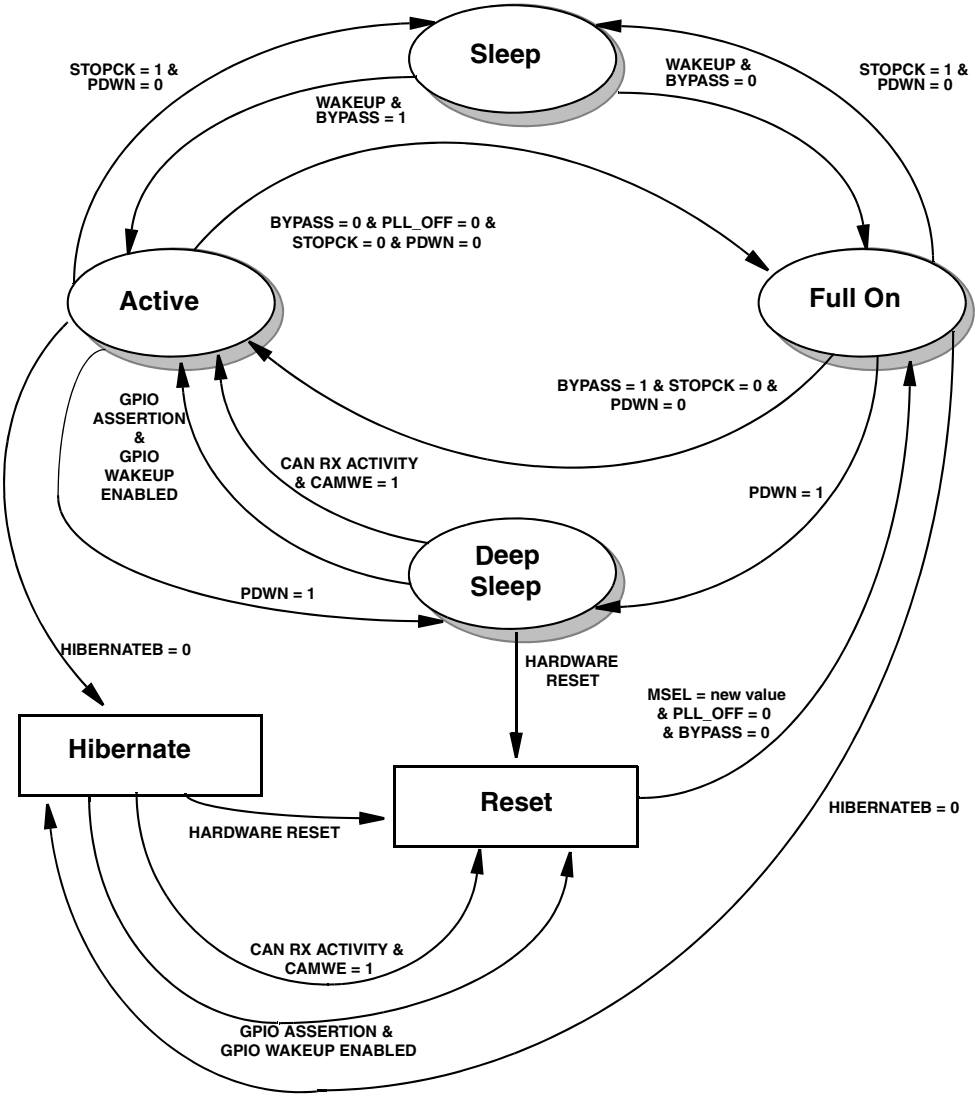


Figure 8-2. Operating Mode Transitions

In addition to the mode transitions shown in Figure 8-2, the PLL can be modified while in active operating mode. Changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect (see "Programming Operating Mode Transitions" on page 8-15).

- PLL disabled: In addition to being bypassed in the active mode, the PLL can be disabled.

  When the PLL is disabled, additional power savings are achieved although they are relatively small. To disable the PLL, set the `PLL_OFF` bit in the `PLL_CTL` register, and then execute the PLL programming sequence.

- PLL enabled: When the PLL is disabled, it can be re-enabled later when additional performance is required.

  The PLL must be re-enabled before transitioning to the full-on or sleep operating modes. To re-enable the PLL, clear the `PLL_OFF` bit in the `PLL_CTL` register, and then execute the PLL programming sequence.

- New multiplier ratio: The multiplier ratio can also be changed while in full-on mode.

  The PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full-on mode. To program a new `CLKIN` to VCO multiplier, write the new `MSEL[5:0]` and/or `DF` values to the `PLL_CTL` register; then execute the PLL programming sequence (on page 8-15).

Table 8-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 8-6 causes unpredictable behavior.

Table 8-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full-On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	Allowed
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control (PLL_CTL) register. Merely modifying the bits of the PLL_CTL register does not change the operating mode or behavior of the PLL. Changes to the PLL_CTL register are realized only after a specific code sequence is executed. This sequence is managed by a user-callable routine in the on-chip ROM called bfrom_SysControl(). When calling this function, no further precautions have to be taken. See "System Control ROM Function" on page 8-24 for more information.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or power is reapplied to the PLL, the PLL needs to relock. To relock, the PLL lock counter is cleared first, then starts incrementing once per SCLK cycle. After the PLL lock counter reaches the value programmed in the PLL lock count (PLL_LOCKCNT) register, the PLL sets the PLL_LOCKED bit in the PLL status (PLL_STAT) register, and the PLL asserts the PLL wake-up interrupt.

When the bfrom_SysControl() routine reprograms the PLL_CTL register with a new value, the bfrom_SysControl() routine executes a subsequent IDLE instruction and prevents all other system interrupt sources, other

than the DPMC, from waking up the core from the IDLE state. If the lock counter expires, the PLL issues an interrupt, and the code execution continues the instruction after the IDLE instruction. Therefore, the system is in the new state by the time the bfrom_SysControl() routine returns.

If the new value written to the PLL_CTL or VR_CTL register is the same as the previous value, the PLL wake-up occurs immediately (PLL is already locked), but the core and system clock are bypassed for the PLL_LOCKCNT duration. For this interval, code executes at the CLKIN rate instead of the expected CCLK rate. Software guards against this condition by comparing the current value to the new value before writing the new value.

- When the wake-up signal is asserted, the code execution continues the instruction after the IDLE instruction, causing a transition to:
    - Active mode if BYPASS in the PLL_CTL register is set
    - Full-on mode if the BYPASS bit is cleared
- If the PLL_CTL register is programmed to enter the sleep operating mode, the processor transitions immediately to sleep mode and waits for a wake-up signal before continuing code execution. If the PLL_CTL register is programmed to enter the deep sleep operating mode, the processor immediately transitions to deep sleep mode and waits for a hardware reset signal, GPIO wakeup, or CAN wakeup:
    - A hardware reset causes the processor to execute the reset sequence. For more information, see “System Reset and Booting” on page 24-1.
    - A GPIO or CAN wakeup event causes the processor to enter active operating mode and return from the bfrom_SysControl() routine.

If no operating mode transition is programmed, the PLL generates a wake-up signal, and the `bfrom_SysControl()` routine returns.

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor's core is capable of running at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses multiple power domains. Each power domain has a separate V_{DD} supply. Note that the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See the product data sheet for details on the allowed voltage ranges for each power domain and power dissipation data.

Power Supply Management

V_{DDINT} is supplied by an external regulator and pin $\overline{PG}$ is used to accept an active-low power-good indicator from the regulator. Note that the external regulator must comply with the V_{DDINT} specifications defined in the processor data sheet.

Changing Voltage

When changing the voltage using an external regulator, a specific programming sequence must be followed.

Unlike other Blackfin derivatives that feature an internal voltage regulator; the voltage level for the ADSP-BF50x cannot be changed by programming the `VR_CTL` register. With an internal voltage regulator, the PLL would automatically enter the active mode when the processor enters the `IDLE` state. At that point the voltage level would change and the PLL would re-lock to the new voltage. After the `PLL_LOCKCNT` has expired, the part returns to the full-on state.

With an external voltage regulator, this sequence must be reproduced in the program code by the user. The PLL_LOCKCNT register cannot be used in this case, but the value is still needed for calculating the required delay. A larger PLL_LOCKCNT value may be necessary for changing voltages than when changing just the PLL frequency. See the processor data sheet for details.

The processor must enter active mode before the user can access the external voltage regulator and program a new voltage level. See the data sheet of external voltage regulator for information on changing voltage levels. See the processor data sheet for more information about voltage tolerances and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior the recommended procedure is to bring the processor to the sleep operating mode before substantially varying the voltage.

The user must ensure a stable voltage and give the PLL time to re-lock at the new voltage level. This can be done by running the core in a loop for a certain amount of time before leaving active mode.

After the voltage has been changed to the new level, the processor can safely return to any operational mode—so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the processor data sheet for the new operating voltage level.

Please see "Changing Voltage Levels" on page 8-40 for more details on mode transitions and changing voltage levels.

The VSTAT bit in the PLL_STAT register can be used to indicate whether V_{DDINT} is stable and ready to use. The VSTAT bit works in conjunction with the $\overline{PG}$ (Power Good) input signal of the ADSP-BF50x. The inverted version of a "power good" signal from the external regulator is fed to the ADSP-BF50x to indicate that the voltage has reached its programmed

value. That in turn will set the VSAT bit, which should be considered the end of your “wait” state for the voltage regulator to settle.

Powering Down the Core (Hibernate State)

The external regulator can be signaled to shut off V_{DDINT} using the EXT_WAKE signal. Writing 0 to the HIBERNATEB bit of the VR_CTL register, which disables CCLK and SCLK, will also make EXT_WAKE go low. EXT_WAKE will transition high if any wakeup sources occur, which will signal the external voltage regulator to turn V_{DDINT} on again. The wakeup sources are several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the $\overline{\text{RESET}}$ pin always exits hibernate state and requires no modification to VR_CTL.
- External GPIO event. Set a GPIO wakeup enable control bit (PH0WE, PF8WE, PF9WE) to enable wakeup on assertion of a signal on the corresponding pin.
- External CAN RX event. Set the CAN RX wakeup enable control (CANWE) bit to enable wakeup on the occurrence of a CAN RX event.
- Pin EXT_WAKE is provided to indicate the occurrence of wakeup. EXT_WAKE is an output pin, which is a logical OR of the above wakeup sources, except hardware reset. The pin follows the wakeup signal of the various wakeup sources.

When the core is powered down, V_{DDINT} is set to 0 V, and the internal state of the processor is not maintained, with the exception of the VR_CTL register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a three-state level unless specified otherwise.

To signal the external regulator to power down V_{DDINT}:

1. Write 0 to the appropriate bits in the SIC_IWRx registers to prevent enabled peripheral resources from interrupting the hibernate process.
2. Call the bfrom_SysControl() routine; ensure that the HIBERNATEB bit in the VR_CTL register is set to 0, and the appropriate wakeup enable bit or bits (PH0WE, PF8WE, PF9WE, or CANWE) are set to 1.
3. The bfrom_SysControl() routine executes until V_{DDINT} transitions to 0 V. The bfrom_SysControl() routine never returns.
4. When the processor is woken up, the PLL relocks and the boot sequence defined by the BMODE[2:0] pin settings takes effect.

The WURESET bit in the SYSCTRL register is set and stays set until the next hardware reset. The WURESET bit may control a conditional boot process.

PLL and VR Registers

The user interface to the PLL and VR registers is through the system control ROM function (bfrom_SysControl()) described in "System Control ROM Function" on page 8-24. The memory-mapped registers (MMRs) are shown in Table 8-7 and illustrated in Figure 8-3 through Figure 8-7.

Table 8-7 shows the functions of the PLL/VR registers.

Table 8-7. PLL/VR Register Mapping

Register Name	Function	Notes	For More Information See:
PLL_CTL	PLL control register	Requires reprogramming sequence when written	Figure 8-4 on page 8-22
PLL_DIV	PLL divisor register	Can be written freely	Figure 8-3 on page 8-21
PLL_STAT	PLL status register	Monitors active modes of operation	Figure 8-5 on page 8-22
PLL_LOCKCNT	PLL lock count register	Number of SCLKs allowed for PLL to relock	Figure 8-6 on page 8-23
VR_CTL	Voltage regulator control register	Requires PLL reprogramming sequence when written	Figure 8-7 on page 8-23

PLL_DIV Register

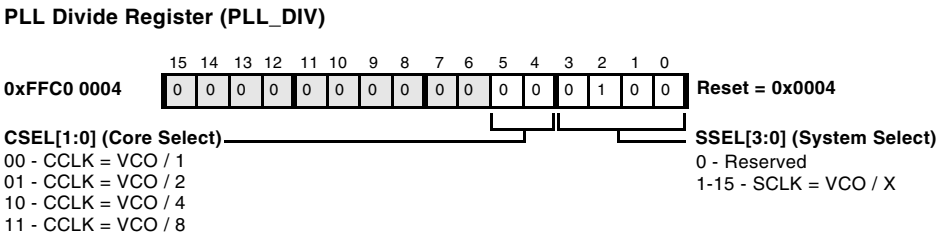
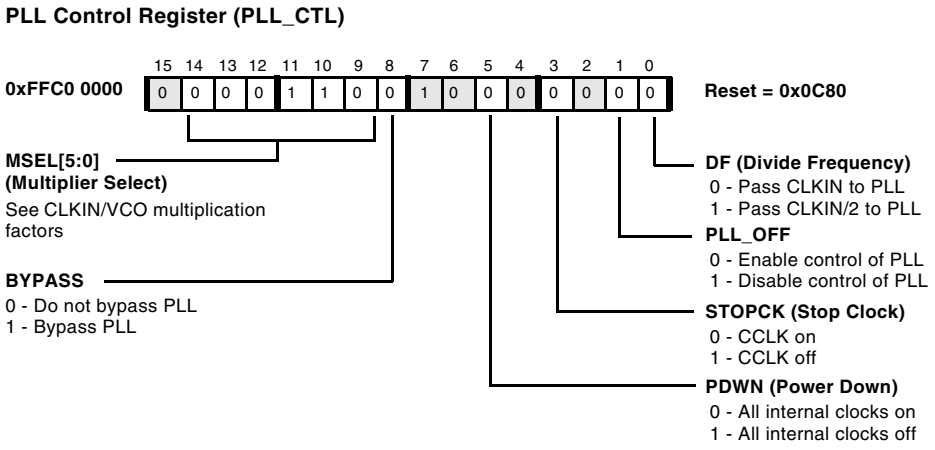


Figure 8-3. PLL Divide Register

PLL_CTL Register



For CLKIN/VCO multiplication factors, see Table 8-1 on page 8-4.

Figure 8-4. PLL Control Register

PLL_STAT Register

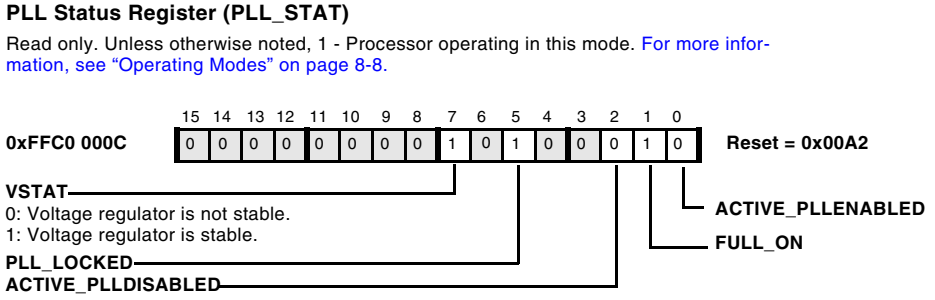


Figure 8-5. PLL Status Register

PLL_LOCKCNT Register

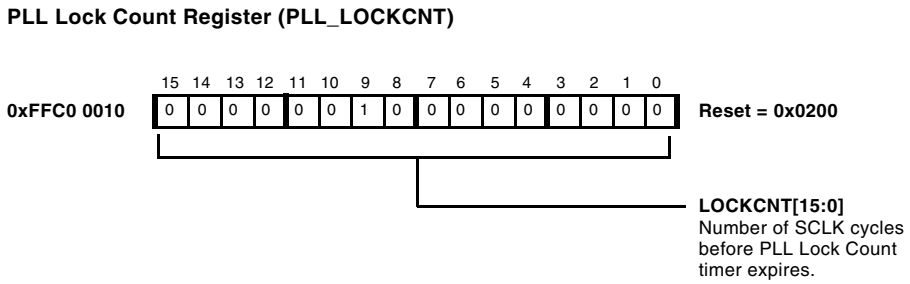


Figure 8-6. PLL Lock Count Register

VR_CTL Register

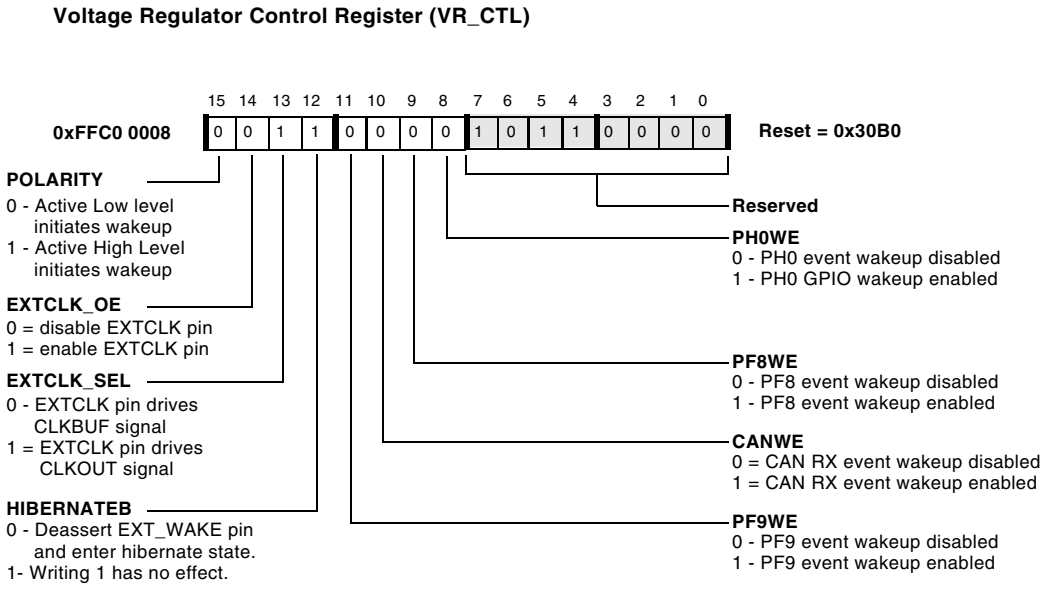


Figure 8-7. Voltage Regulator Control Register

The external clock select (EXTCLK_SEL) control bit configures the EXTCLK pin to output either the SCLK frequency (called CLKOUT) or to output an input buffered CLKIN frequency (called CLKBUF). When configured to output SCLK (CLKOUT), the EXTCLK pin acts as a reference signal in many timing specifications. While CLKOUT is active by default, it can be disabled using the EBIU_AMGCTL register. When configured to output CLKIN (CLKBUF), the EXTCLK pin allows another device *and* the Blackfin processor to run from a single crystal oscillator.

The external clock output enable (EXTCLK_OE) control bit configures the EXTCLK pin to *either* enable (when set, =1) *or* disable (when cleared, =0) the output of the clock signal selected by EXTCLK_SEL. When EXTCLK_OE is cleared, the EXTCLK pin is three-stated.

The POLARITY control bit configure the active level of the wakeup event on the programmable flags. Note that the CAN RX wakeup event is always active low and is *not* affected by the POLARITY bit.

System Control ROM Function

The PLL and voltage regulator registers should not be accessed directly. Instead, use the bfrom_SysControl() function to alter or read the register values. The function resides in the on-chip ROM and can be called by the user following C-language style calling conventions.

Entry address: 0xEF00 0038

Arguments:

- dActionFlags word in R0
- pSysCtrlSettings pointer in R1
- zero value in R2

The system control ROM function does not verify the correctness of the forwarded arguments. Therefore, it is up to the programmer to choose the correct values.

C prototype: `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);`

The first argument (`u32 dActionFlags`) to the system control ROM function holds the instruction flags. The following flags are supported.

```
#define SYSCTRL_READ          0x00000000
#define SYSCTRL_WRITE         0x00000001
#define SYSCTRL_SYSRESET      0x00000002
#define SYSCTRL_SOFTRESET     0x00000004
#define SYSCTRL_VRCTL         0x00000010
#define SYSCTRL_EXTVOLTAGE    0x00000020
#define SYSCTRL_PLLCTL        0x00000100
#define SYSCTRL_PLLDIV        0x00000200
#define SYSCTRL_LOCKCNT       0x00000400
#define SYSCTRL_PLLSTAT       0x00000800
```

With `SYSCTRL_READ` and `SYSCTRL_WRITE`, a read or a write operation is initialized. The `SYSCTRL_SYSRESET` flag performs a system reset, while the `SYSCTRL_SOFTRESET` flag combines a core and system reset. The `SYSCTRL_EXTVOLTAGE` flag indicates that V_{DDINT} is supplied externally. Five of the flags (`_VRCTL`, `_PLLCTL`, `_PLLDIV`, `_LOCKCNT`, `_PLLSTAT`) tell the system control ROM function which registers to be written to or read from. Note that `SYSCTRL_PLLSTAT` flag is read-only.

The second argument (`ADI_SYSCTRL_VALUES *pSysCtrlSettings`) to the system control ROM function passes a pointer to a special structure, which has entries for all PLL and voltage regulator registers. It is predefined in the `bfrom.h` header file as follows.

```
typedef struct
{
   u16 uwVrCtl;
   u16 uwPllCtl;
   u16 uwPllDiv;
   u16 uwPllLockCnt;
   u16 uwPllStat;
} ADI_SYSCTRL_VALUES;
```

The third argument to the system control ROM function is reserved and should be kept zero (NULL pointer).

The system control ROM function executes the correct steps and programming sequence for the Dynamic Power Management System of the Blackfin processor.

Programming Model

The programming model for the system control ROM function in C/C++ and Assembly is described in the following sections.

Accessing the System Control ROM Function in C/C++

To read the `PLL_DIV` and `PLL_CTL` register values, for example, specify the `SYSCTRL_READ` instruction flag along with the `SYSCTRL_PLLCTL` and `SYSCTRL_PLLDIV` register flags:

```
ADI_SYSCTRL_VALUES read;
bfrom_SysControl (SYSCTRL_READ | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV,
&read, NULL);
```

The `read.uwPllCtl` and `read.uwPllDiv` variables access the `PLL_CTL` and `PLL_DIV` register values, respectively. To update the register values, specify the `SYSCTRL_WRITE` instruction flag along with the register flags of those

registers that should be modified and have valid data in the respective ADI_SYSCTRL_VALUES variables:

```
ADI_SYSCTRL_VALUES write;
write.uwPllCtl = 0x1480;
write.uwPllDiv = 0x0004;
bfrom_SysControl (SYSCTRL_WRITE | SYSCTRL_PLLCTL |SYSCTRL_PLLDIV,
&write, NULL);
```

Accessing the System Control ROM Function in Assembly

The assembler supports C structs, which is required to import the file bfrom.h:

```
#include <bfrom.h>
.IMPORT "bfrom.h";
.STRUCT ADI_SYSCTRL_VALUES dpm;
```

You can pre-load the struct:

```
.STRUCT ADI_SYSCTRL_VALUES dpm = { 0x70B0, 0x1480, 0x0004,
0x0200, 0x00A2 };
```

or load the values dynamically inside the code:

```
P5.H = hi(dpm);

P5.L = lo(dpm->uwVrCtl);
R7 = 0x70B0 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllCtl);
R7 = 0x1480 (z);
w[P5] = R7;
```

```
P5.L = lo(dpm->uwPllDiv);
R7 = 0x0004 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllLockCnt);
R7 = 0x0200 (z);
w[P5] = R0;
```

The function `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);` can be accessed by `BFROM_SYSCONTROL`. Following the C/C++ run-time environment conventions, the parameters passed are hold by the data registers `R0`, `R1`, and `R2`.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;

R0 = SYSCTRL_WRITE      |
     SYSCTRL_VRCTL      |
     SYSCTRL_EXTVOLTAGE |
     SYSCTRL_PLLCTL     |
     SYSCTRL_PLLDIV     ;
R1.H = hi(dpm);
R1.L = lo(dpm);
R2 = 0 (z);
P5.H = hi(BFROM_SYSCONTROL);
```

```
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

The processor's internal scratchpad memory can be used as an alternative for taking a C struct. Therefore, the stack/frame pointer must be loaded and passed.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;

R7 = 0;
R7.L = 0x70B0;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R7;
R7.L = 0x1480;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R7;
R7.L = 0x0004;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R7;
R7.L = 0x0200;
```

```
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R7;

R0 = SYSCTRL_WRITE      |
     SYSCTRL_VRCTL      |
     SYSCTRL_EXTVOLTAGE |
     SYSCTRL_PLLCTL     |
     SYSCTRL_PLLDIV     ;
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0;

P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

Programming Examples

The following code examples illustrate how to use the system control ROM function to effect various operating mode transitions.

The following examples are only meant to demonstrate how to program the PLL registers. Do not assume that the voltages and frequencies shown in the examples are supported by your processor. Instead, check your product's data sheet for supported voltages and frequencies.

Some setup code has been removed for clarity, and the following assumptions are made.

- PLL control (PLL_CTL) register setting: 0x0A80
- PLL divider (PLL_DIV) register setting: 0x0004
- PLL lock count (PLL_LOCKCNT) register setting: 0x0200
- Clock in (CLKIN) frequency: 25 MHz

VCO frequency is 125 MHz, core clock frequency is 125 MHz, and system clock frequency is 31.25 MHz.

- Voltage regulator control (VR_CTL) register setting: 0x70B0
- Logical voltage level (VDDINT) is at 1.20 V

For operating mode transition and voltage regulator examples:

- **C**
    - `#include  <blackfin.h>`
    - `#include <bfrom.h>`
- **Assembly**
    - `#include  <blackfin.h>`
    - `#include <bfrom.h>`
    - `.IMPORT "bfrom.h";`
    - `#define IMM32(reg,val) reg##.H=hi(val);`
    - `reg##.L=lo(val);`

Full-on Mode to Active Mode and Back

Listing 8-1 and Listing 8-2 provide code for transitioning from the full-on operating mode to active mode in C and Blackfin assembly code, respectively.

Listing 8-1. Transitioning from Full-on Mode to Active Mode (C)

```
void active(void)
{
ADI_SYSCTRL_VALUES active;
bfrom_SysControl(SYSCTRL_READ | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);
active.uwPllCtl |= (BYPASS | PLL_OFF); /* PLL_OFF bit optional */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);
return;
}
```

Listing 8-2. Transitioning from Full-on Mode to Active Mode (ASM)

```
__active:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = (SYSCTRL_READ | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(BYPASS));
bitset(R0,bitpos(PLL_OFF)); /* optional */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__active.end:
```

To return from active mode (go back to full-on mode), the `BYPASS` bit and the `PLL_OFF` bit must be cleared again, respectively.

Transition to Sleep Mode or Deep Sleep Mode

Listing 8-3 and Listing 8-4 provide code for transitioning from the full-on operating mode to sleep or deep sleep mode in C and Blackfin assembly code, respectively.

Listing 8-3. Transitioning to Sleep Mode or Deep Sleep Mode (C)

```
void sleep(void)
{
ADI_SYSCTRL_VALUES sleep;
bfrom_SysControl(SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &sleep, NULL);
sleep.uwPllCtl |= STOPCK;    /* either: Sleep Mode */
sleep.uwPllCtl |= PDWN;      /* or: Deep Sleep Mode */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &sleep, NULL);
return;
}
```

Listing 8-4. Transitioning to Sleep Mode or Deep Sleep Mode (ASM)

```
__sleep:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = (SYSCTRL_READ | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(STOPCK)); /* either: Sleep Mode */
bitset(R0,bitpos(PDWN)); /* or: Deep Sleep Mode */
```

```
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__sleep.end:
```

Set Wakeup Events and Enter Hibernate State

Listing 8-5 and Listing 8-6 provide code for configuring the regulator wakeups (PH0, PF8, PF9, and CAN_RX) and placing the regulator in the hibernate state in C and Blackfin processor assembly code, respectively.

Listing 8-5. Configuring Regulator Wakeups and Entering Hibernate State (C)

```
void hibernate(void)
{
ADI_SYSCTRL_VALUES hibernate;
hibernate.uwVrCtl=WAKE_EN0 | /* PH0 Wake-Up Enable */
WAKE_EN1 | /* PF8 Wake-Up Enable */
WAKE_EN2 | /* PF9 Wake-Up Enable */
CANWE | /* CAN Rx Wake-Up Enable */
HIBERNATE; / *Powerdown */
```

```
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_VRCTL |
SYSCTRL_EXTVOLTAGE, &hibernate, NULL);
/* Hibernate State: no code executes until wakeup triggers
reset */
}
```

Listing 8-6. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```
__hibernate:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
cli R6; /* disable interrupts, copy IMASK to R6 */
R0.L = WAKE_EN0 | /* PH0 Wake-Up Enable */
WAKE_EN1 | /* PF8 Wake-Up Enable */
WAKE_EN2 | /* PF9 Wake-Up Enable */
CANWE | /* CAN Rx Wake-Up Enable */
HIBERNATE; / *Powerdown */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_VRCTL | SYSCTRL_EXTVOLTAGE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
/* Hibernate State: no code executes until wakeup triggers
reset */
__hibernate.end:
```

Note that there may be a need to call `bfrom_SysControl()` twice, once to setup the polarity and the wakeup sources and once to enter hibernate.

Perform a System Reset or Soft-Reset

Listing 8-7 and Listing 8-8 provide code for executing a system reset *or* a soft-reset (system and core reset) in C and Blackfin assembly code, respectively.

Listing 8-7. Execute a System Reset or a Soft-Reset (C)

```
void reset(void)
{
bfrom_SysControl(SYSCTRL_SYSRESET, NULL, NULL); /* either */
bfrom_SysControl(SYSCTRL_SOFTRESET, NULL, NULL); /* or */
return;
}
```

Listing 8-8. Execute a System Reset or a Soft-Reset (ASM)

```
__reset:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = SYSCTRL_SYSRESET; /* either */
R0 = SYSCTRL_SOFTRESET; /* or */
R1 = 0 (z);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

```
__reset.end:
```

In Full-on Mode, Change VCO Frequency, Core Clock Frequency, and System Clock Frequency

Listing 8-9 and Listing 8-10 provide C and Blackfin assembly code for changing the CLKIN to VCO multiplier (from 10x to 21x), keeping the CSEL divider at 1, and changing the SSEL divider (from 5 to 4) in the full-on operating mode.

Listing 8-9. Transition of Frequencies (C)

```
void frequency(void)
{
ADI_SYSCTRL_VALUES frequency;

/* Set MSEL = 5-63 --> VCO = CLKIN*MSEL */
frequency.uwPllCtl = SET_MSEL(21) ;

/* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
/* CCLK = VCO / 1 */
frequency.uwPllDiv = SET_SSEL(4) |
                     CSEL_DIV1   ;

frequency.uwPllLockCnt = 0x0200;

bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL | SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT |, &frequency,
NULL);
return;
}
```

Listing 8-10. Transition of Frequencies (ASM)

```
__frequency:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

/* write the struct */
R0 = 0;

R0.L = SET_MSEL(21) ; /* Set MSEL = 5-63 --> VCO = CLKIN*MSEL */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0.L = SET_SSEL(4) | /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
       CSEL_DIV1   ; /* CCLK = VCO / 1 */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;

R0.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;

/* argument 1 in R0 */
R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV);

/* argument 2 in R1: structure lays on local stack */
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);

/* argument 3 must always be NULL */
R2 = 0;
```

```
/* call of SysControl function */
IMM32(P4,BFROM_SYSCONTROL);
call (P4); /* R0 contains the result from SysControl */

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__frequency.end:
```

Changing Voltage Levels

Listing 8-11 provides C code for changing the voltage level dynamically. The User must include his own code for accessing the external voltage regulator.

Listing 8-11. Changing Core Voltage (C)

```
void voltage(void)
{
ADI_SYSCTRL_VALUES voltage;
u32 ulCnt = 0;
bfrom_SysControl( SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &init, NULL );
init.uwPllCtl |= BYPASS;
init.uwPllLockCnt = 0x0200;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL | SYSCTRL_LOCKCNT
| SYSCTRL_EXTVOLTAGE, &voltage, NULL);
/* Put your code for accessing the external voltage regulator
here */
```

```
/* A delay loop is required to ensure VDDint is stable and the
PLL has re-locked. As this is depending on the external voltage
regulator circuitry the user must ensure timings are kept. The
compiler (no optimization enabled) will create a loop that takes
about 10 cycles. Time base is CLKIN as the PLL is bypassed. We
need 0x0200 CLKIN cycles that represent PLL_LOCKCNT and addition-
ally the time required by the circuitry */
ulCnt = 0x0200 + 0x0200;
while (ulCnt != 0) {ulCnt--;}
init.uwPllCtl &= ~BYPASS;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL |
SYSCTRL_EXTVOLTAGE, &voltage, NULL);
return;
}
```

9 GENERAL-PURPOSE PORTS

This chapter describes the general-purpose ports. Following an overview and a list of key features is a block diagram of the interface and a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The ADSP-BF50x Blackfin processors feature a rich set of peripherals, which, through a powerful pin multiplexing scheme, provides great flexibility to the external application space.

Features

The peripheral pins are functionally organized into general-purpose ports designated port F, port G, and port H.

Port F provides 16 pins:

- PPI data signals
- UART0 and UART1 signals
- SPORT0 signals
- SPI0 signals
- CNT0 (GP Counter 0) signals

- PWM0 signals
- GP Timer signals
- Additional SPI0 and SPI1 slave selects
- GPIOs

Port G provides 16 pins:

- SPORT1 signals
- CAN signals
- ACM signals
- PPI signals
- GP Timer signals
- PWM1 signals
- CNT1 (GP Counter 1) signals
- SPI1 signals
- UART0 signals
- RSI signals
- GPIOs

Port H provides 3 pins:

- SPORT1 signals
- ACM signals
- SPI0 and SPI1 slave select signals

- GP Timer signals
- GPIOs

Additionally, the TWI signals are provided on separate pins, independent of the ports.

Interface Overview

By default, all port F, port G, and port H pins are in general-purpose I/O (GPIO) mode. In this mode, a pin can function as a digital input, digital output, or interrupt input. See "General-Purpose I/O Modules" on page 9-13 for details. Peripheral functionality must be explicitly enabled by the function enable registers (PORTF_FER, PORTG_FER, and PORTH_FER). The competing peripherals on port F, port G, and port H are controlled by the respective multiplexer control register (PORTF_MUX, PORTG_MUX, PORTH_MUX).

In this chapter, the naming convention for registers and bits uses a lowercase x to represent F, G, or H. For example, the name PORTx_FER represents PORTF_FER, PORTG_FER, and PORTH_FER. The bit name PxO represents PF0, PG0, and PH0. This convention is used to discuss registers common to these three ports.

External Interface

The external interface of the general-purpose ports are described in the following sections.

Port F Structure

Table 9-1 shows the multiplexer scheme for port F. Port F is controlled by the PORTF_MUX and the PORTF_FER registers.

Port F consists of 16 pins, referred to as PF0 to PF15, as shown in Table 9-1. All the input signals in the "Additional Use" column are enabled by their module only, regardless of the state of the PORTx_MUX and PORTx_FER registers.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in PORTF_FER is cleared.

Table 9-1. Port F Multiplexing Scheme

PORTF_MUX	00	01	10		
	1st Function	2nd Function	3rd Function	Additional Use	GPIO
Bit[1:0]	TSCLK0	UA0_RX	TMR6	RW0_CUD	PF0
Bit[3:2]	RSCLK0	UA0_TX	TMR5	RW0_CGD	PF1
Bit[5:4]	DT0PRI	PWM0_BH	PPI_DATA8	RW0_CZM	PF2
	TFS0	PWM0_BL	PPI_DATA9	RW0_CGD	PF3
	RFS0	PWM0_CH	PPI_DATA10	TACLK0	PF4
	DR0PRI	PWM0_CL	PPI_DATA11	TACLK1	PF5
Bit[7:6]	UA1_TX	$\overline{\text{PWM0\_TRIP}}$	PPI_DATA12		PF6
	UA1_RX	PWM0_SYNC	PPI_DATA13	TACI3	PF7
Bit[9:8]	$\overline{\text{UA1\_RTS}}$	DT0SEC	PPI_DATA7		PF8
	$\overline{\text{UA1\_CTS}}$	DR0SEC	PPI_DATA6	RW0_CZM/WAKEUP	PF9
Bit[11:10]	SPI0_SCK	TMR2	PPI_DATA5		PF10
	SPI0_MISO	$\overline{\text{PWM0\_TRIP}}$	PPI_DATA4	TACLK2	PF11
	SPI0_MOSI	PWM0_SYNC	PPI_DATA3		PF12
Bit[13:12]	$\overline{\text{SPI0\_SSEL1}}$	TMR3	PPI_DATA2	$\overline{\text{SPI0\_SS}}$	PF13
Bit[15:14]	$\overline{\text{SPI0\_SSEL2}}$	PWM0_AH	PPI_DATA1		PF14
	$\overline{\text{SPI0\_SSEL3}}$	PWM0_AL	PPI_DATA0		PF15

Port G Structure

Table 9-2 shows the multiplexer scheme for port G. Port G is controlled by the PORTG_MUX and PORTG_FER registers.

Port G consists of 16 pins, referred to as PG0 to PG15, as shown in Table 9-2.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTG_FER register is cleared.

Table 9-2. Port G Multiplexing Scheme

PORTG_MUX	00	01	10		
	1st Function	2nd Function	3rd Function	Additional Use	GPIO
Bit[1:0]	SPI1_SSEL3	TMRCLK/PPICLK	UA1_RX	TACI4	PG0
Bit[3:2]	SPI1_SSEL2	PPI_FS3	CAN_RX	TACI5/WAKEUP	PG1
Bit[5:4]	SPI1_SSEL1	TMR4	CAN_TX	SPI1_SS	PG2
Bit[7:6]	SPI1_SCK	DT1SEC	UA1_TX		PG3/HWAIT
	SPI1_MOSI	DR1SEC	PWM1_SYNC	TACLK6	PG4
	SPI1_MISO	TMR7	PWM1_TRIP		PG5
Bit[9:8]	ACM_SE_DIFF	SD_DATA3	PWM1_AH		PG6
	ACM_RANGE	SD_DATA2	PWM1_AL		PG7
	DR1SEC	SD_DATA1	PWM1_BH		PG8
Bit[11:10]	DR1PRI	SD_DATA0	PWM1_BL		PG9
	RFS1	SD_CMD	PWM1_CH	TACI6	PG10
	RSCLK1	SD_CLK	PWM1_CL	TACLK7	PG11
Bit[13:12]	UA0_RX	SD_DATA4	PPI_DATA15	TACI2	PG12
	UA0_TX	SD_DATA5	PPI_DATA14	RW1_CZM	PG13
Bit[15:14]	UA0_RTS	SD_DATA6	TMR0/PPIFS1	RW1_CUD	PG14
	UA0_CTS	SD_DATA7	TMR1/PPIFS2	RW1_CGD	PG15

Port H Structure

Table 9-3 shows the multiplexer scheme for port H. Port H is controlled by the PORTH_MUX and PORTH_FER registers.

Port H consists of 3 pins. PH0 to PH2 (shown in Table 9-3) are GPIO capable and operate in the same fashion as the Port F and Port G pins.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTH_FER register is cleared.

Table 9-3. Port H Multiplexing Scheme

PORTH_MUX	00	01	10		
	1st function	2nd function	3rd function	Additional Use	GPIO
Bit[1:0]	ACM_A2	DT1PRI	$\overline{\text{SPI0\_SSEL3}}$	WAKEUP	PH0
Bit[3:2]	ACM_A1	TFS1	$\overline{\text{SPI1\_SSEL3}}$	TACLK3	PH1
Bit[5:4]	ACM_A0	TSCLK1	$\overline{\text{SPI1\_SSEL2}}$	TACI7	PH2

Bits 6-15 in the PORTH_MUX register are reserved.

Input Tap Considerations

Input taps are shown in Table 9-1, Table 9-2, and Table 9-3 under the “Additional Use” column.When input taps (as well as GPIO based taps) are used with other functionality enabled on the GPIO pins, the signals seen by the input tap modules might be different from what is seen on the pins. This is because different pin functions have different signal requirements with respect to when the signal is latched, if at all. Because of this, input taps multiplexed on certain pins may behave differently than those

on other pins, depending on which pin function is selected. The input taps will see different signals than at the pins in the following cases:

- All GPIO inputs except PF0, PF1, PG0, PG2, PG6, PG7, PG11, and PH2 when GPIO is tapped with the respective PORTx_FER set to 1.
- CZM0 if PORTF_FER[2] = 1 and PORTF_MUX[5:4] = b#00 or b#10
- CDG0 if PORTF_FER[3]= 1 and PORTF_MUX[5:4] = b#00 or b#10
- TACLK0 if PORTF_FER[4] = 1 and PORTF_MUX[5:4] = b#00 or b#10
- TACLK1 if PORTF_FER[5] = 1 and PORTF_MUX[5:4] = b#00 or b#10
- TACI3 if PORTF_FER[7] = 1 and PORTF_MUX[7:6] = b#10
- CZM0 if PORTF_FER[9] = 1 and PORTF_MUX[9:8] = b#01 or b#10
- TACLK2 if PORTF_FER[11] = 1 and PORTF_MUX[11:10] = b#00 or b#10
- SPI0_SS if PORTF_FER[13] = 1 and PORTF_MUX[13:12] = b#10
- TACI5 if PORTG_FER[1] = 1 and PORTG_MUX[3:2] = b#01
- TACLK6 if PORTG_FER[4] = 1 and PORTG_MUX[7:6] = b#00 or b#01
- TACI6 if PORTG_FER[10] = 1 and PORTG_MUX[11:10] = b#00
- TACI2 if PORTG_FER[12] = 1 and PORTG_MUX[13:12] = b#10
- CZM1 if PORTG_FER[13] = 1 and PORTG_MUX[13:12] = b#10
- CUD1 if PORTG_FER[14] = 1 and PORTG_MUX[15:14] = b#10
- CDG1 if PORTG_FER[15] = 1 and PORTG_MUX[15:14] = b#10
- TACLK3 if PORTH_FER[1] = 1 and PORTH_MUX[3:2] = b#01

PWM Unit Considerations

The PWM0_SYNC signal appears twice within Port F: on PF7 and PF12. If both are configured as PWM0_SYNC and selected, inputs will only be enabled on PF7.

The $\overline{\text{PWM0\_TRIP}}$ signal appears twice within Port F: on PF6 and PF11. If both are configured as $\overline{\text{PWM0\_TRIP}}$ and selected, inputs will only be enabled on PF6.

If $\overline{\text{PWM0\_TRIP}}$ is not selected on either PF6 or PF11, then the internal $\overline{\text{PWM0\_TRIP}}$ signal to the PWM module will be driven low. That is, the PWM unit will be tripped if neither of these signals is selected via the PORTF_MUX register.

The same principle holds true for the $\overline{\text{PWM1\_TRIP}}$ signal on PG5, in the PORTG_MUX register.

RSI Considerations

Pull up/pull down enabling for RSI:

- Pull down for SD_DATA[3] will be enabled only if SD_DATA[3] is selected on PG6 (that is, PORTG_MUX[9:8] == b#01) and the PD_Dat3 bit is set in the RSI_CONFIG register.
- Pull up for SD_DATA[3] will be enabled only if SD_DATA[3] is selected on PG6 (that is, PORTG_MUX[9:8] == b#01) and the PU_Dat3 bit is set in the RSI_CONFIG register.
- Pull up for SD_DATA[0] will be enabled only if SD_DATA[0] is selected on PG9 (that is, PORTG_MUX[11:10] == b#01) and the PU_Dat bit is set in the RSI_CONFIG register.
- Pull up for SD_DATA[1] will be enabled only if SD_DATA[1] is selected on PG8 (that is, PORTG_MUX[9:8] == b#01) and the PU_Dat bit is set in the RSI_CONFIG register.

- Pull up for SD_DATA[2] will be enabled only if RSI is selected on PG7 (that is, PORTG_MUX[9:8] == b#01) and the PU_Dat bit is set in the RSI_CONFIG register.
- Pull up for SD_DATA[7:4] will be enabled only if SD_DATA[7:4] is selected on PG[15:12] (that is, PORTG_MUX[13:12]==b#01 and PORTG_MUX[15:14]==b#01)) and the PU_Dat bit is set in the RSI_CONFIG register.

If SD_DATA[3] is not selected on PG6 (that is, PG_MUX[9:8] ≠ b#01) then the SD_DATA[3] signal to the RSI module will be driven low. This is to prevent a spurious card detect interrupt generated by the RSI due to data toggling on the PG6 pin when it is selected for alternate function operation.

GP Counter Considerations

If SPORT0 TX operation is not enabled, RW0 is an input tap on pins PF0, PF2, and PF3. Otherwise, RW0 is an input tap on PF0, PF1, and PF9.

SPI Considerations

If SPI0 or SPI1 is operating in master mode *and* the PSSE bit in the respective SPI_CTL register is set to 1, the $\overline{\text{SPIx\_SSEL1}}$ signal for that SPI interface can *not* be used as a slave select line. This restriction occurs because (in master mode) the $\overline{\text{SPIx\_SS}}$ input tap becomes an error detection input when PSSE=1.

Internal Interfaces

Port control and GPIO registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO module MMRs appear in "System MMR Assignments" on page A-1. Core access to the GPIO configuration registers is through the system bus.

The PORTx_MUX registers control the muxing schemes of port F, port G, and port H.

The function enable registers (PORTF_FER, PORTG_FER, PORTH_FER) enable the peripheral functionality for each individual pin of a port.

GP Timer Interaction With Other Blocks

The TACLKx and TACIx inputs of the GP Timers connect to several different subsystems of the ADSP-BF50x processor. Following are the details of these connections.

Buffered CLKIN (CLKBUF)

TACLK5 and TACLK4 connect internally to the CLKBUF signal.

GP Counter

TACI0 connects to the COUNTER0 TO output internally. TACI1 connects to the COUNTER1 TO output internally.

PPI

TMR0 is internally looped back to PPI_FS1 (to be used as internally generated frame sync). In this case, PPI_CLK is the clock input for the Timer0 module.

TMR1 is internally looped back to PPI_FS2 (to be used as internally generated frame sync) In this case, PPI_CLK is the clock input for the Timer1 module.

PPI_CLK/TMRCLK can be used as a clock input for any of the timers.

UART

TACI2 or TMR6 can be used for autobaud detection of UA0_RX.

TACI3 or TACI4 can be used for autobaud detection of UA1_RX.

UART0 signals that appear in multiple ports, if selected on both, will have inputs and outputs enabled only on PG12-PG13.

UART1 signals that appear in multiple ports, if selected on both, will have inputs and outputs enabled only on PF6-PF7.

SPORT

If TMR5 is configured as an output and PORTF_MUX[3:2] == b#10 and SPORT0's RSCLK0 input enable is active, then TMR5 is the clock input for RSCLK0.

If TMR6 is configured as an output and PORTF_MUX[1:0] == b#10, and SPORT0's TSCLK0 input enable is active, then TMR6 is the clock input for TSCLK0.

If SPORT0's RSCLK0 is configured as an output and PORTF_MUX[3:2] == b#00 and TMR5 input enable is active, then RSCLK0 is the clock input for TMR5.

If SPORT0's TSCLK0 is configured as an output and PORTF_MUX[1:0] == b#00 and TMR6 input enable is active, then TSCLK0 is the clock input for TMR6.

If TACI7 is selected in the TMR7 module, then the signal from the PH2 pin is fed to both SPORT1's TSCLK1 and TACI7.

If SPORT1's DR1SEC is selected on both PG4 and PG8, it will only be enabled on PG8.

ACM

When the ACM is enabled, TMR2 and TMR7 are internally routed into the ACM block.

Performance/Throughput

The PFx, PGx, and PHx pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of port pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the signal is asserted on the pin and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

Description of Operation

The operation of the general-purpose ports is described in the following sections.

Operation

The GPIO pins on port F, port G, and port H can be controlled individually by the function enable registers (PORTx_FER). With a control bit in these registers cleared, the peripheral function is fully decoupled from the pin. It functions as a GPIO pin only. To drive the pin in GPIO output mode, set the respective direction bit in the PORTxIO_DIR register. To make the pin a digital input or interrupt input, enable its input driver in the PORTxIO_INEN register.

By default all peripheral pins are configured as inputs after reset. port F, port G, and port H pins are in GPIO mode. However, GPIO input drivers are disabled to minimize power consumption and any need of external pulling resistors.

When the control bit in the function enable registers (PORTx_FER) is set, the pin is set to its peripheral functionality and is no longer controlled by the GPIO module. However, the GPIO module can still sense the state of the pin. When using a particular peripheral interface, pins required for the peripheral must be individually enabled. Keep the related function enable bit cleared if a signal provided by the peripheral is not required by your application. This allows it to be used in GPIO mode.

General-Purpose I/O Modules

The processor supports 35 bidirectional or general-purpose I/O (GPIO) signals. These 35 GPIOs are managed by three different GPIO modules, which are functionally identical. One is associated with port F, one with port G, and one with port H. Port F and port G each consist of 16 GPIOs (PF15-0 and PG15-0), respectively. Port H consists of three GPIOs (PH7-0).

Each GPIO can be individually configured as either an input or an output by using the GPIO direction registers (PORTxIO_DIR).

When configured as output, the GPIO data registers (PORTFIO, PORTGIO, and PORTHIO) can be directly written to specify the state of the GPIOs.

The GPIO direction registers are read-write registers with each bit position corresponding to a particular GPIO. A logic 1 configures a GPIO as an output, driving the state contained in the GPIO data register if the peripheral function is not enabled by the function enable registers. A logic 0 configures a GPIO as an input.

Note when using the GPIO as an input, the corresponding bit should also be set in the GPIO input enable register. Otherwise, changes at the input pins will not be recognized by the processor.

The GPIO input enable registers (PORTFIO_INEN, PORTGIO_INEN, and PORTHIO_INEN) are used to enable the input buffers on any GPIO that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx, PGx, or PHx pin is not used in the system. By default, the input buffers are disabled.

Once the input driver of a GPIO pin is enabled, the GPIO is not allowed to operate as an output anymore. Never enable the input driver (by setting PORTxIO_INEN bits) and the output driver (by setting PORTxIO_DIR bits) for the same GPIO.

A write operation to any of the GPIO data registers sets the value of all GPIOs in this port that are configured as outputs. GPIOs configured as inputs ignore the written value. A read operation returns the state of the GPIOs defined as outputs and the sense of the inputs, based on the polarity and sensitivity settings, if their input buffers are enabled. Table 9-4 helps to interpret read values in GPIO mode, based on the settings of the PORTxIO_POLAR, PORTxIO_EDGE, and PORTxIO_BOTH registers.

Table 9-4. GPIO Value Register Pin Interpretation

POLAR	EDGE	BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For GPIOs configured as edge-sensitive, a readback of 1 from one of these registers is sticky. That is, once it is set it remains set until cleared by user code. For level-sensitive GPIOs, the pin state is checked every cycle, so the readback value will change when the original level on the pin changes.

The state of the output is reflected on the associated pin only if the function enable bit in the PORTx_FER register is cleared.

Write operations to the GPIO data registers modify the state of all GPIOs of a port. In cases where only one or a few GPIOs need to be changed, the user may write to the GPIO set registers, PORTxIO_SET, the GPIO clear registers, PORTxIO_CLEAR, or to the GPIO toggle registers, PORTxIO_TOGGLE instead.

While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO set register can be used to set a single or a few bits only. No read-modify-write operations are required. The GPIO set registers are write-1-to-set registers. All 1s contained in the value written to a GPIO set register sets the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF0 is configured as an output. Writing 0x0001 to the GPIO set register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. The GPIO set registers are typically also used to generate GPIO interrupts by software. Read operations from the GPIO set registers return the content of the GPIO data registers.

The GPIO clear registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO clear register can be used to clear individual bits only. No read-modify-write operations are required. The clear registers are write-1-to-clear registers. All 1s contained in the value written to the GPIO clear register clears the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF4 and PF5 are configured as outputs. Writing 0x0030 to the PORTFIO_CLEAR register drives a logic 0 on the PF4 and PF5 pins without affecting the state of any other PFx pins.

If an edge-sensitive pin generates an interrupt request, the service routine must acknowledge the request by clearing the respective GPIO latch. This is usually performed through the clear registers.

Read operations from the GPIO clear registers return the content of the GPIO data registers.

The GPIO toggle registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a toggle register can be used to toggle individual bits. No read-modify-write operations are required. The GPIO toggle registers are write-1-to-toggle registers. All 1s contained in the value written to a GPIO toggle register toggle the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PG1 is configured as an output. Writing 0x0002 to the PORTGIO_TOGGLE register changes the pin state (from logic 0 to logic 1, or from logic 1 to logic 0) on the PG1 pin without affecting the state of any other PGx pins. Read operations from the GPIO toggle registers return the content of the GPIO data registers.

The state of the GPIOs can be read through any of these data, set, clear, or toggle registers. However, the returned value reflects the state of the input pin only if the proper input enable bit in the PORTxIO_INEN register is set. Note that GPIOs can still sense the state of the pin when the function enable bits in the PORTx_FER registers are set.

Since function enable registers and GPIO input enable registers reset to zero, no external pull-ups or pull-downs are required on the unused pins of port F, port G, and port H.

GPIO Interrupt Processing

Each GPIO can be configured to generate an interrupt. The processor can sense up to 35 asynchronous off-chip signals, requesting interrupts through six interrupt channels. To make a pin function as an interrupt pin, the associated input enable bit in the PORTxIO_INEN register must be set. The function enable bit in the PORTx_FER register is typically cleared.

Then, an interrupt request can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high and high to low). Input sensitivity is defined on a per-bit basis by the GPIO polarity registers (PORTFIO_POLAR, PORTGIO_POLAR, and PORTHIO_POLAR), and the GPIO interrupt sensitivity registers (PORTFIO_EDGE, PORTGIO_EDGE, and PORTHIO_EDGE). If configured for edge sensitivity, the GPIO set on both edges registers (PORTFIO_BOTH, PORTGIO_BOTH, and PORTHIO_BOTH) let the interrupt request generate on both edges.

The GPIO polarity registers are used to configure the polarity of the GPIO input source. To select active high or rising edge, set the bits in the GPIO polarity register to 0. To select active low or falling edge, set the bits in the GPIO polarity register to 1. This register has no effect on GPIOs that are defined as outputs. The contents of the GPIO polarity registers are cleared at reset, defaulting to active high polarity.

The GPIO interrupt sensitivity registers are used to configure each of the inputs as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. The GPIO interrupt sensitivity register has no effect on GPIOs that are defined as outputs. The contents of the GPIO interrupt sensitivity registers are cleared at reset, defaulting to level sensitivity.

The GPIO set on both edges registers are used to enable interrupt generation on both rising and falling edges. When a given GPIO has been set to edge-sensitive in the GPIO interrupt sensitivity register, setting the respective bit in the GPIO set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on GPIOs that are defined as level-sensitive or as outputs. See Table 9-4 on page 9-14 for information on how the GPIO set on both edges register interacts with the GPIO polarity and GPIO interrupt sensitivity registers.

When the GPIO's input drivers are enabled while the GPIO direction registers configure it as an output, software can trigger a GPIO interrupt by writing to the data/set/toggle registers. The interrupt service routine should clear the GPIO to acknowledge the request.

Each of the three GPIO modules provides two independent interrupt channels. Identical in functionality, these are called interrupt A and interrupt B. Both interrupt channels have their own mask register which lets you assign the individual GPIOs to none, either, or both interrupt channels.

Since all mask registers reset to zero, none of the GPIOs is assigned any interrupt by default. Each GPIO represents a bit in each of these registers. Setting a bit means enabling the interrupt on this channel.

Interrupt A and interrupt B operate independently. For example, writing 1 to a bit in the mask interrupt A register does not affect interrupt channel B. This facility allows GPIOs to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

A GPIO interrupt is generated by a logical OR of all unmasked GPIOs for that interrupt. For example, if `PF0` and `PF1` are both unmasked for GPIO interrupt channel A, GPIO interrupt A will be generated when triggered by `PF0` or `PF1`. The interrupt service routine must evaluate the GPIO data register to determine the signaling interrupt source. Figure 9-1 illustrates the interrupt flow of any GPIO module's interrupt A channel.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate bit in the GPIO clear register.

At reset, all interrupts are masked and disabled.

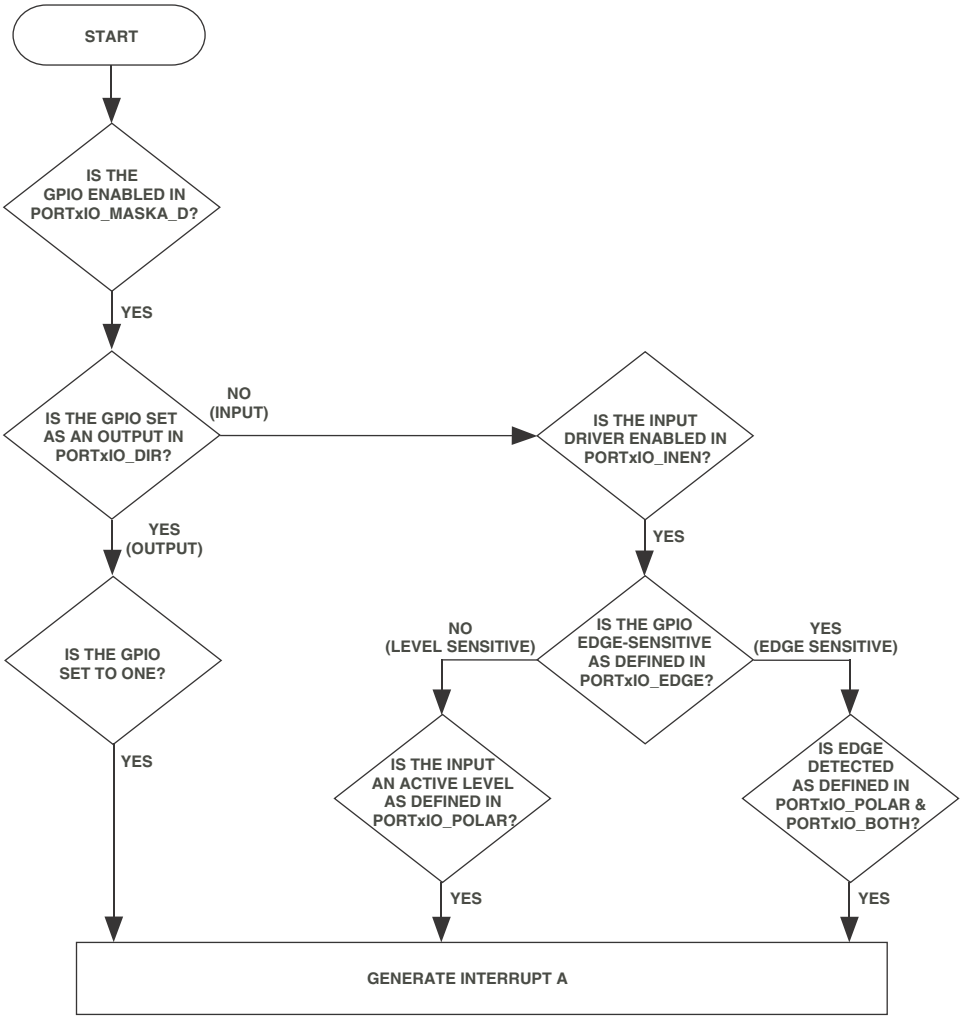


Figure 9-1. GPIO Interrupt Generation Flow for Interrupt Channel A

Similarly to the GPIOs themselves, the mask register can either be written through the GPIO mask data registers (PORTxIO_MASKA, PORTxIO_MASKB) or be controlled by the mask A/mask B set, clear and toggle registers.

The GPIO mask interrupt set registers (PORTxIO_MASKA_SET, PORTxIO_MASKB_SET) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt set register can be used to set a single or a few bits only. No read-modify-write operations are required.

The mask interrupt set registers are write-1-to-set registers. All ones contained in the value written to the mask interrupt set register set the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit enables the interrupt for the respective GPIO.

The GPIO mask interrupt clear registers (PORTxIO_MASKA_CLEAR, PORTxIO_MASKB_CLEAR) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to the mask interrupt clear register can be used to clear a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt clear registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt clear register clear the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit disables the interrupt for the respective GPIO.

The GPIO mask interrupt toggle registers (PORTxIO_MASKA_TOGGLE, PORTxIO_MASKB_TOGGLE) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt toggle register can be used to toggle a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt toggle registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt toggle register toggle the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit toggles the interrupt for the respective GPIO.

Figure 9-1 illustrates the interrupt flow of any GPIO module's interrupt A channel. The interrupt B channel behaves identically.

All GPIOs assigned to the same interrupt channel are OR'ed. (See Figure 9-2.) If multiple GPIOs are assigned to the same interrupt channel, it is up to the interrupt service routine to evaluate the GPIO data registers to determine the signaling interrupt source.

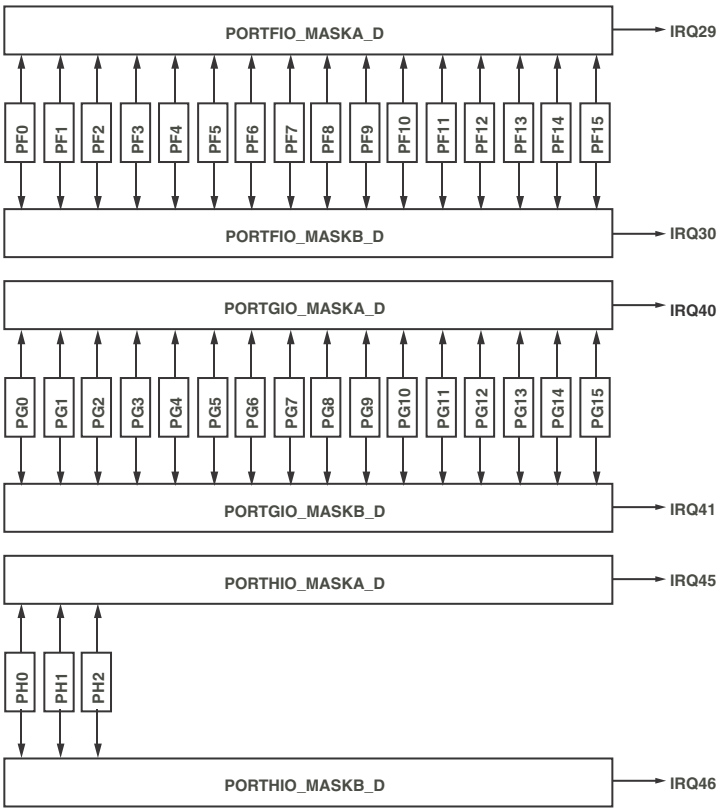


Figure 9-2. GPIO Interrupt Channels

Programming Model

Figure 9-3 and Figure 9-4 show the programming model for the general-purpose ports.

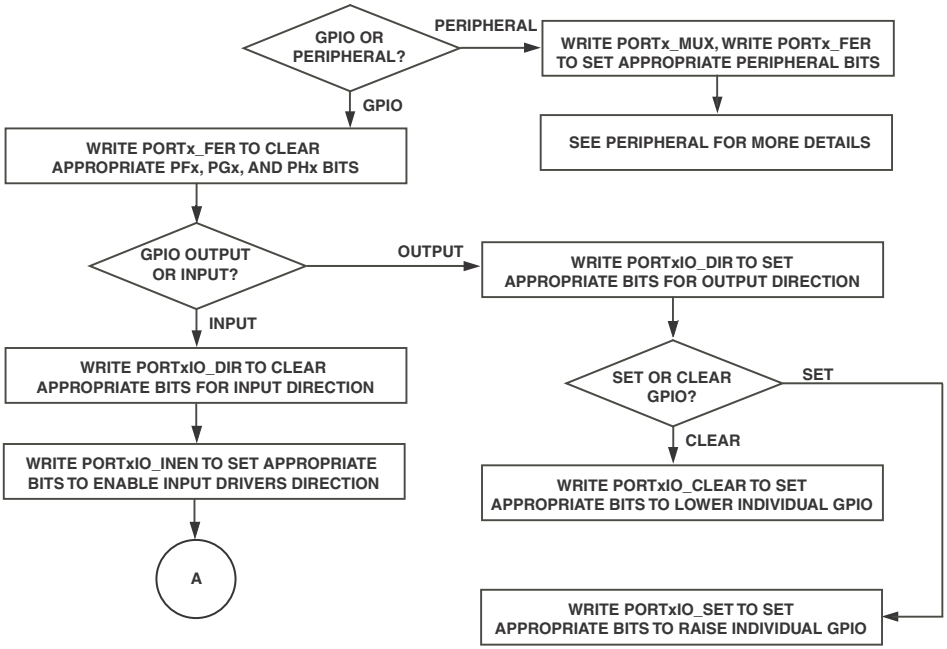


Figure 9-3. GPIO Flow Chart (Part 1 of 2)

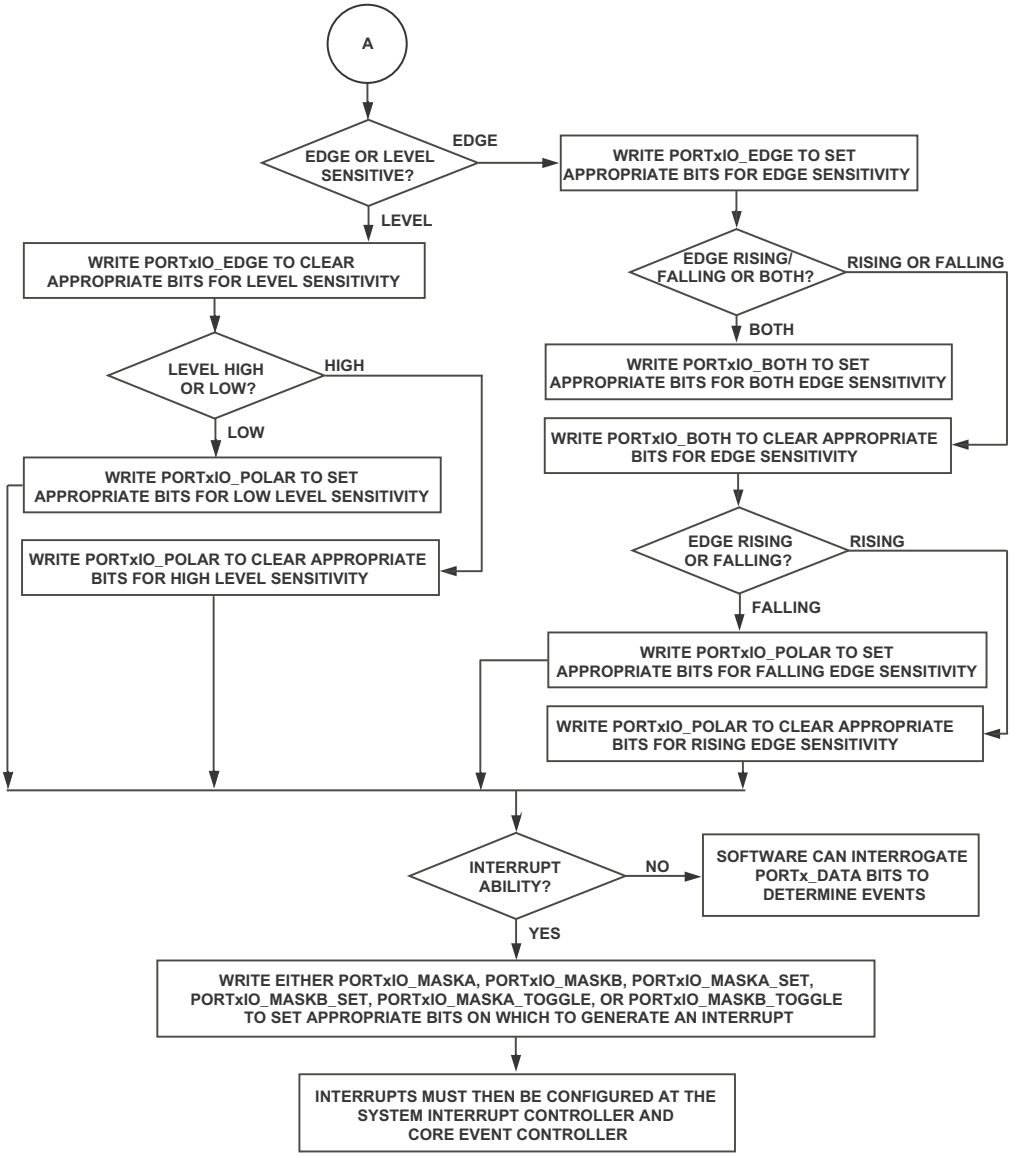


Figure 9-4. GPIO Flow Chart (Part 2 of 2)

Hysteresis Control

The ADSP-BF50x contains additional registers controlling the hysteresis (via Schmitt triggering) for Port F, Port G, and Port H. These are also included for pins other than GPIOs. Figure 9-5 to Figure 9-7 show the bit descriptions of these registers.

PORTx Hysteresis (PORTx_HYSTERESIS) Register

This register configures Schmitt triggering (SE) for the PORTx inputs. The Schmitt trigger can be set only for pin groups, classified by the pin muxing controls. For each controlled group of pins, `b#00` will disable Schmitt triggering, while `b#01` will enable it. Combinations of `b#1x` are reserved.

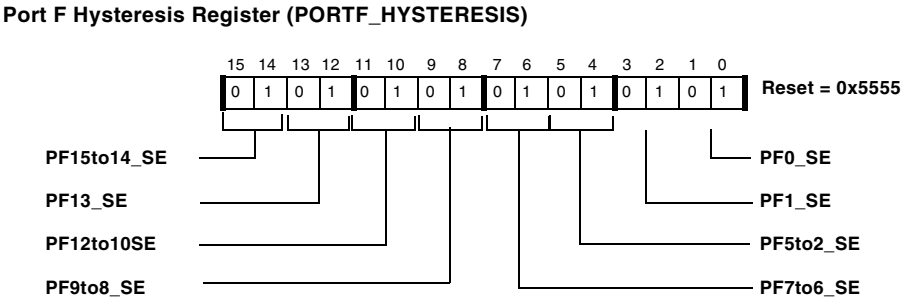


Figure 9-5. Port F Hysteresis Register

Port G Hysteresis Register (PORTG_HYSTERESIS)

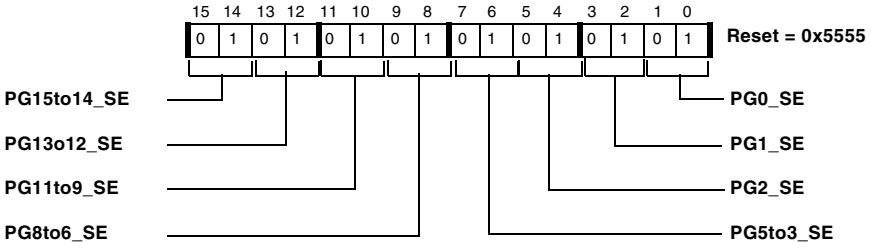


Figure 9-6. Port G Hysteresis Register

Port H Hysteresis Register (PORTH_HYSTERESIS)

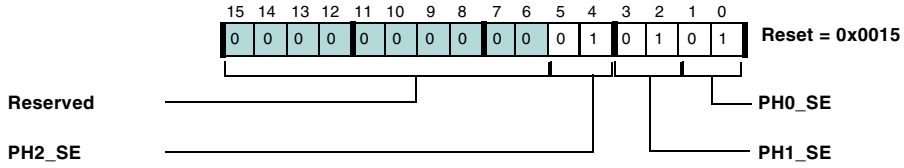


Figure 9-7. Port H Hysteresis Register

The NONGPIO_HYSTERESIS register sets the Schmitt trigger (SE) for various ADSP-BF50x signals.

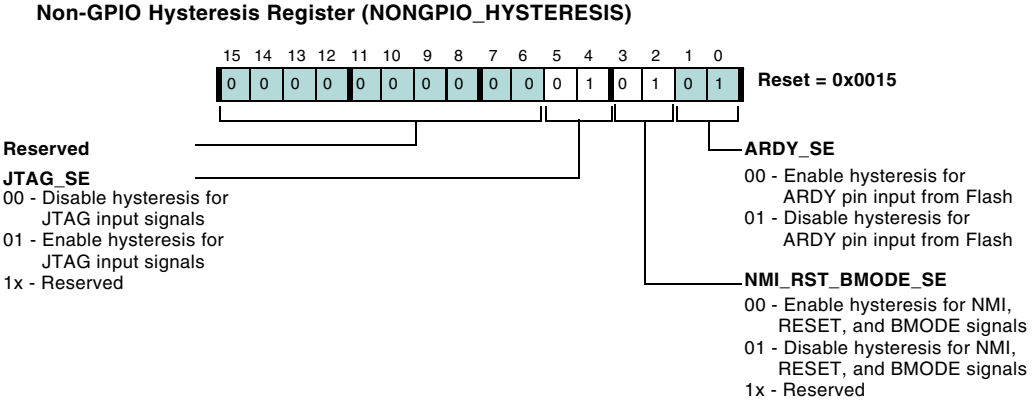


Figure 9-8. Non-GPIO Hysteresis Register

Drive Strength Control

The NONGPIO_DRIVE register sets the drive strength and tolerance for the TWI signals on the ADSP-BF50x as specified in the diagram.

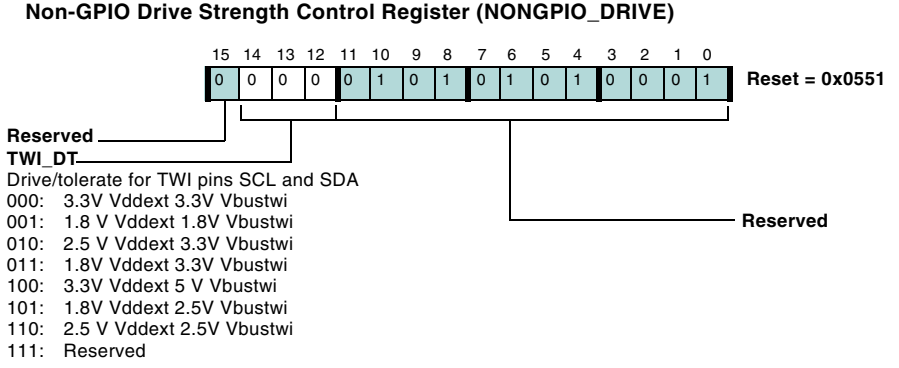


Figure 9-9. Non-GPIO Drive Strength Control Register

Memory-Mapped GPIO Registers

The GPIO registers are part of the system memory-mapped registers (MMRs). Figure 9-10 through Figure 9-30 on page 9-41 illustrate the GPIO registers. The addresses of the programmable flag MMRs appear in Appendix B.

In Figure 9-10 through Figure 9-30, bits 3-15 are reserved for Port H register descriptions.

Port Multiplexer Control Registers (PORTx_MUX)

Figure 9-10 shows the Port F Multiplexer Control register. Refer to Table 9-1 on page 9-4 for more information on multiplexed configurations within Port F.

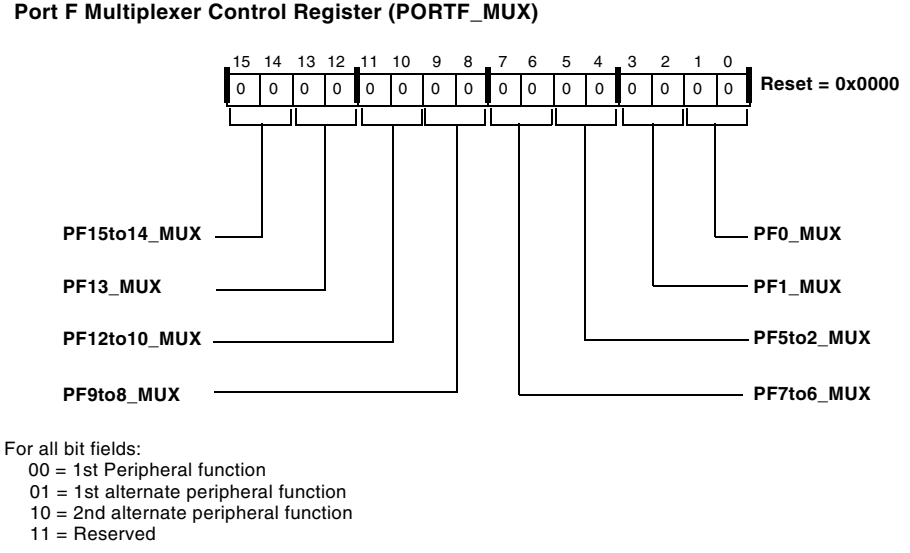
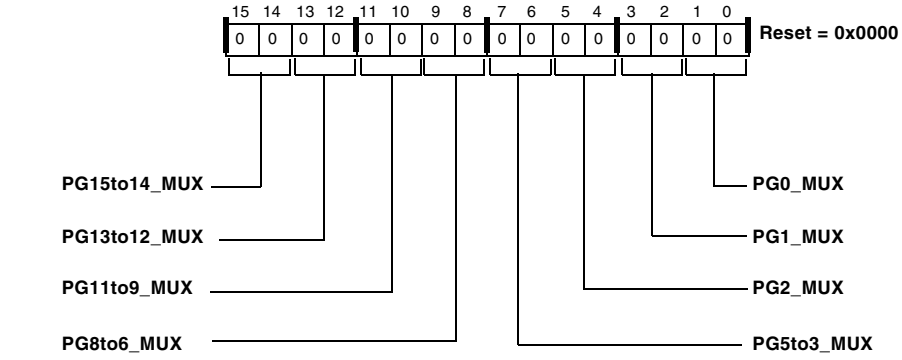


Figure 9-10. Port F Multiplexer Control Register

Figure 9-11 shows the Port G Multiplexer Control register. Refer to Table 9-2 on page 9-5 for more information on multiplexed configurations within Port G.

Port G Multiplexer Control Register (PORTG_MUX)



For all bit fields:
00 = 1st Peripheral function
01 = 1st alternate peripheral function
10 = 2nd alternate peripheral function
11 = Reserved

Figure 9-11. Port G Multiplexer Control Register

Figure 9-12 shows the Port H Multiplexer Control register. Refer to Table 9-3 on page 9-6 for more information on multiplexed configurations within Port H.

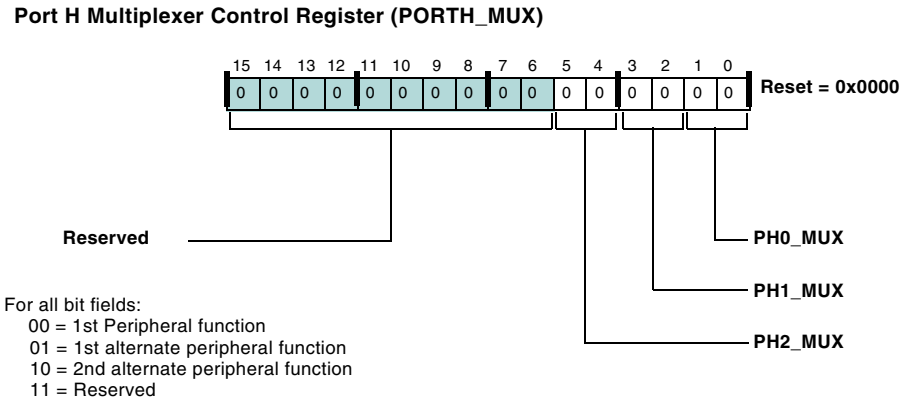


Figure 9-12. Port H Multiplexer Control Register

Function Enable Registers (PORTx_FER)

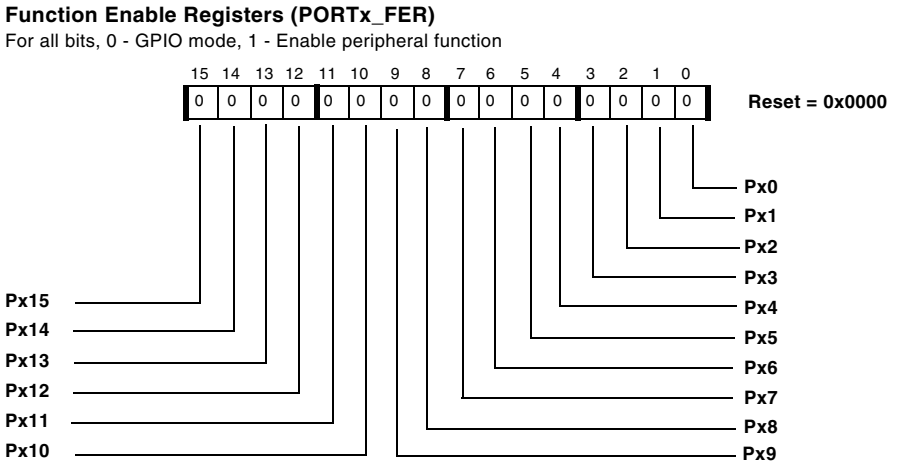


Figure 9-13. Function Enable Registers

GPIO Direction Registers (PORTxIO_DIR)

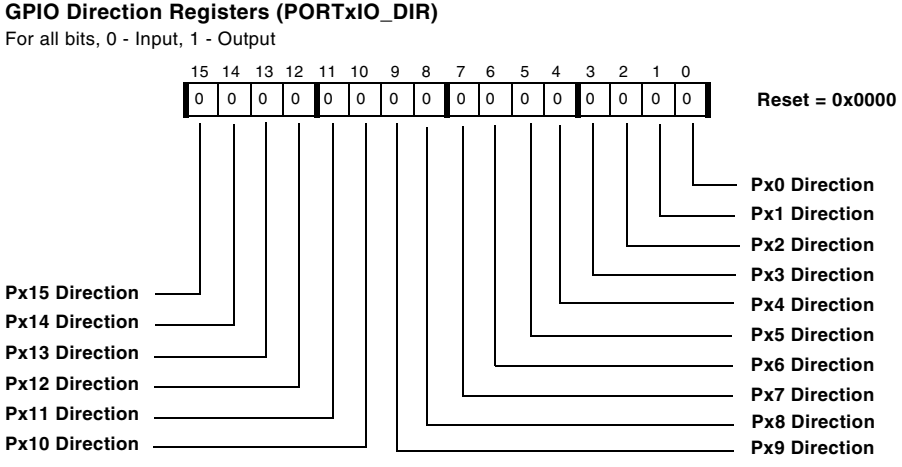


Figure 9-14. GPIO Direction Registers

GPIO Input Enable Registers (PORTxIO_INEN)

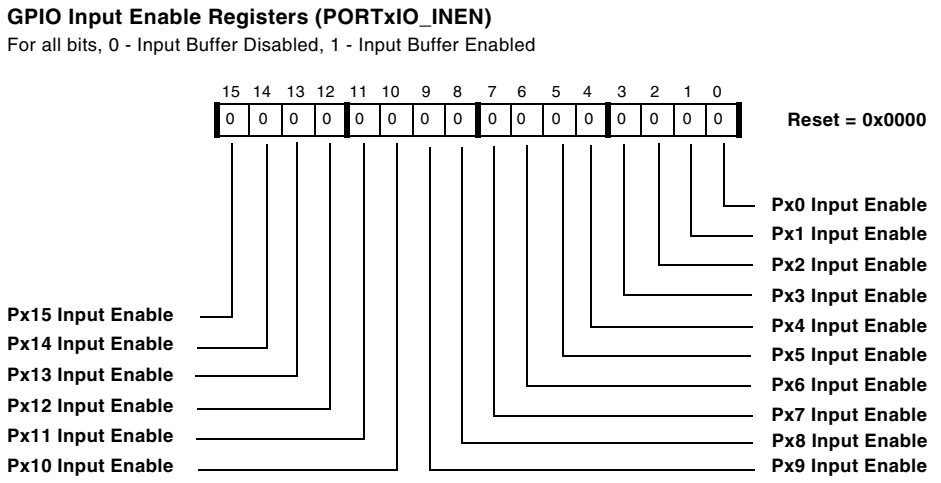


Figure 9-15. GPIO Input Enable Registers

GPIO Data Registers (PORTxIO)

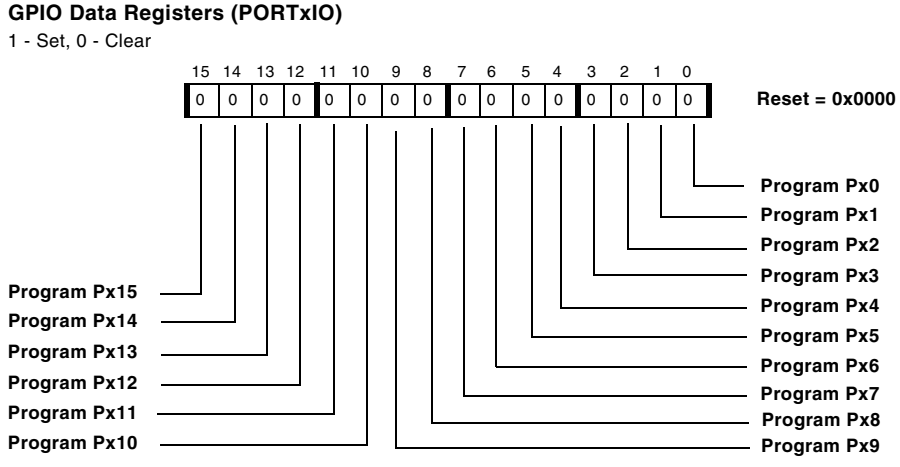


Figure 9-16. GPIO Data Registers

GPIO Set Registers (PORTxIO_SET)

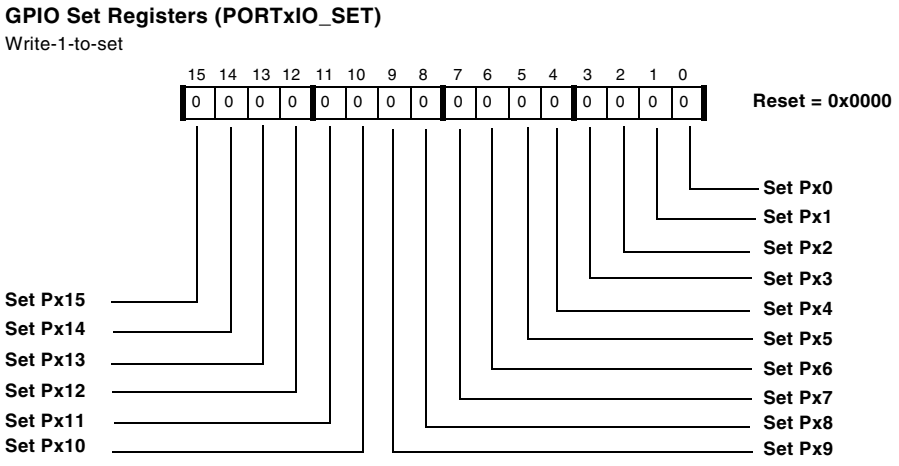


Figure 9-17. GPIO Set Registers

GPIO Clear Registers (PORTxIO_CLEAR)

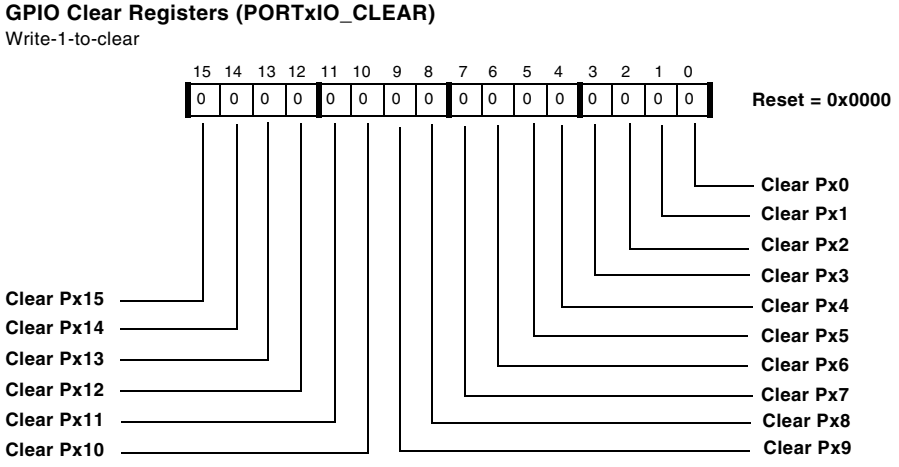


Figure 9-18. GPIO Clear Registers

GPIO Toggle Registers (PORTxIO_TOGGLE)

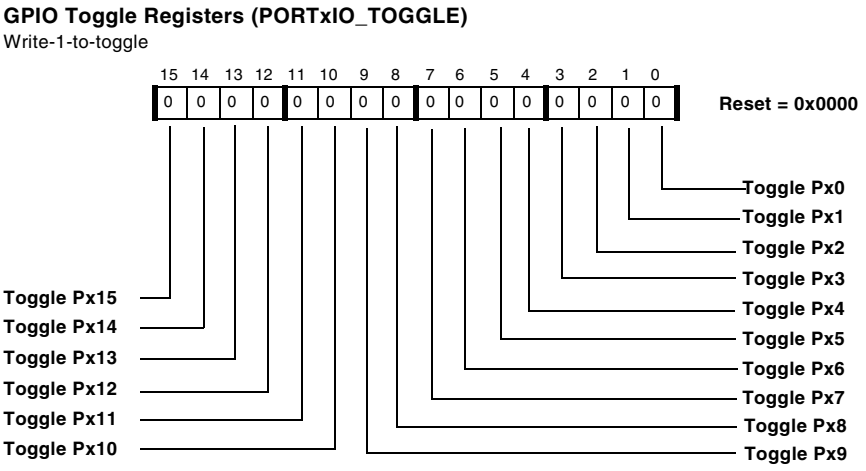


Figure 9-19. GPIO Toggle Registers

GPIO Polarity Registers (PORTxIO_POLAR)

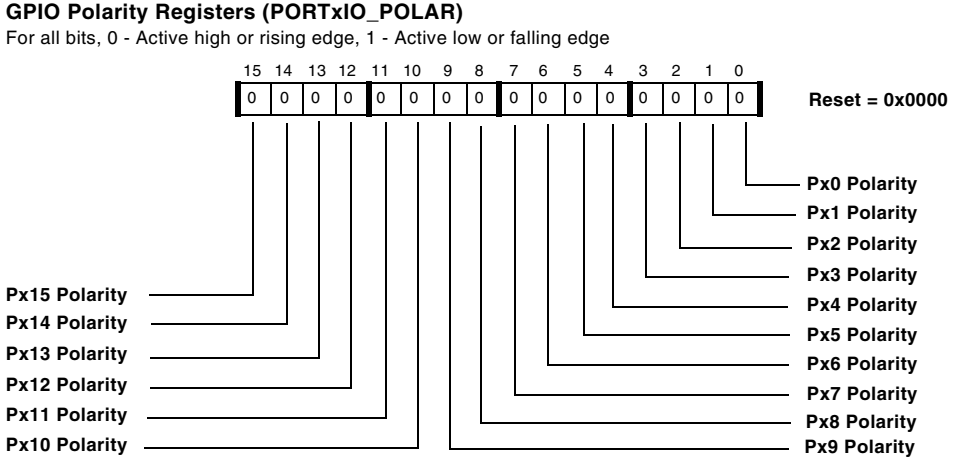


Figure 9-20. GPIO Polarity Registers

Interrupt Sensitivity Registers (PORTxIO_EDGE)

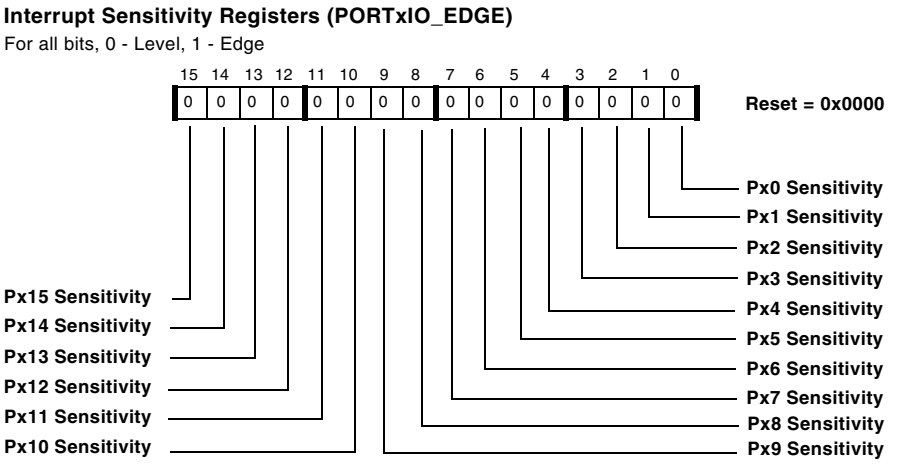


Figure 9-21. Interrupt Sensitivity Registers

GPIO Set on Both Edges Registers (PORTxIO_BOTH)

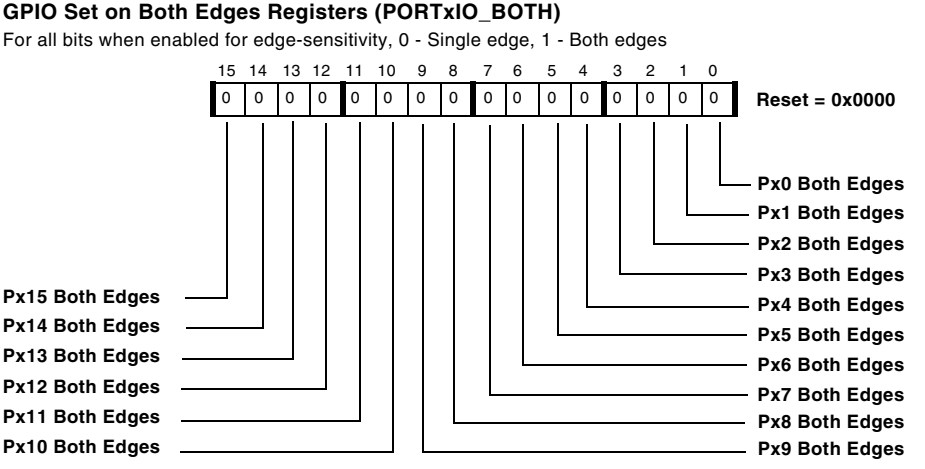


Figure 9-22. GPIO Set on Both Edges Registers

GPIO Mask Interrupt Registers (PORTxIO_MASKA/B)

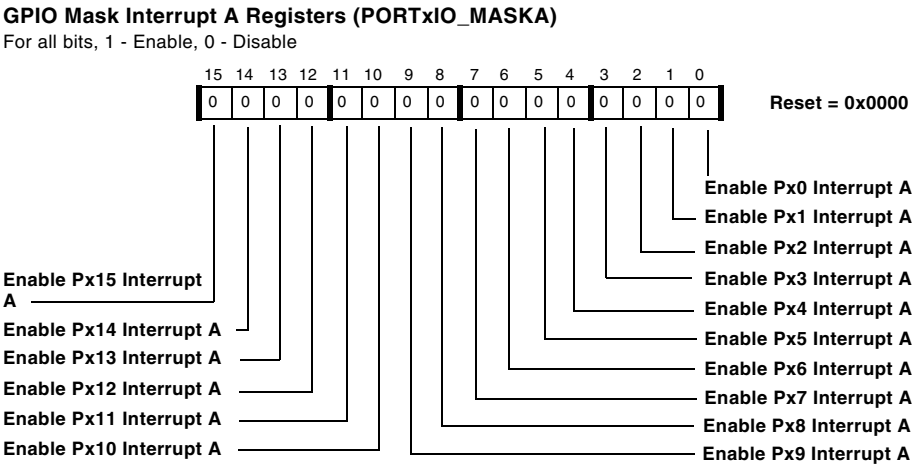


Figure 9-23. GPIO Mask Interrupt A Registers

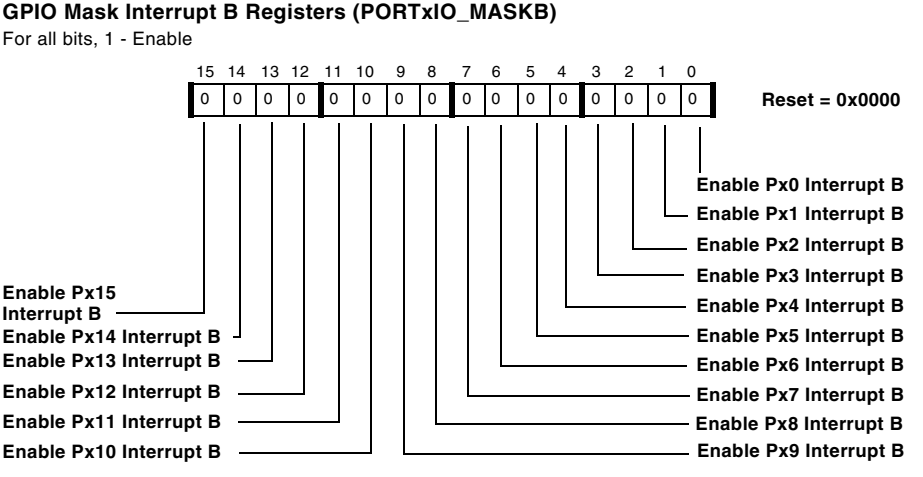


Figure 9-24. GPIO Mask Interrupt B Registers

GPIO Mask Interrupt Set Registers (PORTxIO_MASKA/B_SET)

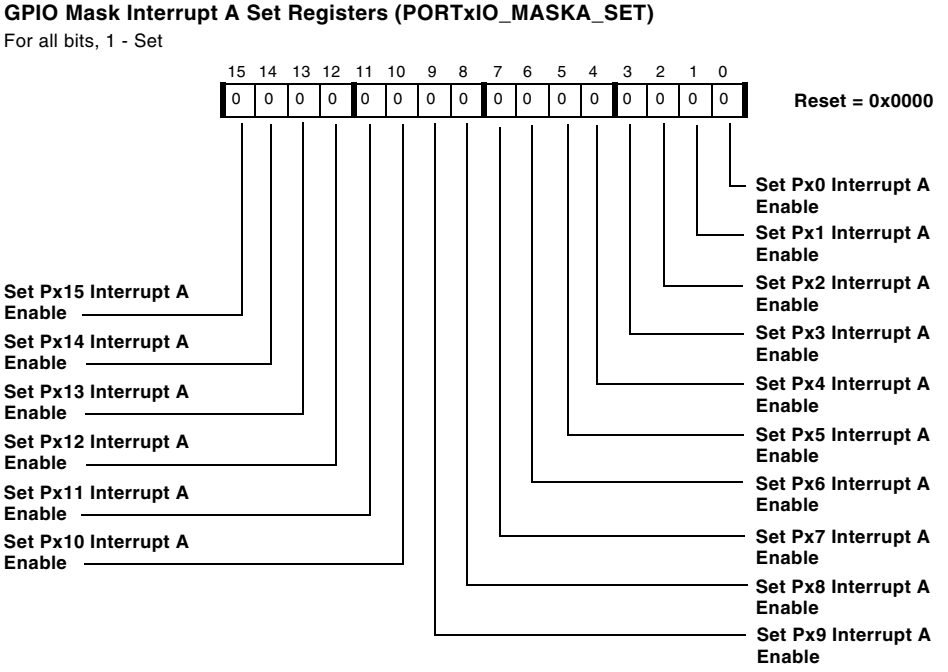


Figure 9-25. GPIO Mask Interrupt A Set Registers

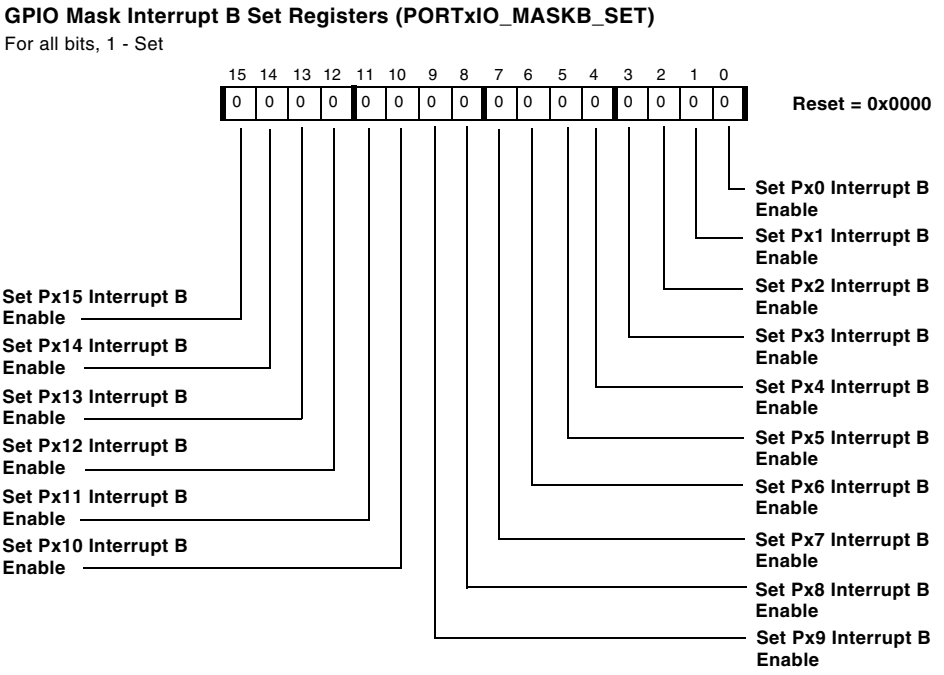


Figure 9-26. GPIO Mask Interrupt B Set Registers

GPIO Mask Interrupt Clear Registers (PORTxIO_MASKA/B_CLEAR)

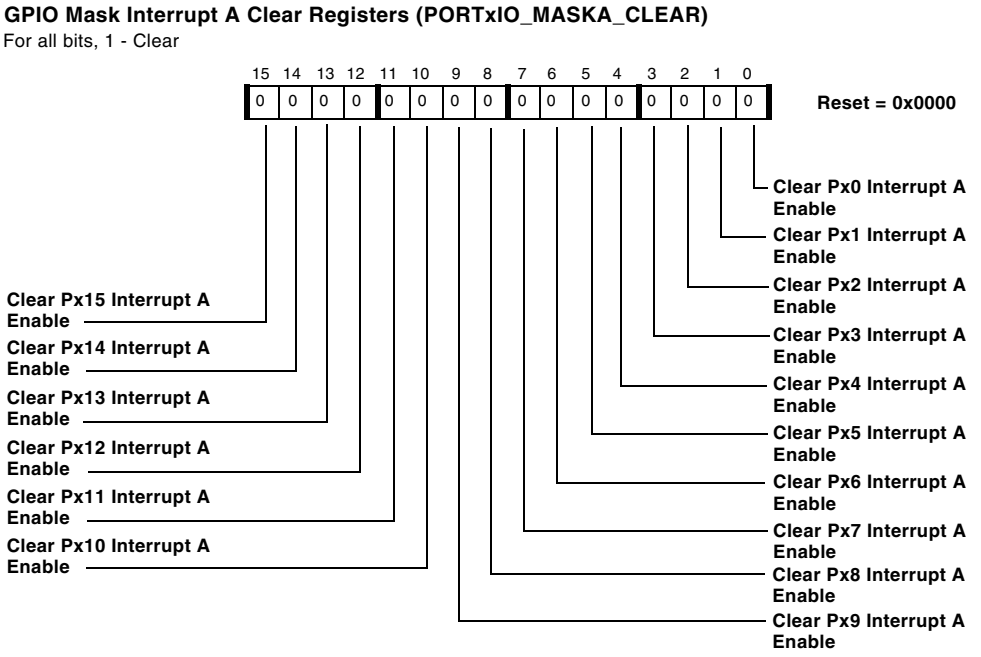


Figure 9-27. GPIO Mask Interrupt A Clear Registers

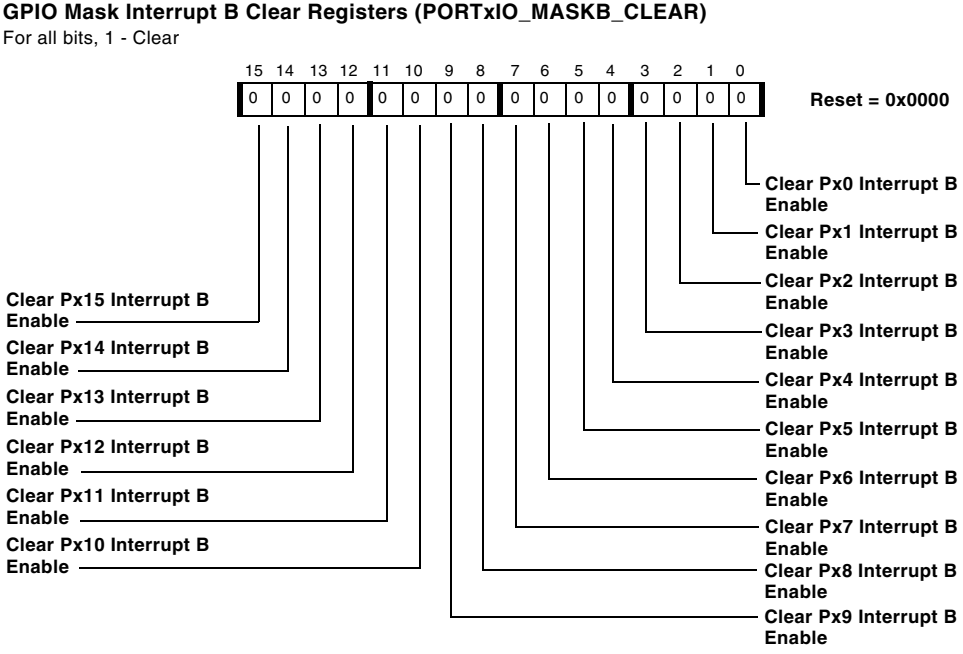


Figure 9-28. GPIO Mask Interrupt B Clear Registers

GPIO Mask Interrupt Toggle Registers (PORTxIO_MASKA/B_TOGGLE)

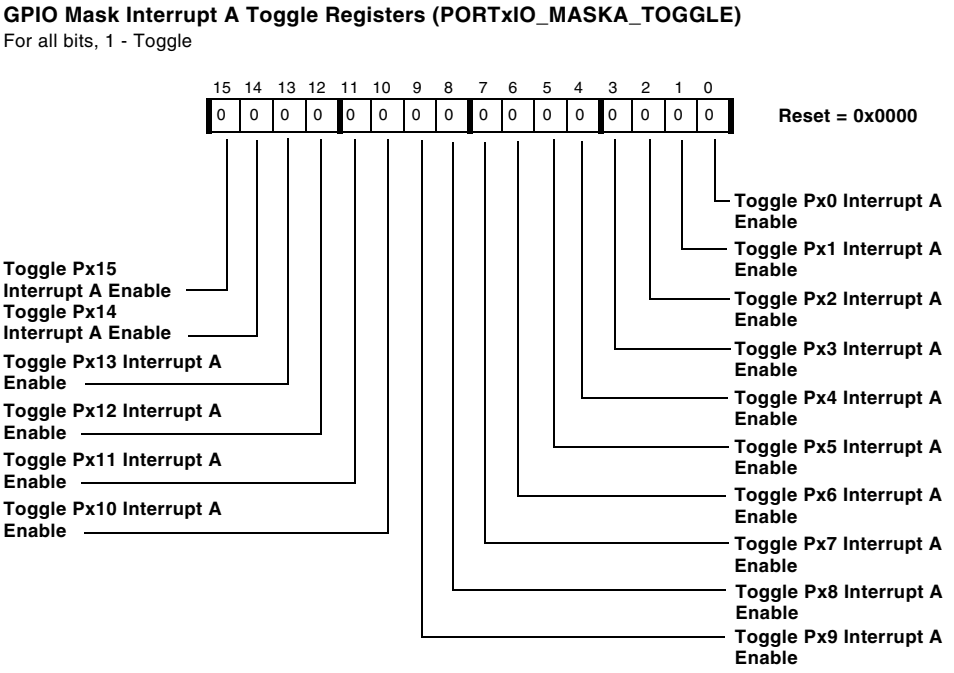


Figure 9-29. GPIO Mask Interrupt A Toggle Registers

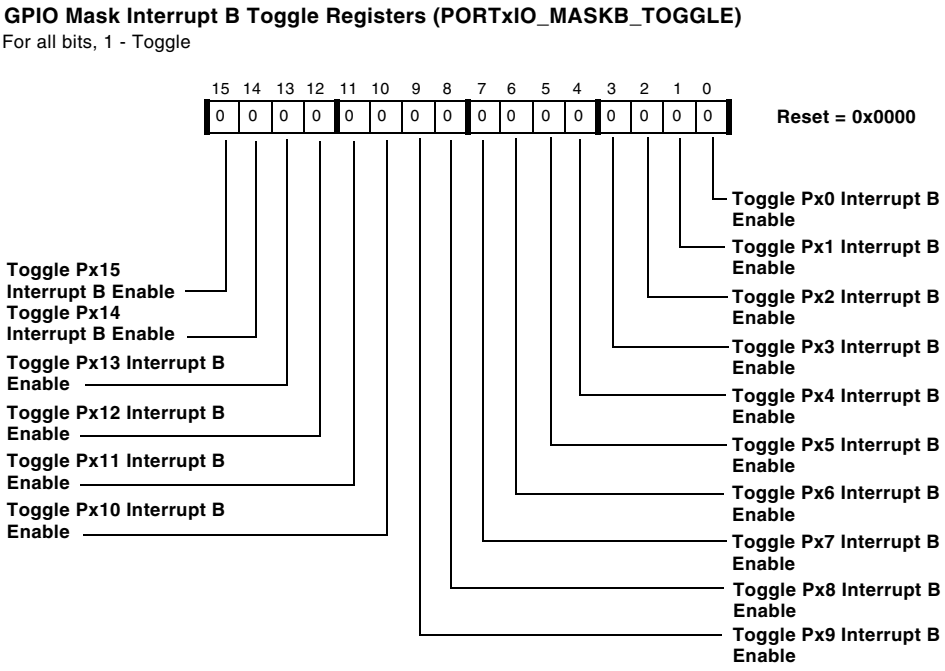


Figure 9-30. GPIO Mask Interrupt B Toggle Registers

Programming Examples

Listing 9-1 provides examples for using the general-purpose ports.

Listing 9-1. General-Purpose Ports

```
/* set port f function enable register to GPIO (not peripheral)
*/
p0.l = lo(PORTF_FER);
p0.h = hi(PORTF_FER);

R0.h = 0x0000;
r0.l = 0x0000;
```

```
w[p0] = r0;

/* set port f direction register to enable some GPIO as output,
remaining are input */
p0.l = lo(PORTFIO_DIR);
p0.h = hi(PORTFIO_DIR);
r0.h = 0x0000;
r0.l = 0x0FC0;
w[p0] = r0;
ssync;

/* set port f clear register */
p0.l = lo(PORTFIO_CLEAR);
p0.h = hi(PORTFIO_CLEAR);
    r0.l = 0xFC0;
    w[p0] = r0;
    ssync;

/* set port f input enable register to enable input drivers of
some GPIOs */
p0.l = lo(PORTFIO_INEN);
p0.h = hi(PORTFIO_INEN);
r0.h = 0x0000;
r0.l = 0x003C;
w[p0] = r0;
ssync;

/* set port f polarity register */
p0.l = lo(PORTFIO_POLAR);
p0.h = hi(PORTFIO_POLAR);
r0 = 0x00000;
w[p0] = r0;
ssync;
```

10 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose (GP) timer module. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of GP timers for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For GP Timer interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the GP Timers is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 4-3 on page 4-19 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each GP Timer, refer to Chapter A, "System MMR Assignments".

GP timer behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF50x Processor" on page 10-57.

Overview

The general-purpose timers support the following operating modes:

- Single-shot mode for interval timing and single pulse generation
- Pulse width modulation (PWM) generation with consistent update of period and pulse width values
- External signal capture mode with consistent update of period and pulse width values
- External event counter mode

Feature highlights are:

- Synchronous operation
- Consistent management of period and pulse width values
- Interaction with PPI module for video frame sync operation
- Autobaud detection for UART module
- Graceful bit pattern termination when stopping
- Support for center-aligned PWM patterns
- Error detection on implausible pattern values
- All read and write accesses to 32-bit registers are atomic
- Every timer has its dedicated interrupt request output
- Unused timers can function as edge-sensitive pin interrupts

The internal structure of the individual timers is illustrated by Figure 10-1, which shows the details of timer 0 as a representative example. The other timers have identical structure.

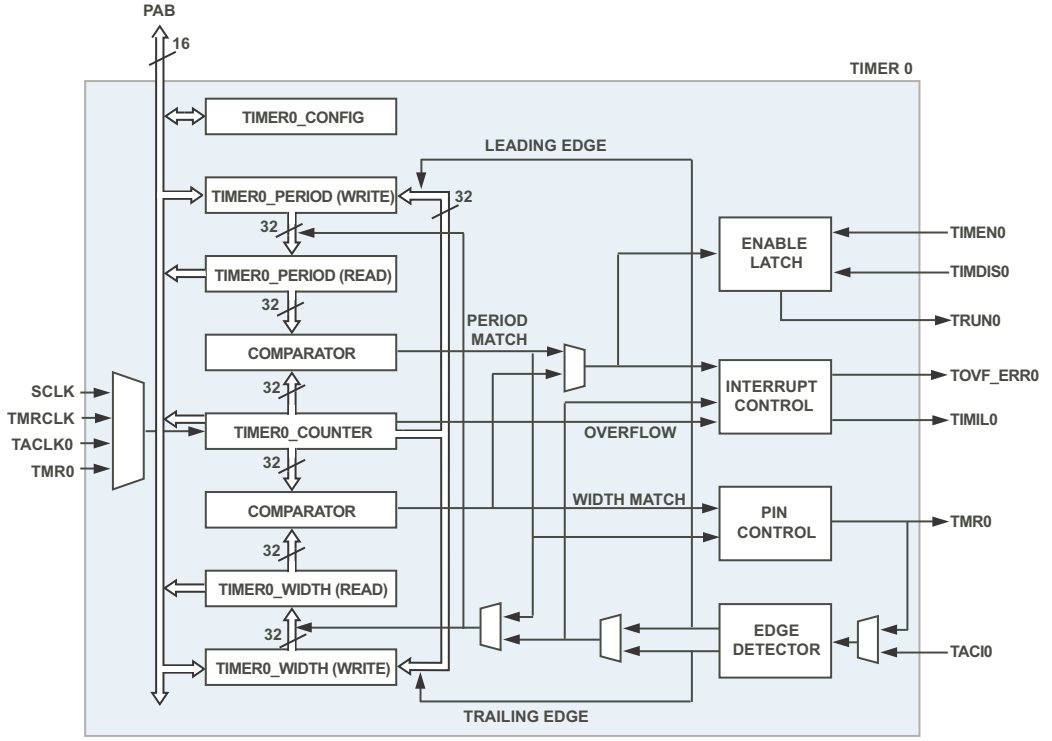


Figure 10-1. Internal Timer Structure

External Interface

Every timer has a dedicated TMR pin. If enabled, the TMR pins output the single-pulse or PWM signals generated by the timer. The TMR pins function as input in capture and counter modes. Polarity of the signals is programmable.

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is $((2^{32}-1) / 133\ \text{MHz}) = 32.2$ seconds.

Clock and capture input pins are sampled every SCLK cycle. The duration of every low or high state must be at least one SCLK. Therefore, the maximum allowed frequency of timer input signals is SCLK/2.

Internal Interface

Timer registers are always accessed by the core through the 16-bit PAB bus. Hardware ensures that all read and write operations from and to 32-bit timer registers are atomic.

Every timer has a dedicated interrupt request output that connects to the system interrupt controller (SIC).

Description of Operation

The core of every timer is a 32-bit counter, that can be interrogated through the read-only TIMER_COUNTER register. Depending on the mode of operation, the counter is reset to either 0x0000 0000 or 0x0000 0001 when the timer is enabled. The counter always counts upward. Usually, it is clocked by SCLK. In PWM mode it can be clocked by the alternate clock input TACLK or, alternatively, the common timer clock input TMRCLK. In counter mode, the counter is clocked by edges on the TMR input pin. The significant edge is programmable.

After 2^{32}-1 clocks, the counter overflows. This is reported by the overflow/error bit TOVF_ERR in the TIMER_STATUS register. In PWM and counter mode, the counter is reset by hardware when its content reaches the values stored in the TIMER_PERIOD register. In capture mode, the counter is reset by leading edges on the TMR or TACI input pin. If enabled, these events cause the interrupt latch TIMIL in the TIMER_STATUS register to be set and issue a system interrupt request. The TOVF_ERR and TIMIL latches are sticky and should be cleared by software using W1C (write-1-to-clear) operations to clear the interrupt request. The global TIMER_STATUS register

is 32-bits wide. A single atomic 32-bit read can report the status of all corresponding timers.

Before a timer can be enabled, its mode of operation is programmed in the individual timer-specific TIMER_CONFIG register. Then, the timers are started by writing a "1" to the representative bits in the global TIMER_ENABLE register.

The TIMER_ENABLE register can be used to enable all timers simultaneously. The register contains W1S (write-1-to-set) control bits, one for each timer. Correspondingly, the TIMER_DISABLE register contains W1C control bits to allow simultaneous or independent disabling of the timers. Either register can be read to check the enable status of the timers. A "1" indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMEN bit is set.

While the PWM mode is used to generate PWM patterns, the capture mode (WDTH_CAP) is designed to "receive" PWM signals. A PWM pattern is represented by a pulse width and a signal period. This is described by the TIMER_WIDTH and TIMER_PERIOD register pair. In capture mode these registers are read only. Hardware always captures both values. Regardless of whether in PWM or capture mode, shadow buffers always ensure consistency between the TIMER_WIDTH and TIMER_PERIOD values. In PWM mode, hardware performs a plausibility check by the time the timer is enabled. If there is an error, the type is reported by the TIMER_CONFIG register and signalled by the TOVF_ERR bit.

Interrupt Processing

Each timer can generate a single interrupt. The resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source.

Figure 10-2 shows the interrupt structure of the timers.

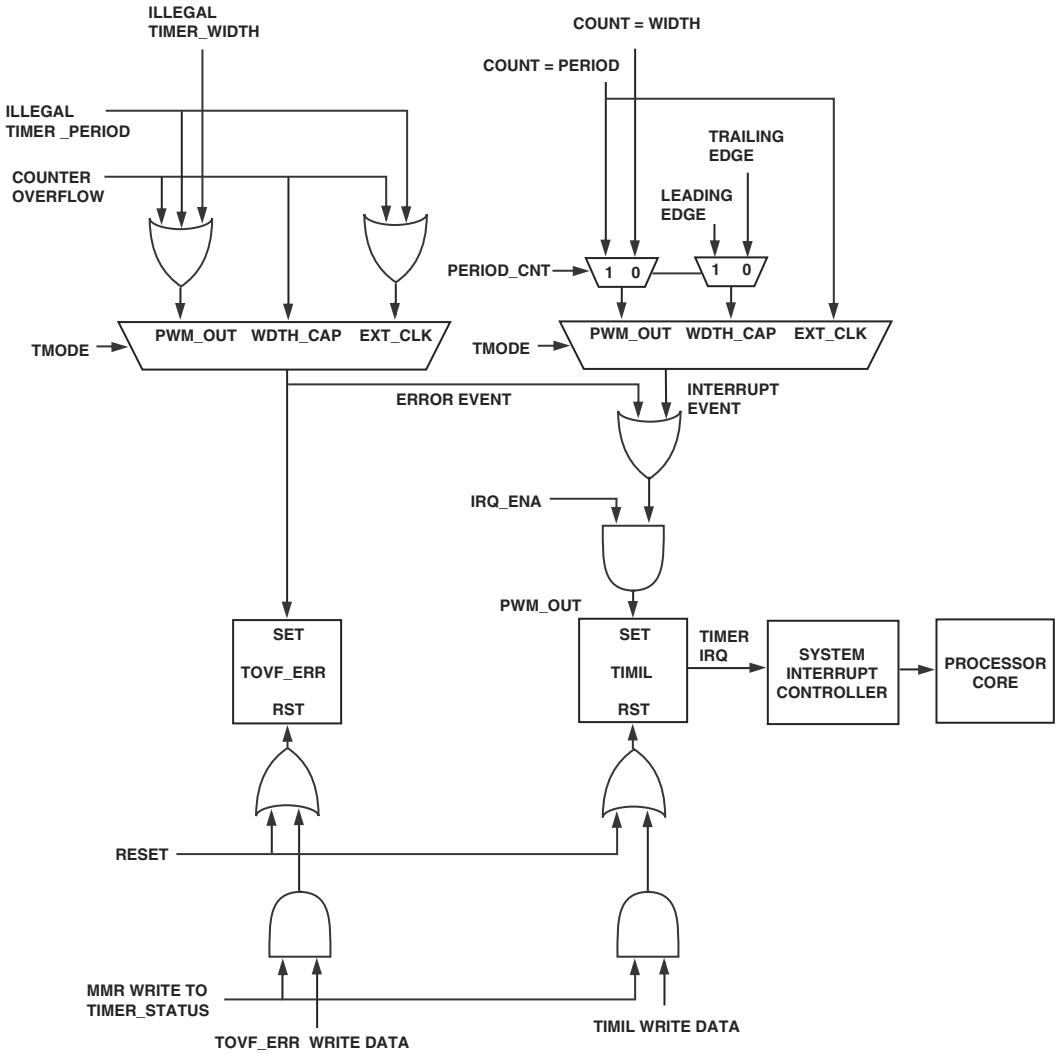


Figure 10-2. Timers Interrupt Structure

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the IMASK and SIC_IMASK registers. To poll the TIMIL bit

without interrupt generation, set IRQ_ENA but leave the interrupt masked at the system level. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions as reported by the TOVF_ERR bits.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same CEC interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, multiple timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMIL latch bits at once by writing 0x000F 000F to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMIL bit in the TIMER_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMIL clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMIL bit in the TIMER_STATUS register at the very beginning of the interrupt service routine to avoid missing any timer events.

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width vs. period plausibility checks. Errors are reported by the TOVF_ERR bits in the TIMER_STATUS register and the ERR_TYP bit field in the individual TIMER_CONFIG registers. Table 10-1 provides a summary of error conditions, using these terms:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMER_PERIOD and the counter is reloaded with the value “1”.

- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads "0" if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads "1".

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write "1" to clear TOVF_ERR to acknowledge the error.

The following table can be read as: "In mode __ at event __, if TIMER_PERIOD is __ and TIMER_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __."

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMR pin.

Table 10-1. Overview of Illegal States

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_CNT = 1	Startup (No boundary condition tests performed on TIMER_WIDTH)	== 0	Anything	b#10	Set
		== 1	Anything	b#10	Set
		≥ 2	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		== 1	Anything	b#11	Set
		≥ 2	== 0	b#11	Set
		≥ 2	< TIMER_PERIOD	No change	No change
		≥ 2	≥ TIMER_PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set
PWM_OUT, PERIOD_CNT = 0	Startup	Anything	== 0	b#01	Set
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		Anything	≥ 1	No change	No change
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMER_WIDTH == 0	Anything	Anything	b#01	Set

Table 10-1. Overview of Illegal States (Cont'd)

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
WDTH_CAP	Startup	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set

Modes of Operation

The following sections provide a functional description of the general-purpose timers in various operating modes.

Pulse Width Modulation (PWM_OUT) Mode

Use the PWM_OUT mode for PWM signal or single-pulse generation, for interval timing or for periodic interrupt generation. Figure 10-3 illustrates PWM_OUT mode.

Setting the TMODE field to b#01 in the TIMER_CONFIG register enables PWM_OUT mode. Here, the TMR pin is an output, but it can be disabled by setting the OUT_DIS bit in the TIMER_CONFIG register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

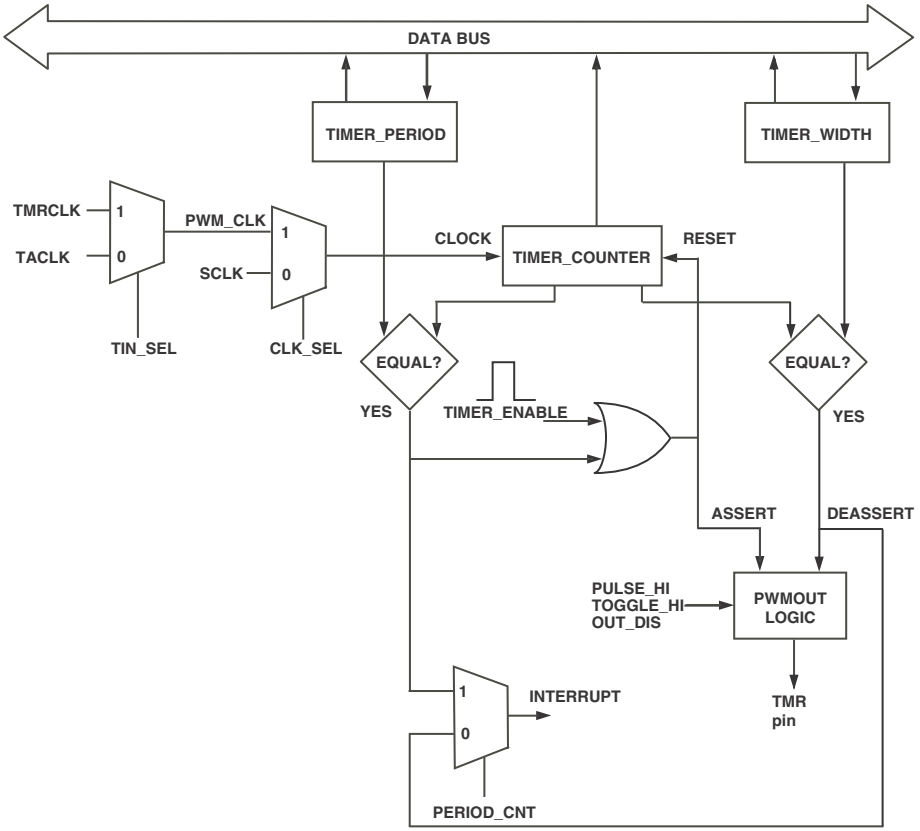


Figure 10-3. Timer Flow Diagram, PWM_OUT Mode

Once a timer has been enabled, the timer counter register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the timer period register. For either setting of CLK_SEL, when the timer counter equals the timer period, the timer counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMER_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMER_PERIOD and TIMER_WIDTH registers and generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of PERIOD_CNT = 0 counts to the end of the width; a setting of PERIOD_CNT = 1 counts to the end of the period.

The TIMER_PERIOD and TIMER_WIDTH registers are read-only in some operation modes. Be sure to set the TMODE field in the TIMER_CONFIG register to b#01 before writing to these registers.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the TIMER_CONFIG register. The TMR pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used. The TMR pin can also be disabled by the function enable and the multiplexer control registers.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMR pin. This mode can also be used to implement a precise delay.

The pulse width is defined by the TIMER_WIDTH register, and the TIMER_PERIOD register is not used. See Figure 10-4.

At the end of the pulse, the timer interrupt latch bit TIMIL is set, and the timer is stopped automatically. No writes to the TIMER_DISABLE register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMR pin. If PULSE_HI is not set, the pulse is active low.

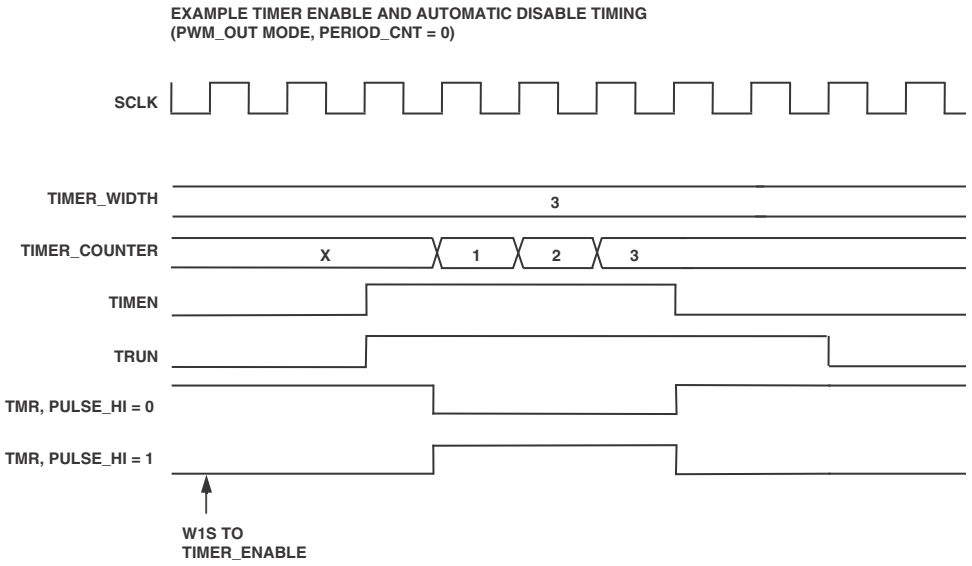


Figure 10-4. Timer Enable and Automatic Disable Timing

The pulse width may be programmed to any value from 1 to (2^{32}-1), inclusive.

Pulse Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle (PWM patterns). This mode also generates periodic interrupts for real-time signal processing.

The 32-bit TIMER_PERIOD and TIMER_WIDTH registers are programmed with the values required by the PWM signal.

When the timer is enabled in this mode, the TMR pin is pulled to a deasserted state each time the counter equals the value of the pulse width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMR pin, the PULSE_HI bit in the corresponding TIMER_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMR pin is driven to the deasserted level.

Figure 10-5 shows timing details.

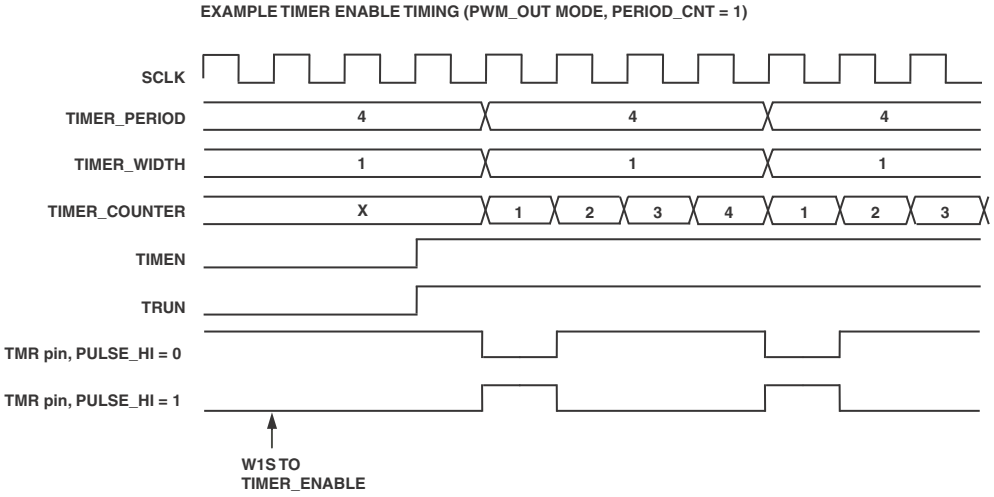


Figure 10-5. Timer Enable Timing

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine must clear the interrupt latch bit (TIMIL) and might alter period and/or width values. In PWM applications, the software needs to update period and pulse width values while the timer is

running. When software updates either the TIMER_PERIOD or TIMER_WIDTH registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. Reads from TIMER_PERIOD and TIMER_WIDTH registers return the old values until the period expires.

The TOVF_ERR status bit signifies an error condition in PWM_OUT mode. The TOVF_ERR bit is set if TIMER_PERIOD = 0 or TIMER_PERIOD = 1 at startup, or when the timer counter register rolls over. It is also set if the timer pulse width register is greater than or equal to the timer period register by the time the counter rolls over. The ERR_TYP bits are set when the TOVF_ERR bit is set.

Although the hardware reports an error if the TIMER_WIDTH value equals the TIMER_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the TOVL_ERR flags. Pulse width values greater than the period value are not recommended. Similarly, TIMER_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

To generate the maximum frequency on the TMR output pin, set the period value to "2" and the pulse width to "1". This makes the pin toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (period – 1), inclusive.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMER_WIDTH register). When two or more timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 10-6.

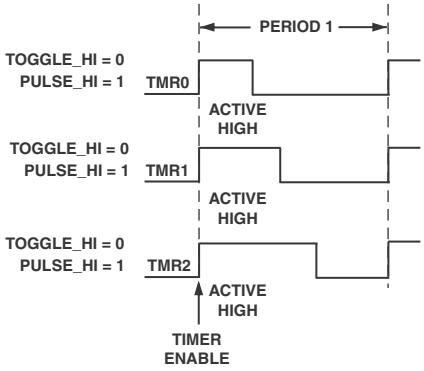


Figure 10-6. Example of Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a symmetrical rectangular waveform. The effective waveform is active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of the TOGGLE_HI bit has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when Count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 10-7 shows an example with three timers running with the same period settings. When software does not alter the PWM settings at run-time, the duty cycle is 50%. The values of the TIMER_WIDTH registers control the phase between the signals.

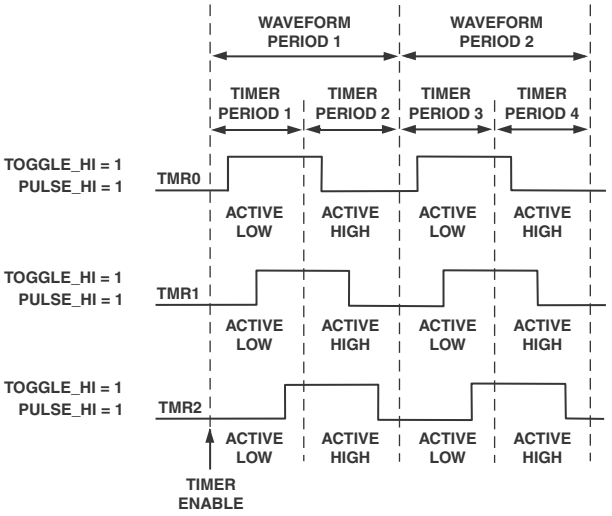


Figure 10-7. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (see Figure 10-8).

When TOGGLE_HI = 0, software updates the TIMER_PERIOD and TIMER_WIDTH registers once per waveform period. When TOGGLE_HI = 1, software updates the TIMER_PERIOD and TIMER_WIDTH registers twice per waveform. Period values are half as large. In odd-numbered periods, write (Period – Width) instead of Width to the TIMER_WIDTH register in order to obtain center-aligned pulses.

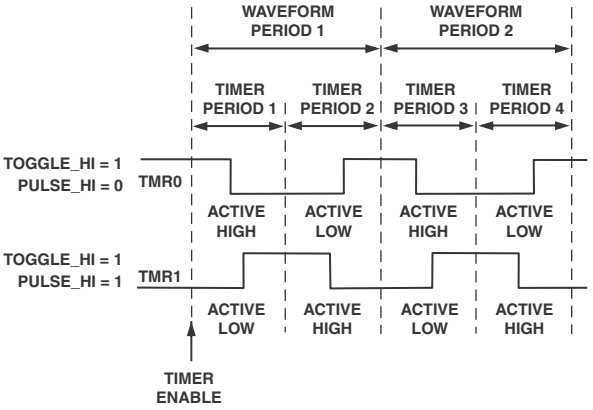


Figure 10-8. Two Timers With Non-Overlapping Clocks

For example, if the pseudo-code when `TOGGLE_HI` = 0 is:

```
int period, width;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, period) ;
     write (TIMER_WIDTH, width) ;
}
```

Then when `TOGGLE_HI` = `1`, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;
```

```
        per1 = period/2 ;
        wid1 = width/2 ;

        per2 = period/2 ;
        wid2 = width/2 ;

        waitfor (interrupt) ;

        write (TIMER_PERIOD, per1) ;
        write (TIMER_WIDTH, per1 - wid1) ;

        waitfor (interrupt) ;

        write (TIMER_PERIOD, per2) ;
        write (TIMER_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (`wid1` does not need to equal `wid2`). The period can be offset to adjust the phase of the pulses produced (`per1` does not need to equal `per2`).

The `TRUN` bit in the `TIMER_STATUS` register is updated only at the end of even-numbered periods in `TOGGLE_HI` mode. When `TIMER_DISABLE` is written to "1", the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when `TOGGLE_HI` = 0, errors are reported if the `TIMER_PERIOD` register is either set to "0" or "1", or when the width value is greater than or equal to the period value.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by `SCLK`. Alternatively, if the `CLK_SEL` bit in the `TIMER_CONFIG` register is set, the timer is clocked by

PWM_CLK. The PWM_CLK is normally input from the TACLK pin, but may be taken from the common TMRCLK pin regardless of whether the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMER_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMR pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMR pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The alternate timer clock inputs (TACLK) are enabled when a timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, without regard to the content of the multiplexer control and function enable registers.

Using PWM_OUT Mode With the PPI

Some timers may be used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to “Frame Synchronization in GP Modes” in Chapter 20, Parallel Peripheral Interface.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a “stop is pending” condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMR pin. The processor can determine when the timer stops running by polling for the corresponding TRUN bit in the TIMER_STATUS register to read “0” or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMER_CONFIG cannot be written to a new value) until after the timer stops and TRUN reads “0”.

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and TIMER_DISABLE) is cleared, and the corresponding TRUN bit is cleared. See Figure 10-4 on page 10-13. To generate multiple pulses, write a “1” to TIMER_ENABLE, wait for the timer to stop, then write another “1” to TIMER_ENABLE.

In continuous PWM generation mode (PWM_OUT, PERIOD_CNT = 1) software can stop the timer by writing to the TIMER_DISABLE register. To prevent the ongoing PWM pattern from being stopped in an unpredictable way, the timer does not stop immediately when the corresponding “1” has been written to the TIMER_DISABLE register. Rather, the write simply clears the enable latch and the timer still completes the ongoing PWM patterns gracefully. It stops cleanly at the end of the first period when the enable latch is cleared. During this final period the TIMEN bit returns “0”, but the TRUN bit still reads as a “1”.

If the TRUN bit is not cleared explicitly, and the enable latch can be cleared and re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself.

Figure 10-9 shows detailed timing.

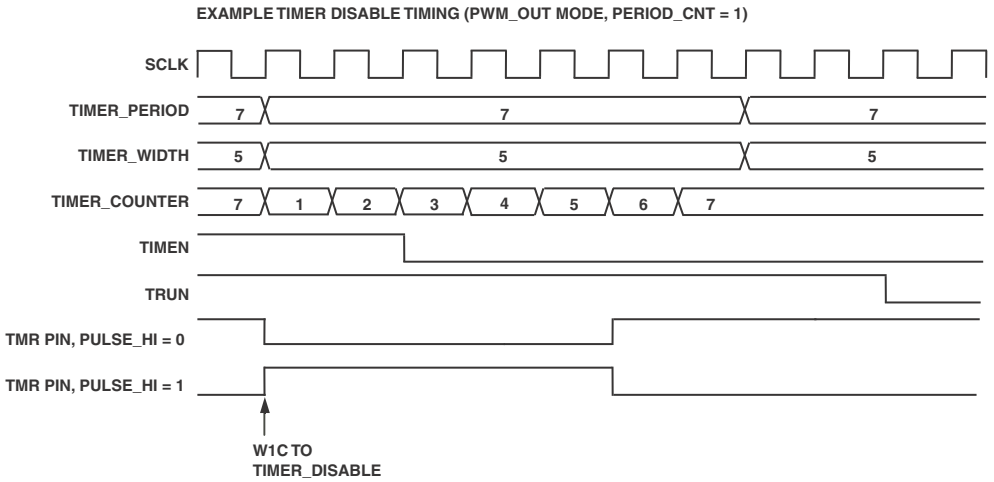


Figure 10-9. Timer Disable Timing

If necessary, the processor can force a timer in PWM_OUT mode to abort immediately. Do this by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMR pin.

When a timer is disabled, the TIMER_COUNTER register retains its state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The TIMER_COUNTER register is read-only. Software cannot overwrite or preset the timer counter value directly.

Pulse Width Count and Capture (WDTH_CAP) Mode

Use the WDTH_CAP mode, often simply called “capture mode,” to measure pulse widths on the TMR or TACI input pins, or to “receive” PWM signals. Figure 10-10 shows a flow diagram for WDTH_CAP mode.

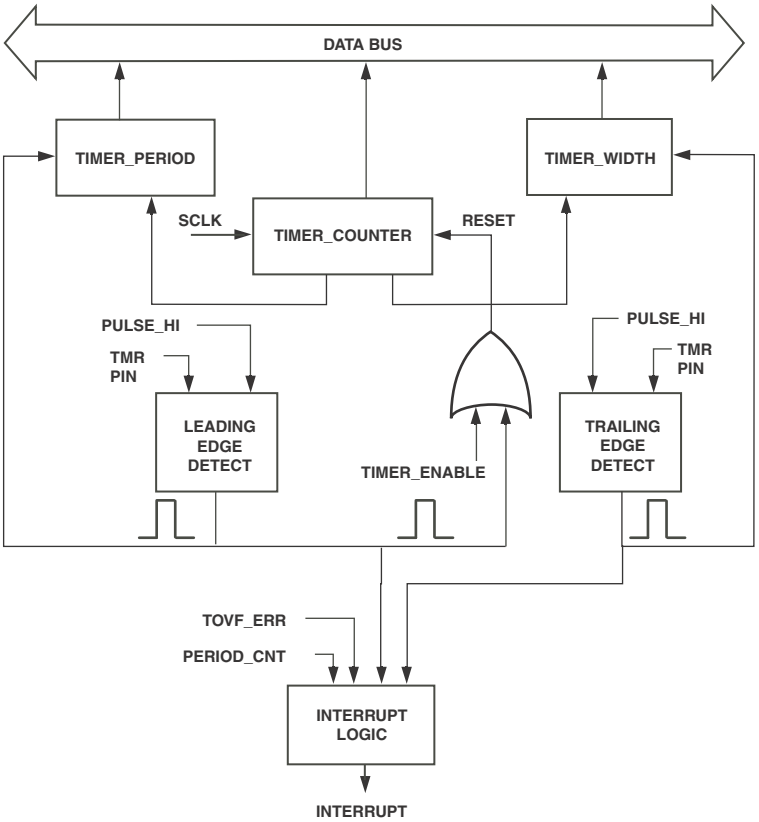


Figure 10-10. Timer Flow Diagram, WDTH_CAP Mode

In `WDTH_CAP` mode, the `TMR` pin is an input pin. The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the `TMODE` field to `b#10` in the `TIMER_CONFIG` register enables this mode.

When enabled in this mode, the timer resets the count in the TIMER_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMR pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMER_COUNTER register into the width buffer. At the next leading edge, the timer transfers the current 32-bit value of the TIMER_COUNTER register into the period buffer. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the TMR pin, the PULSE_HI bit in the TIMER_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the content of the counter register is captured to the pulse width buffer on the rising edge, and to the period buffer on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the counter value is captured to the pulse width buffer on the falling edge, and to the period buffer on the next rising edge.

In WDTH_CAP mode, these three events always occur at the same time:

1. The TIMER_PERIOD register is updated from the period buffer.
2. The TIMER_WIDTH register is updated from the width buffer.
3. The TIMIL bit gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMER_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these three events are called a measurement report. The TOVF_ERR bit does not get set at a measurement report. A measurement report occurs, at most, once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMER_PERIOD and TIMER_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (see Figure 10-11), then the TIMER_PERIOD and TIMER_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (see Figure 10-12), then the TIMER_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMER_PERIOD register reports the pulse period measured at the end of the previous period.

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMER_PERIOD value in this case returns “0”, as shown in Figure 10-12. To measure the pulse width of a waveform that has only one leading edge and one trailing edge, set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both TIMER_WIDTH and TIMER_PERIOD read “0” (because no measurement report occurred to copy the value captured in the width buffer to TIMER_WIDTH). See the first interrupt in Figure 10-13.

When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in

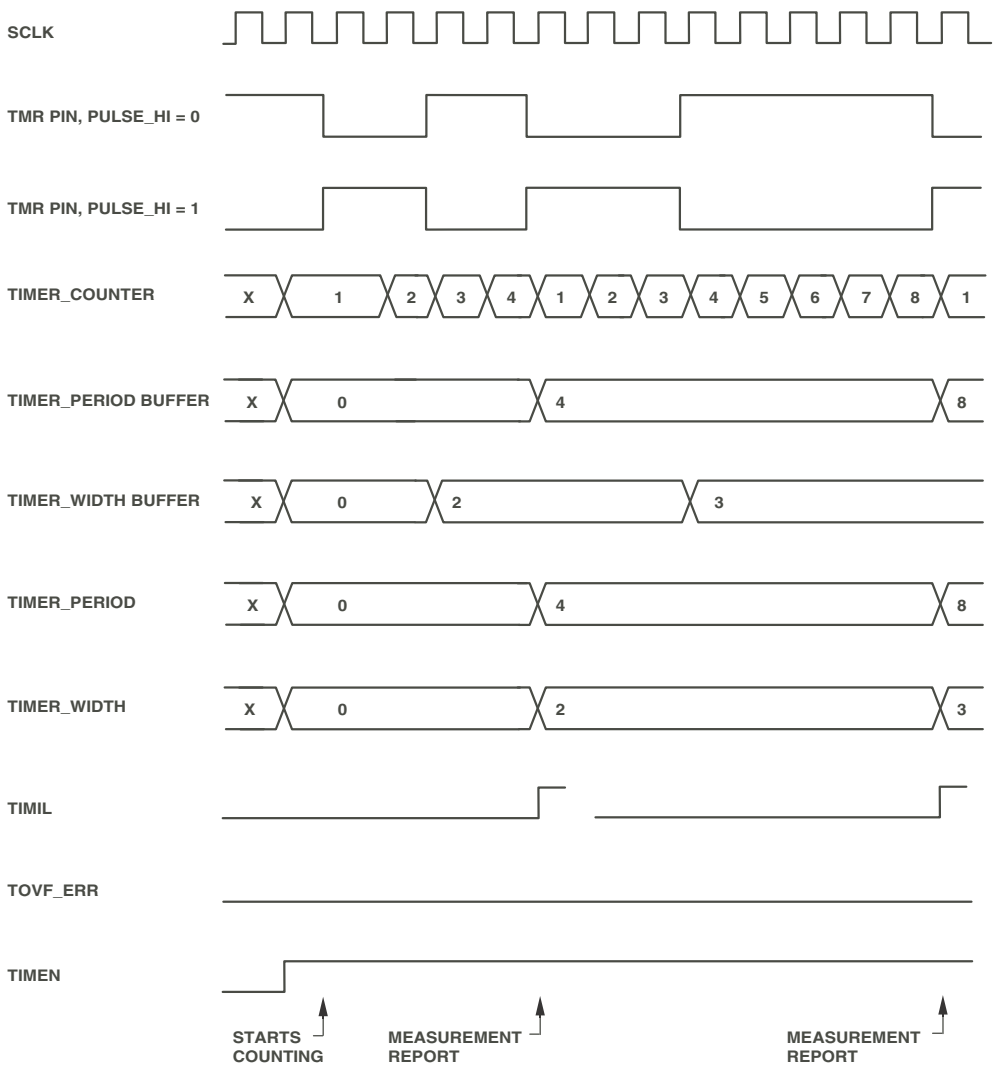


Figure 10-11. Example of Period Capture Measurement Report Timing (`WDTH_CAP` mode, `PERIOD_CNT` = 1)

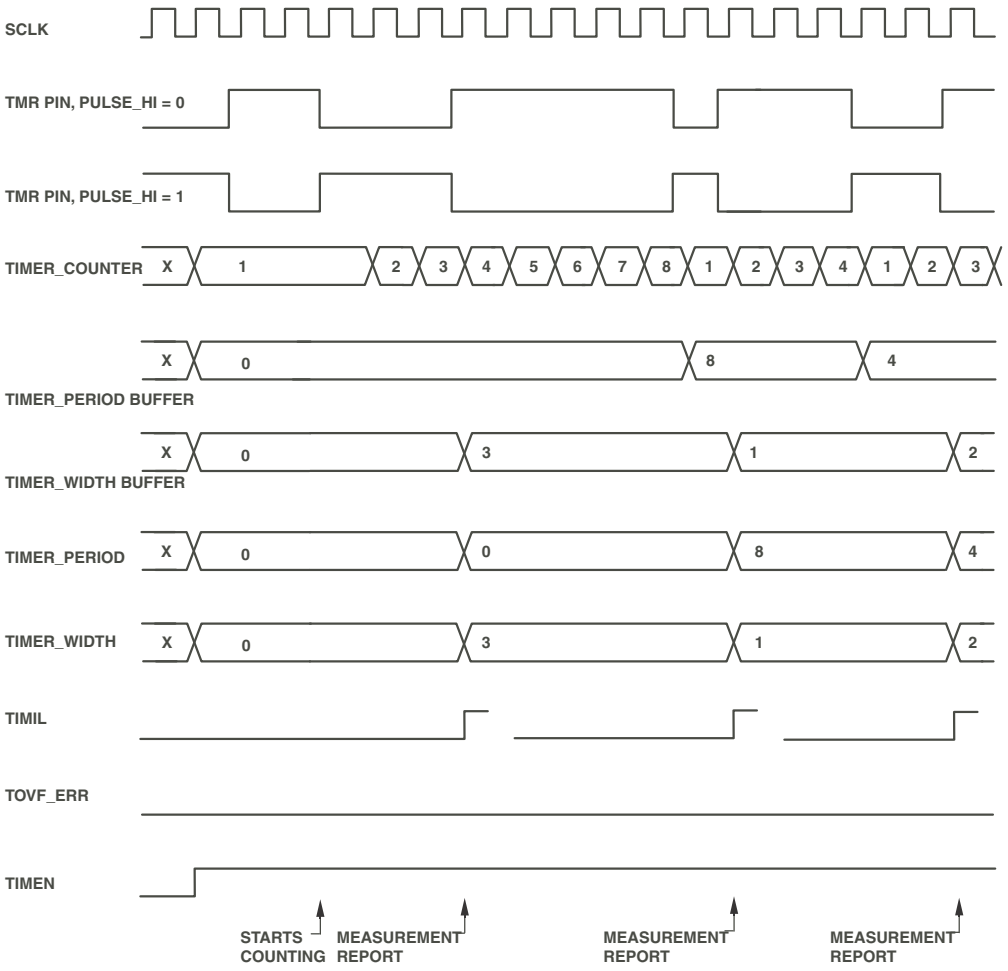
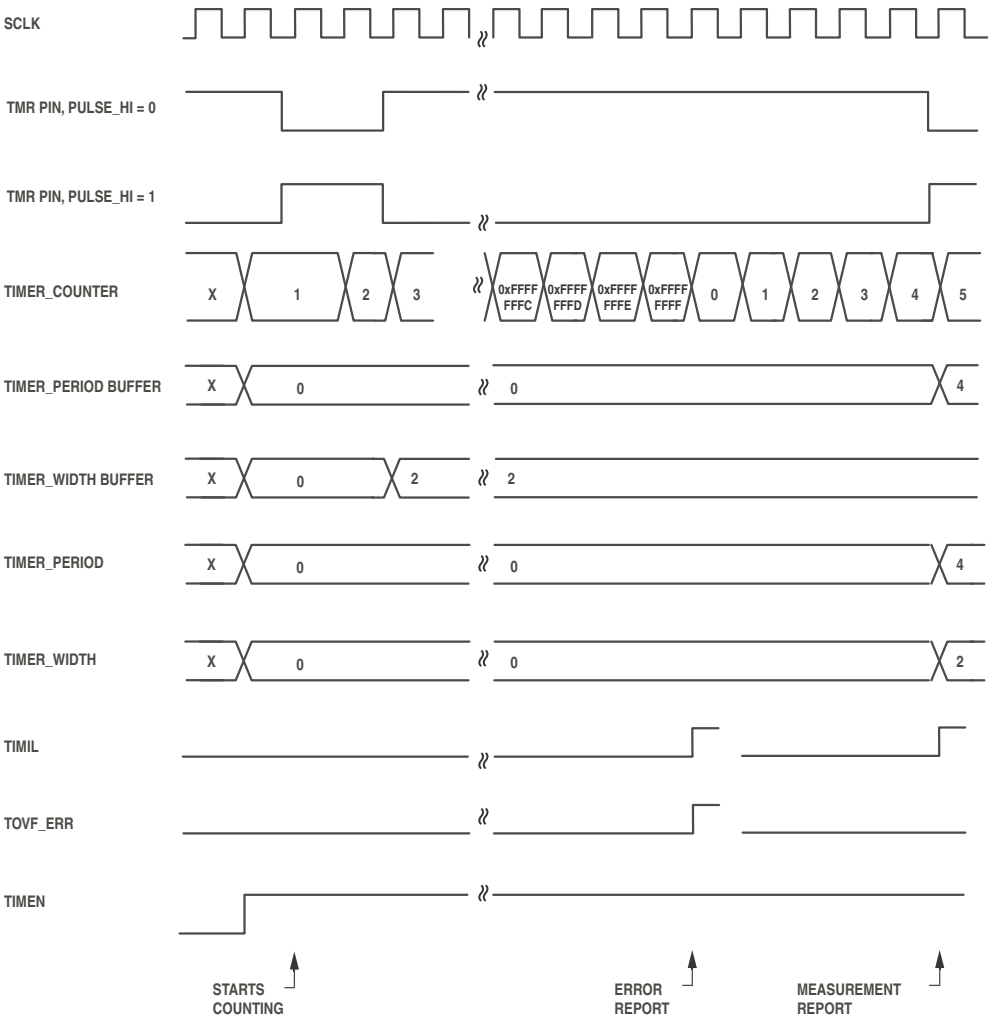


Figure 10-12. Example of Width Capture Measurement Report Timing (`WDTH_CAP` mode, `PERIOD_CNT` = 0)

preparation for another measurement. This procedure prevents the timer from free-running after the width measurement, and from logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to "0" in the absence of a leading edge. At that point, the TOVF_ERR bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMER_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMER_PERIOD and TIMER_WIDTH registers are never updated at the time an error is signaled.

Refer to Figure 10-13 and Figure 10-14 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMR PIN EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-13. Example Timing for Period Overflow Followed by Period Capture (`WDTH_CAP` mode, `PERIOD_CNT` = 1)

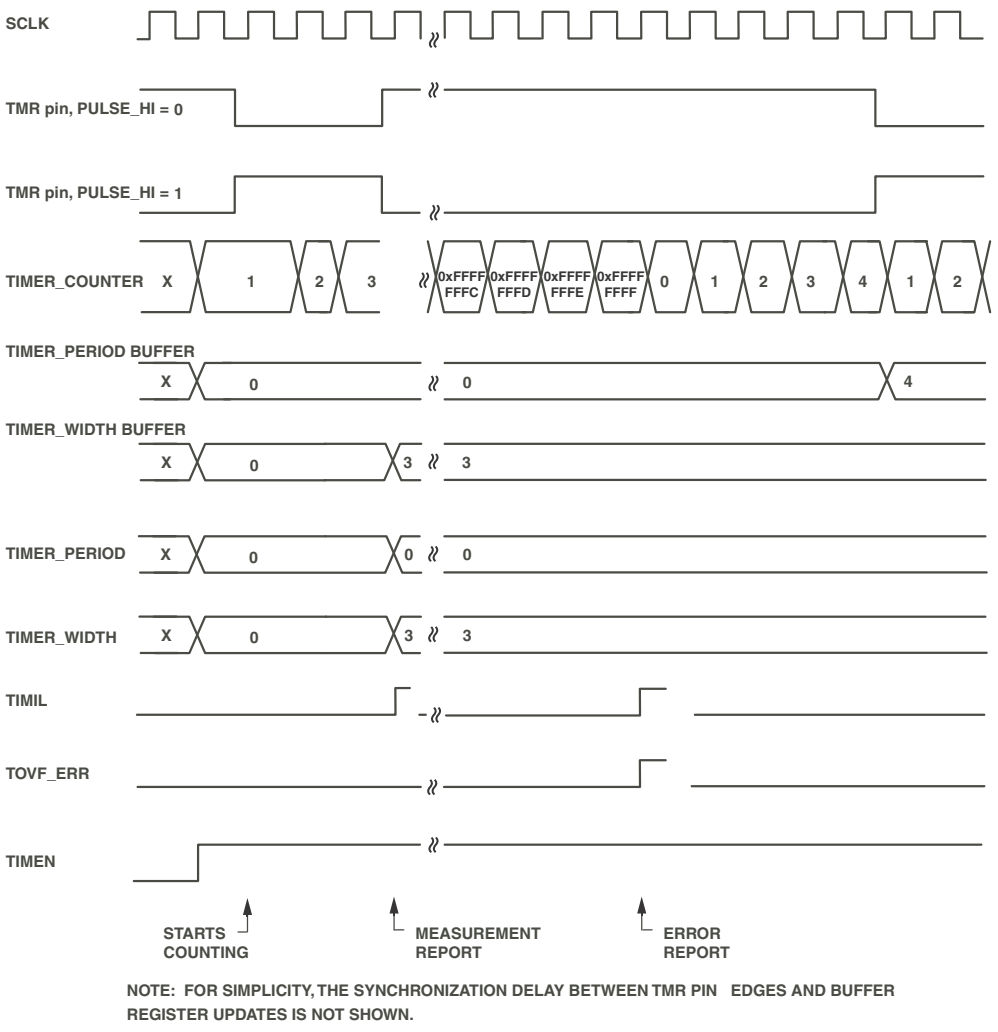


Figure 10-14. Example Timing for Width Capture Followed by Period Overflow (`WDTH_CAP` mode, `PERIOD_CNT` = 0)

Both TIMIL and TOVF_ERR are sticky bits, and software must explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMER_PERIOD nor the TIMER_WIDTH register were updated. If the timer overflowed and PERIOD_CNT = 0, the TIMER_PERIOD and TIMER_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 10-13 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR pin low time is one SCLK period and the minimum TMR pin high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure Period = 2 and Pulse Width = 1.

Autobaud Mode

On some devices, in WDTH_CAP mode, some of the timers can provide autobaud detection for the Universal Asynchronous Receiver/Transmitter (UART) interface(s). The TIN_SEL bit in the TIMER_CONFIG register causes the timer to sample the TACI pin instead of the TMR pin when enabled for WDTH_CAP mode. Autobaud detection can be used for initial bit rate negotiations as well as for detection of bit rate drifts while the interface is in operation.

External Event (EXT_CLK) Mode

Use the EXT_CLK mode (sometimes referred to as the counter mode) to count external events—that is, signal edges on the TMR pin (which is an

input in this mode). Figure 10-15 shows a flow diagram for EXT_CLK mode.

The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMER_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMER_CONFIG register enables this mode. The TIMER_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR low time is one SCLK period, and the minimum TMR high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

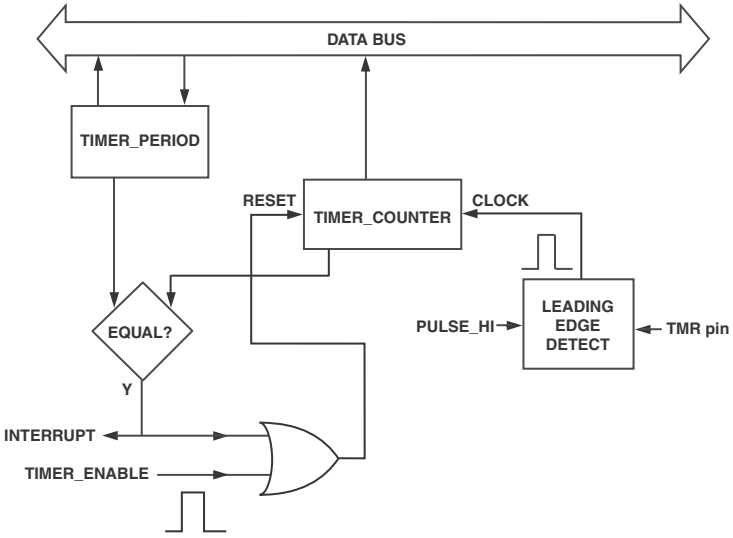


Figure 10-15. Timer Flow Diagram, EXT_CLK Mode

After the timer has been enabled, it resets the TIMER_COUNTER register to 0x0 and then waits for the first leading edge on the TMR pin. This edge

causes the TIMER_COUNTER register to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMIL bit is set, and an interrupt is generated. The next leading edge reloads the TIMER_COUNTER register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

The configuration bits TIN_SEL and PERIOD_CNT have no effect in this mode. The TOVF_ERR and ERR_TYP bits are set if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to “0” or if Period = “0” at startup or when the TIMER_COUNTER register rolls over (from Count = Period to Count = 0x1). The TIMER_WIDTH register is unused.

Programming Model

The architecture of the timer block enables any of the timers within this block to work individually or synchronously along with others as a group of timers. Regardless of the operating mode, the programming model is always straightforward. Because of the error checking mechanism, always follow this order when enabling timers:

1. Set timer mode.
2. Write TIMER_WIDTH and TIMER_PERIOD registers as applicable.
3. Enable timer.

If this order is not followed, the plausibility check may fail because of undefined width and period values, or writes to TIMER_WIDTH and TIMER_PERIOD may result in an error condition, because the registers are read-only in some modes. The timer may not start as expected.

If in PWM_OUT mode the PWM patterns of the second period differ from the patterns of the first one, the initialization sequence above might become:

1. Set timer mode to PWM_OUT.
2. Write first TIMER_WIDTH and TIMER_PERIOD value pair.
3. Enable timer.
4. Immediately write second TIMER_WIDTH and TIMER_PERIOD value pair.

Hardware ensures that the buffered width and period values become active when the first period expires.

Once started, timers require minimal interaction with software, which is usually performed by an interrupt service routine. In PWM_OUT mode software must update the pulse width and/or settings as required. In WDTH_CAP mode it must store captured values for further processing. In any case, the service routine should clear the TIMIL bits of the timers it controls.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of multiple identical timer units.

Each timer provides four registers:

- TIMER_CONFIG[15:0] – timer configuration register
- TIMER_WIDTH[31:0] – timer pulse width register
- TIMER_PERIOD[31:0] – timer period register
- TIMER_COUNTER[31:0] – timer counter register

Additionally, three registers are shared between the timers within a block:

- TIMER_ENABLE[15:0] – timer enable register
- TIMER_DISABLE[15:0] – timer disable register
- TIMER_STATUS[31:0] – timer status register

The size of accesses is enforced. A 32-bit access to a TIMER_CONFIG register or a 16-bit access to a TIMER_WIDTH, TIMER_PERIOD, or TIMER_COUNTER register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the TIMER_ENABLE, TIMER_DISABLE, and TIMER_STATUS registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

Timer Enable Register (TIMER_ENABLE)

Figure 10-16 shows an example of the TIMER_ENABLE register for a product with eight timers. The register allows simultaneous enabling of multiple timers so that they can run synchronously. For each timer there is a single W1S control bit. Writing a “1” enables the corresponding timer; writing a “0” has no effect. The bits can be set individually or in any combination. A read of the TIMER_ENABLE register shows the status of the enable for the corresponding timer. A “1” indicates that the timer is enabled. All unused bits return “0” when read.

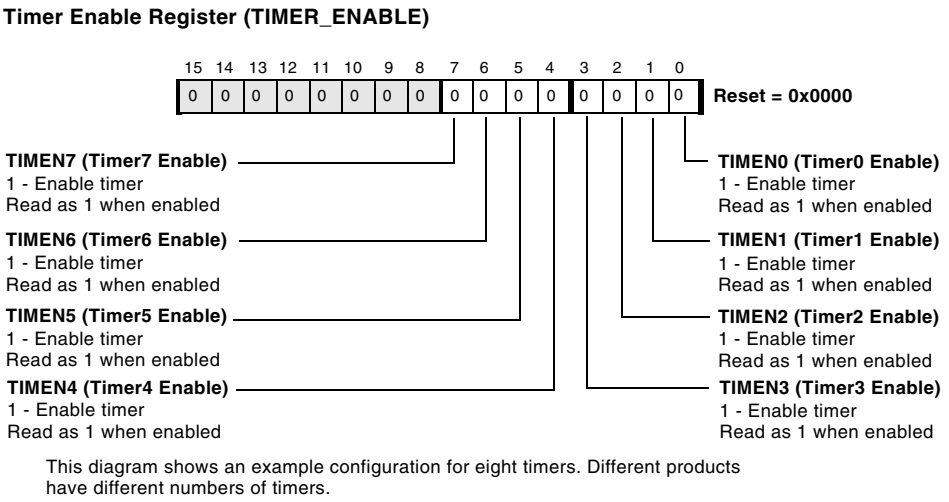
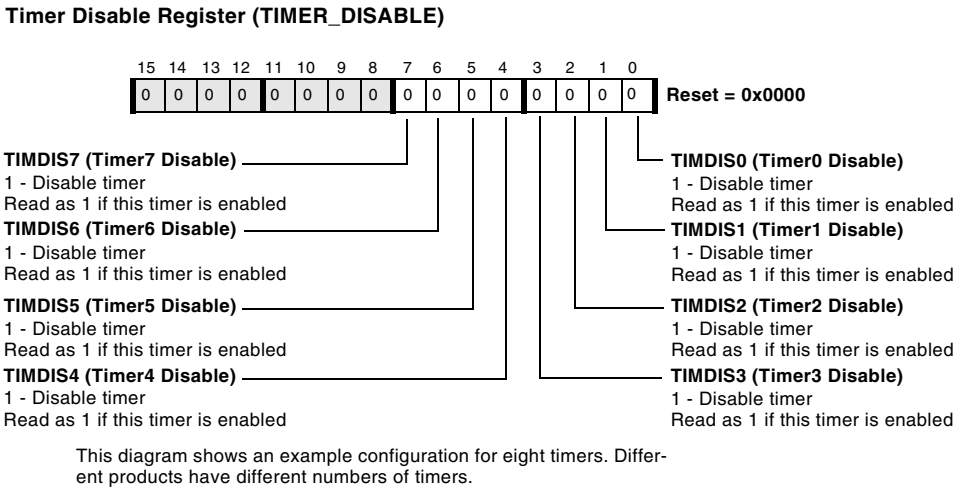


Figure 10-16. Timer Enable Register

Timer Disable Register (TIMER_DISABLE)

Figure 10-17 shows an example of the TIMER_DISABLE register for a product with eight timers. The register allows simultaneous disabling of multiple timers. For each timer there is a single W1C control bit. Writing a “1” disables the corresponding timer; writing a “0” has no effect. The bits can be cleared individually or in any combination. A read of the TIMER_DISABLE register returns a value identical to a read of the TIMER_ENABLE register. A “1” indicates that the timer is enabled. All unused bits return “0” when read.



This diagram shows an example configuration for eight timers. Different products have different numbers of timers.

Figure 10-17. Timer Disable Register

In PWM_OUT mode, a write of a "1" to TIMER_DISABLE does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 10-21.

In WDTH_CAP and EXT_CLK modes, a write of a "1" to TIMER_DISABLE stops the corresponding timer immediately.

Timer Status Register (TIMER_STATUS)

The TIMER_STATUS register indicates the status of the timers and is used to check the status of multiple timers with a single read. Status bits are sticky and W1C. The TRUN bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a TIMER_STATUS register read access, all reserved or unused bits return a "0". Figure 10-18 on page 10-39 shows an example of the TIMER_STATUS register for a product with eight timers.

For detailed behavior and usage of the TRUN bit see "Stopping the Timer in PWM_OUT Mode" on page 10-21. Writing the TRUN bits has no effect in other modes or when a timer has not been enabled. Writing the TRUN bits to "1" in PWM_OUT mode has no effect on a timer that has not first been disabled.

Error conditions are explained in "Illegal States" on page 10-7.

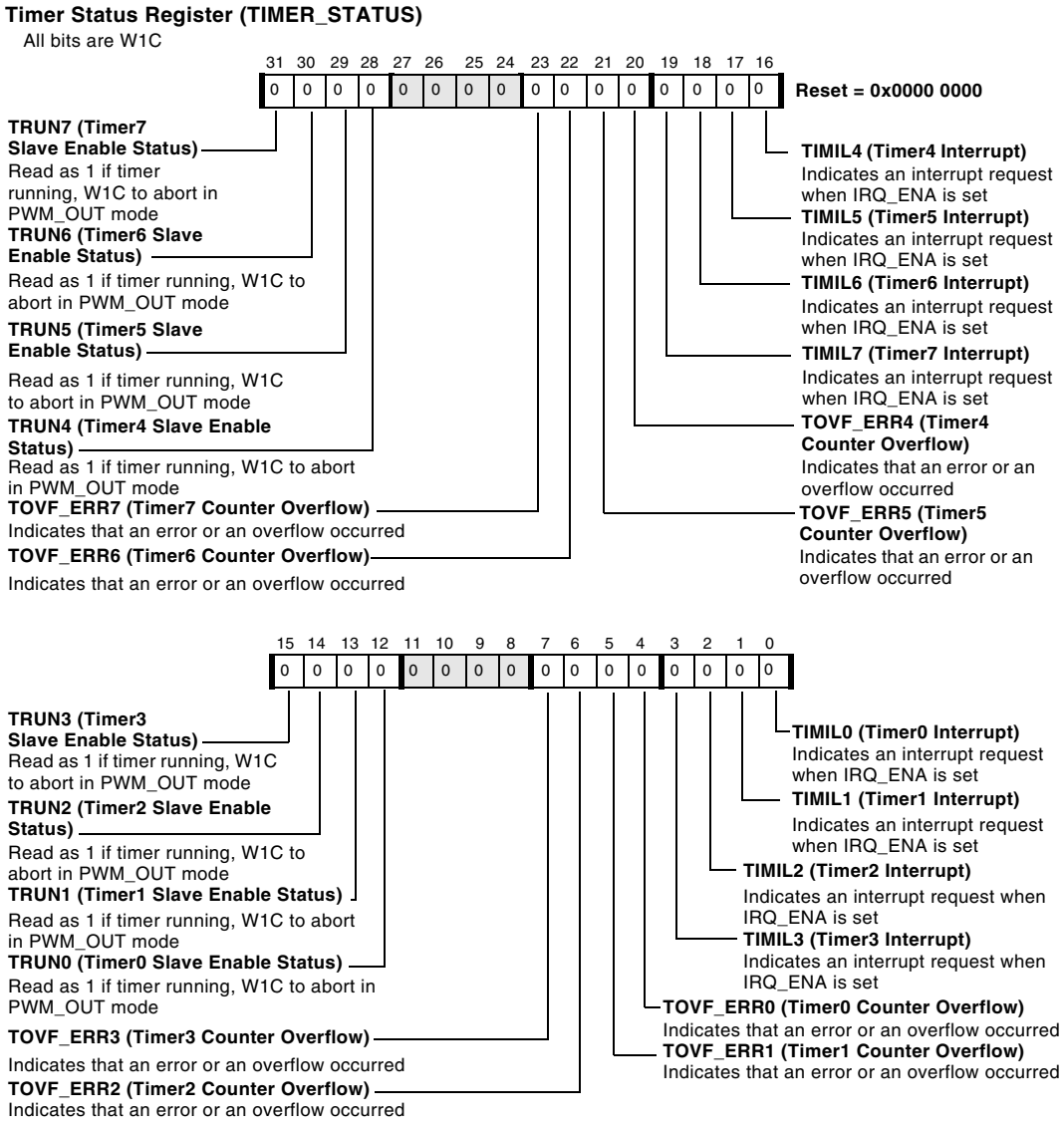


Figure 10-18. Timer Status Register

Timer Configuration Register (TIMER_CONFIG)

The operating mode for each timer is specified by its TIMER_CONFIG register. The TIMER_CONFIG register, shown in Figure 10-19, may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUN bit in TIMER_STATUS before attempting to reprogram TIMER_CONFIG. The TIMER_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled.

Each time TOVF_ERR is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 10-1 on page 10-9. The TIMER_CONFIG register also controls the behavior of the TMR pin, which becomes an output in PWM_OUT mode (TMODE = 01) when the OUT_DIS bit is cleared.

When operating the PPI in GP output modes with internal frame syncs, the CLK_SEL and the TIN_SEL bits for the timers involved must be set to “1”.

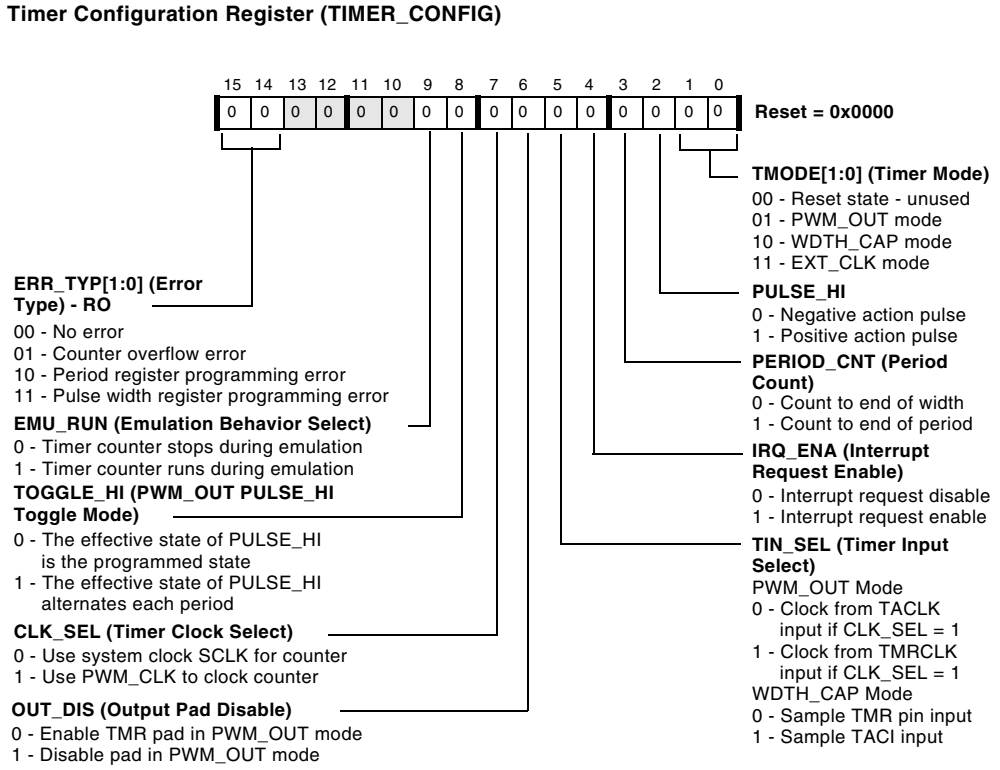


Figure 10-19. Timer Configuration Register

Timer Counter Register (TIMER_COUNTER)

This read-only register retains its state when disabled. When enabled, the TIMER_COUNTER register is reinitialized by hardware based on configuration and mode. The TIMER_COUNTER register, shown in Figure 10-20, may be read at any time (whether the timer is running or stopped), and it returns an atomic 32-bit value. Depending on the operating mode, the incrementing counter can be clocked by four different sources: SCLK, the TMR pin, the alternative timer clock pin TACLK, or the common TMRCLK pin, which is most likely used as the PPI clock (PPI_CLK).

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMER_COUNTER register also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMR pin waveform is "stretched"; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on the TMR pin may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMER_PERIOD and TIMER_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMER_CONFIG to enable this behavior.

Timer Counter Register (TIMER_COUNTER)

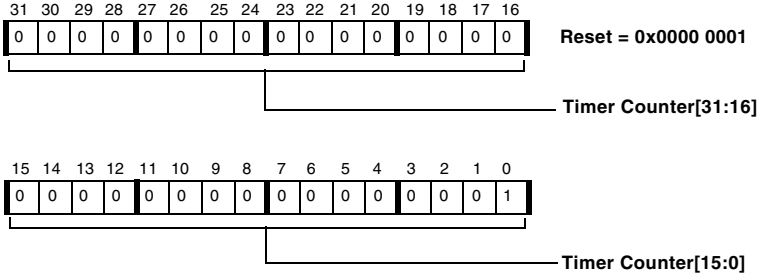


Figure 10-20. Timer Counter Register

Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers

When a timer is enabled and running, and the software writes new values to the TIMER_PERIOD register and the TIMER_WIDTH register, the writes are buffered and do not update the registers until the end of the current period (when TIMER_COUNTER equals TIMER_WIDTH).

Usage of the TIMER_PERIOD register, shown in Figure 10-21, and the TIMER_WIDTH register, shown in Figure 10-22, varies depending on the mode of the timer:

- In PWM_OUT mode, both the TIMER_PERIOD and TIMER_WIDTH register values can be updated “on-the-fly” since the values change simultaneously.
- In WDTH_CAP mode, the timer period and timer pulse width buffer values are captured at the appropriate time. The TIMER_PERIOD and TIMER_WIDTH registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In EXT_CLK mode, the TIMER_PERIOD register is writable and can be updated “on-the-fly.” The TIMER_WIDTH register is not used.

If new values are not written to the TIMER_PERIOD register or the TIMER_WIDTH register, the value from the previous period is reused. Writes to the 32-bit TIMER_PERIOD register and TIMER_WIDTH register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the TIMER_PERIOD registers or TIMER_WIDTH registers are always stored in the buffer registers. Reads from the TIMER_PERIOD or TIMER_WIDTH registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the TIMER_PERIOD and TIMER_WIDTH registers are updated from their respective buffers at the end of the current period. See Figure 10-1 on page 10-3.

When the timer is disabled, writes to the buffer registers are immediately copied through to the TIMER_PERIOD or TIMER_WIDTH register so that they will be ready for use in the first timer period. For example, to change the values for the TIMER_PERIOD and/or TIMER_WIDTH registers in order to use a

different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In PWM_OUT mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the TIMER_PERIOD register and the TIMER_WIDTH register. The next period may use one old value and one new value. In order to prevent "pulse width ≥ period" errors, write the TIMER_WIDTH register before the TIMER_PERIOD register when decreasing the values, and write the TIMER_PERIOD register before the TIMER_WIDTH register when increasing the value.

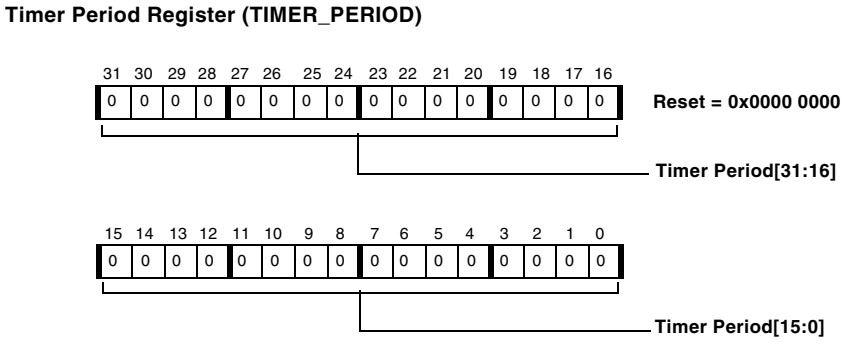


Figure 10-21. Timer Period Register

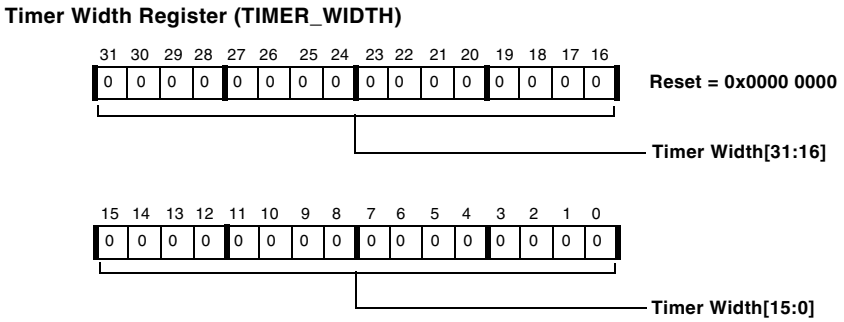


Figure 10-22. Timer Width Register

Summary

Table 10-2 summarizes control bit and register usage in each timer mode.

Table 10-2. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt

Table 10-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count TMRCLK clocks 0 - Count TACLK clocks If CLK_SEL = 0, Unused	1 - Select TACI input 0 - Select TMR pin input	Unused
OUT_DIS	1 - Disable TMR pin 0 - Enable TMR pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports `b#00`, `b#01`, `b#10`, or `b#11`, as appropriate	Reports `b#00` or `b#01`, as appropriate	Reports `b#00`, `b#01`, or `b#10`, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused

Table 10-2. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on TMR pin event
TRUN	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect
TOVF_ERR	Set at startup or rollover if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or rollover if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Programming Examples

Listing 10-1 configures the port control registers in a way that enables TMR pins associated with Port G. This example assumes TMR1-7 are connected to Port G bits 5–11.

Listing 10-1. Port Setup

```
timer_port_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(PORTG_FER);
    p5.l = lo(PORTG_FER);
    r7.l = PG5|PG6|PG7|PG8|PG9|PG10|PG11;
    w[p5] = r7;
    p5.l = lo(PORTG_MUX);
    r7.l = PFTE;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer_port_setup.end:
```

Listing 10-2 generates signals on the TMR4 and TMR5 outputs. By default, timer 5 generates a continuous PWM signal with a duty cycle of 50% (period = 0x40 SCLKs, width = 0x20 SCLKs) while the PWM signal generated by timer 4 has the same period but 25% duty cycle (width = 0x10 SCLKs).

If the preprocessor constant SINGLE_PULSE is defined, every TMR pin outputs only a single high pulse of 0x20 (timer 4) and 0x10 SCLKs (timer 5) duration.

In any case the timers are started synchronously and the rising edges are aligned. That is, the pulses are left aligned.

Listing 10-2. Signal Generation

```
// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7 = 0x20 (z);
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
#ifndef SINGLE_PULSE
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. Thus, Listing 10-3 illustrates how interrupts are generated and how interrupt service routines can be registered. In this example, the timer 5 interrupt is assigned to the IVG12 interrupt channel of the CEC controller.

Listing 10-3. Interrupt Setup

```
timer5_interrupt_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(IMASK);
    p5.l = lo(IMASK);
/* register interrupt service routine */
    r7.h = hi(isr_timer5);
    r7.l = lo(isr_timer5);
    [p5 + EVT12 - IMASK] = r7;
/* unmask IVG12 in CEC */
    r7 = [p5];
    bitset(r7, bitpos(EVT_IVG12));
    [p5] = r7;
/* assign timer 5 IRQ (= IRQ37 in this example) to IVG12 */
    p5.h = hi(SIC_IAR4);
    p5.l = lo(SIC_IAR4);
/*SIC_IAR register mapping is processor dependent*/
    r7.h = 0xFF5F;
    r7.l = 0xFFFF;
    [p5] = r7;
/* enable timer 5 IRQ */
    p5.h = hi(SIC_IMASK1);
    p5.l = lo(SIC_IMASK1);
/*SIC_IMASK register mapping is processor dependent*/
    r7 = [p5];
    bitset(r7, 5);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:
```

The example shown in Listing 10-4 does not drive the TMR pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant SINGLE_PULSE was defined, timer 5 requests an interrupt only once. Unlike in a real application, the purpose of the interrupt service routine shown in this example is just the clearing of the interrupt request and counting interrupt occurrences.

Listing 10-4. Periodic Interrupt Requests

```
// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x1000 (z);
#ifndef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    r7 = 0x1 (z);
#endif
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    r0 = 0 (z);
    rts;
timer5_interrupt_generation.end:
isr_timer5:
    [--sp] = astat;
```

```
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 10-5 illustrates how two timers can generate two non-overlapping clock pulses as typically required for break-before-make scenarios. Both timers are running in `PWM_OUT` mode with `PERIOD_CNT` = 1 and `PULSE_HI` = 1.

Figure 10-23 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 10-3 summarizes the register writes.

Table 10-3. Register Writes for Non-Overlapping Clock Pulses

Register	Before Enable	After Enable	At IRQ1	At IRQ2
TIMER5_PERIOD	P/2			
TIMER5_WIDTH	P/2 - W/2	W/2	P/2 - W/2	W/2
TIMER4_PERIOD	P	P/2		
TIMER4_WIDTH	P - W/2		W/2	P/2 - W-2

Since hardware only updates the written period and width values at the end of periods, software can write new values immediately after the timers have been enabled. Note that both timers' period expires at exactly the

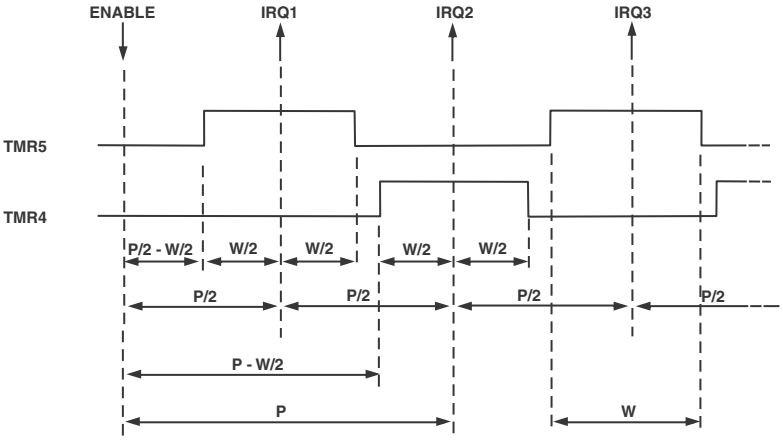


Figure 10-23. Non-Overlapping Clock Pulses

same times with the exception of the first timer 5 interrupt (at `IRQ1`) which is not visible to timer 4.

Listing 10-5. Non-Overlapping Clock Pulses

```
#define P 0x1000    /* signal period */
#define W 0x0600    /* signal pulse width */
#define N 4         /* number of pulses before disable */
timer45_toggle_hi:
    [--sp] = (r7:1, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
/* calculate timers widths and period */
    r0.l = lo(P);
```

```
    r0.h = hi(P);
    r1.l = lo(W);
    r1.h = hi(W);
    r2 = r1 >> 1;    /* W/2 */
    r3 = r0 >> 1;    /* P/2 */
    r4 = r3 - r2;    /* P/2 - W/2 */
    r5 = r0 - r2;    /* P - W/2 */
/* write values for initial period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r4;
/* start timers */
    r7.l = TIMEN5 | TIMEN4 ;
    w[p5 + TIMER_ENABLE - TIMER_ENABLE] = r7;
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r2;
/* r0 functions as signal period counter */
    r0.h = hi(N * 2 - 1);
    r0.l = lo(N * 2 - 1);
    (r7:1, p5:5) = [sp++];
    rts;
timer45_toggle_hi.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:5, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS - TIMER_ENABLE] = r7;
/* toggle width values (width = period - width) */
```

```
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE];
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
/* disable after a certain number of periods */
    r0+= -1;
    CC = r0 == 0;
    r5.l = 0;
    r7.l = TIMDIS5 | TIMDIS4;
    if !CC r7 = r5;
    w[p5 + TIMER_DISABLE - TIMER_ENABLE] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 10-5 generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern anyhow.

Listing 10-6 configures timer 5 in `WDTH_CAP` mode. If looped back externally, this code might be used to receive N PWM patterns generated by one of the other timers. Ensure that the PWM generator and consumer both use the same `PERIOD_CNT` and `PULSE_HI` settings.

Listing 10-6. Timer Configured in WDTH_CAP Mode

```
.section L1_data_a;
.align 4;
#define N 1024
.var buffReceive[N*2];
```

```
.section L1_code;
timer5_capture:
    [--sp] = (r7:7, p5:5);
/* setup DAG2 */
    r7.h = hi(buffReceive);
    r7.l = lo(buffReceive);
    i2 = r7;
    b2 = r7;
    l2 = length(buffReceive)*4;
/* config timer for high pulses capture */
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5 + TIMER_ENABLE - TIMER_ ENABLE] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer5_capture.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
/* clear interrupt request first */
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMER5_PERIOD - TIMER_STATUS];
    [i2++] = r7;
    r7 = [p5 + TIMER5_WIDTH - TIMER_STATUS];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
```

```
    astat = [sp++];
    rti;
isr_timer5.end:
```

Unique Information for the ADSP-BF50x Processor

The ADSP-BF50x processor features one general-purpose timer module that contains eight identical 32-bit timers. Each timer can be individually configured to operate in various modes. Although the timers operate completely independently of each other, all of them can be started and stopped simultaneously for synchronous operation.

Interface Overview

Figure 10-24 shows the ADSP-BF50x specific block diagram of the general-purpose timer module.

External Interface

The TMRCLK input is common to all eight timers. The PPI unit is clocked by the same pin; therefore any of the timers can be clocked by PPI_CLK. Since timer 0 and timer 1 are often used in conjunction with the PPI, they are internally looped back to the PPI module for frame sync generation.

The timer signals TMR0 and TMR1 are multiplexed with the PPI frame syncs when the frame syncs are applied externally. PPI modes requiring only one frame sync free up TMR1. For details, see Chapter 20, “Parallel Peripheral Interface”.

If the PPI frame syncs are applied externally, timer 0 and timer 1 are still fully functional and can be used for other purposes not involving the TMRx pins. Timer 0 and timer 1 must not drive their TMR0 and TMR1 pins. If operating in PWM_OUT mode, the OUT_DIS bit in the TIMER0_CONFIG and TIMER1_CONFIG registers must be set.

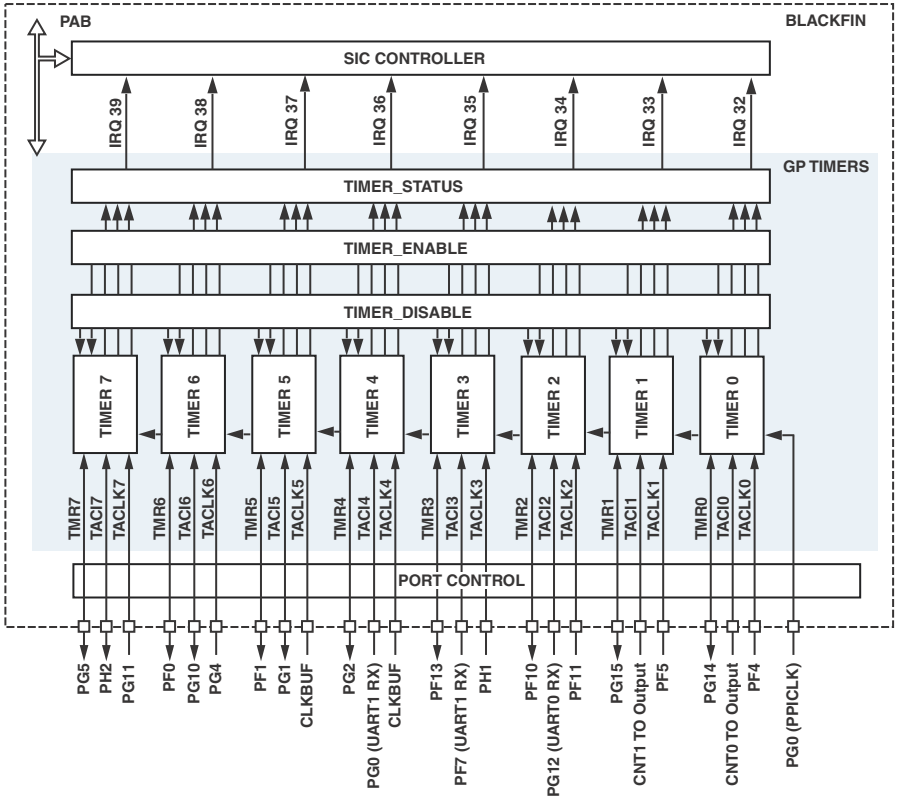


Figure 10-24. Timer Block Diagram

11 CORE TIMER

This chapter describes the core timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with a programming example.

Specific Information for the ADSP-BF50x

For details regarding the number of core timers for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For Core Timer interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

For a list of MMR addresses for each Core Timer, refer to Chapter A, "System MMR Assignments".

Core timer behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF50x Processor" on page 11-9.

Overview and Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Unlike other peripherals, the core timer resides

inside the Blackfin core and runs at the core clock (CCLK) rate. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

Figure 11-1 provides a block diagram of the core timer.

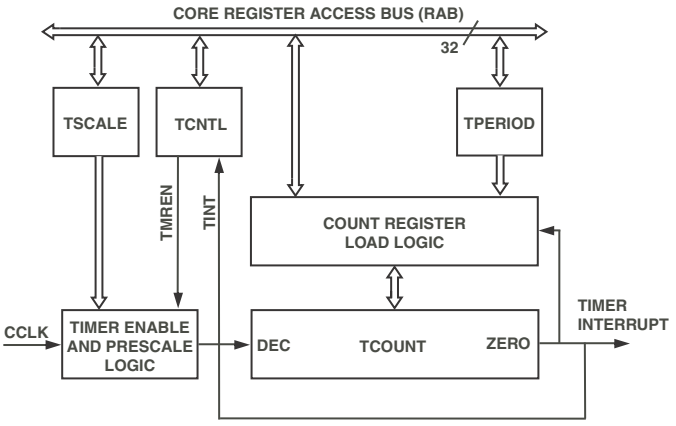


Figure 11-1. Core Timer Block Diagram

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The module is clocked by the core clock CCLK. The timer's dedicated interrupt request is a higher priority than requests from all other peripherals.

Description of Operation

The software should initialize the TCOUNT register *before* the timer is enabled. The TCOUNT register can be written directly, but writes to the TPERIOD register are also passed through to TCOUNT.

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every time the prescaler TSCALE expires, that is, every TSCALE + 1 number of CCLK clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register.

If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. If the TAUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN is set when TMPWR = 0.

Interrupt Processing

The timer's dedicated interrupt request is a higher priority than requests from all other peripherals. The request goes directly to the core event controller (CEC) and does not pass through the system interrupt controller

(SIC). Therefore, the interrupt processing is also completely in the CCLK domain.

The core timer interrupt request is edge-sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module doesn't provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Core Timer Registers

The core timer includes four core memory-mapped registers, the timer control register (TCNTL), the timer count register (TCOUNT), the timer period register (TPERIOD), and the timer scale register (TSCALE). As with all core MMRs, these registers are always accessed by 32-bit read and write operations.

Core Timer Control Register (TCNTL)

The TCNTL register, shown in Figure 11-2, functions as control and status register.

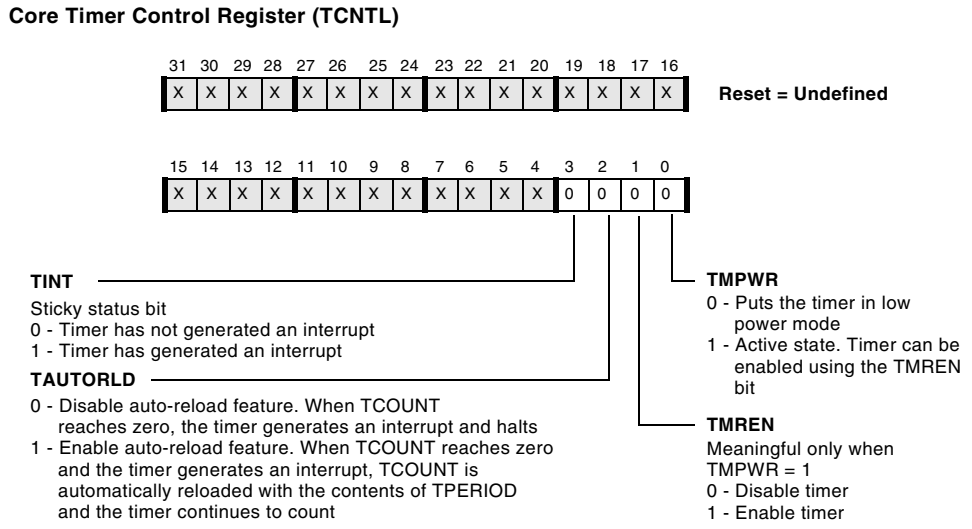


Figure 11-2. Core Timer Control Register

Core Timer Count Register (TCOUNT)

The TCOUNT register, shown in Figure 11-3, decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register. Nevertheless, the TCOUNT register can be written directly. In auto reload mode the value written to TCOUNT may differ from the TPERIOD value to let the initial period be shorter or longer than following periods. To do this, write to TPERIOD first and overwrite TCOUNT afterward.

Writes to TCOUNT are ignored once the timer is running.

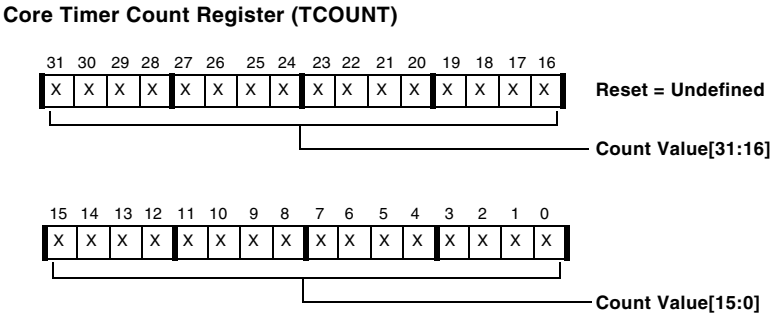


Figure 11-3. Core Timer Count Register

Core Timer Period Register (TPERIOD)

The TPERIOD register is shown in Figure 11-4. When auto-reload is enabled, the TCOUNT register is reloaded with the value of the TPERIOD register whenever TCOUNT reaches 0. Writes to TPERIOD are ignored when the timer is running.

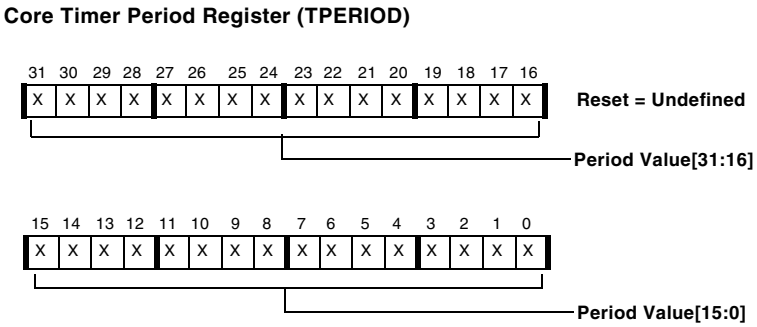


Figure 11-4. Core Timer Period Register

Core Timer Scale Register (TSCALE)

The TSCALE register is shown in Figure 11-5. The register stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every CCLK clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

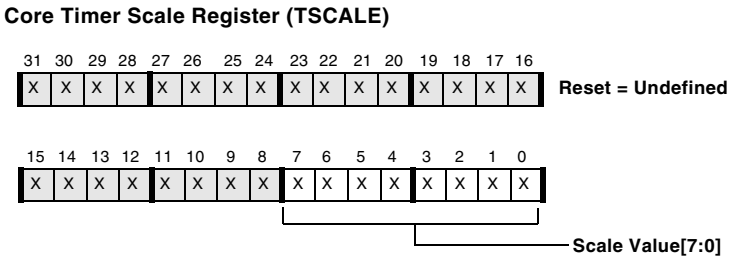


Figure 11-5. Core Timer Scale Register

Programming Examples

Listing 11-1 configures the core timer in auto-reload mode. Assuming a CCLK of 500 MHz, the resulting period is 1 second. The initial period is twice as long as the others.

Listing 11-1. Core Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
.section L1_code;
.global _main;
_main:
/* Register service routine at EVT6 and unmask interrupt */
    p1.l = lo(IMASK);
    p1.h = hi(IMASK);
```

```
    r0.l = lo(isr_core_timer);
    r0.h = hi(isr_core_timer);
    [p1 + EVT6 - IMASK] = r0;
    r0 = [p1];
    bitset(r0, bitpos(EVT_IVTMR));
    [p1] = r0;
/* Prescaler = 50, Period = 10,000,000, First Period = 20,000,000
*/
    p1.l = lo(TCNTL);
    p1.h = hi(TCNTL);
    r0 = 50 (z);
    [p1 + TSCALE - TCNTL] = r0;
    r0.l = lo(10000000);
    r0.h = hi(10000000);
    [p1 + TPERIOD - TCNTL] = r0;
    r0 <<= 1;
    [p1 + TCOUNT - TCNTL] = r0;
/* R6 counts interrupts */
    r6 = 0 (z);
/* start in auto-reload mode */
    r0 = TAUTORLD | TMPWR | TMREN (z);
    [p1] = r0;
_main.forever:
    jump _main.forever;
_main.end:
/* interrupt service routine simple increments R6 */
isr_core_timer:
    [--sp] = astat;
    r6+= 1;
    astat = [sp++];
    rti;
isr_core_timer.end:
```

Unique Information for the ADSP-BF50x Processor

None.

12 WATCHDOG TIMER

This chapter describes the watchdog timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of watchdog timers for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For Watchdog Timer interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

For a list of MMR addresses for each Watchdog Timer, refer to Chapter A, "System MMR Assignments".

Watchdog timer behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF50x Processor" on page 12-11.

Overview and Features

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the watchdog expires before being updated by software.

Watchdog timer key features include:

- 32-bit watchdog timer
- 8-bit disable bit pattern
- System reset on expire option
- NMI on expire option
- General-purpose interrupt option

Typically, the watchdog timer is used to supervise stability of the system software. When used in this way, software reloads the watchdog timer in a regular manner so that the downward counting timer never expires (never becomes 0). An expiring timer then indicates that system software might be out of control. At this point a special error handler may recover the system. For safety, however, it is often better to reset and reboot the system directly by hardware control.

Especially in slave boot configurations, a processor reset cannot automatically force the Blackfin device to be rebooted. In this case, the processor may reset without booting again and may negotiate with the host device by the time program execution starts. Alternatively, a watchdog event can cause an NMI event. The NMI service routine may request the host device reset and/or reboot the Blackfin processor.

The watchdog timer is often programmed to let the processor wake up from sleep mode after a programmable period of time.

For easier debugging, the watchdog timer does not decrement (even if enabled) when the processor is in emulation mode.

Interface Overview

Figure 12-1 provides a block diagram of the watchdog timer.

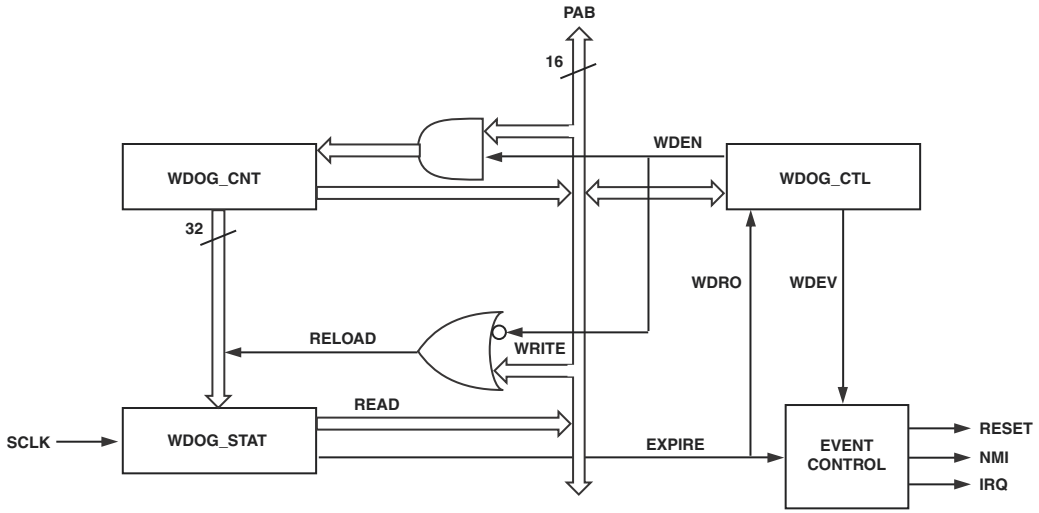


Figure 12-1. Watchdog Timer Block Diagram

External Interface

The watchdog timer does not directly interact with any pins of the chip.

Internal Interface

The watchdog timer is clocked by the system clock `SCLK`. Its registers are accessed through the 16-bit peripheral access bus (PAB). The 32-bit registers `WDOG_CNT` and `WDOG_STAT` must always be accessed by 32-bit read/write operations. Hardware ensures that those accesses are atomic.

When the counter expires, one of three event requests can be generated. Either a reset or an NMI request is issued to the core event controller (CEC) or a general-purpose interrupt request is passed to the system interrupt controller (SIC).

Description of Operation

If enabled, the 32-bit watchdog timer counts downward every SCLK cycle. If it becomes 0, one of three event requests can be issued to either the CEC or the SIC. Depending on how the WDEV bit field in the WDOG_CTL register is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt.

The counter value can be read through the 32-bit WDOG_STAT register. The WDOG_STAT register cannot, however, be written directly. Rather, software writes the watchdog period value into the 32-bit WDOG_CNT register *before* the watchdog is enabled. Once the watchdog is started, the period value cannot be altered.

To start the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (WDOG_CNT). Since the watchdog timer is not enabled yet, the write to the WDOG_CNT registers automatically pre-loads the WDOG_STAT register as well.
2. In the watchdog control register (WDOG_CTL), select the event to be generated upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register.

If software does not service the watchdog in time, WDOG_STAT continues decrementing until it reaches 0. Then, the programmed event is generated. The counter stops decrementing and remains at zero. Additionally,

the WDRO latch bit in the WDOG_CTL register is set and can be interrogated by software in case event generation is not enabled.

When the watchdog is programmed to generate a reset, it resets the processor core and peripherals. If the NOBOOT bit in the SYSCR register was set by the time the watchdog resets the part, the chip is not rebooted. This is recommended behavior in slave boot configurations. The reset handler may evaluate the RESET_WDOG bit in the software reset register SWRST to detect a reset caused by the watchdog. For details, see "System Reset and Booting" on page 24-1.

To prevent the watchdog from expiring, software services the watchdog by performing dummy writes to the WDOG_STAT register. The values written are ignored, but the write commands cause the WDOG_STAT register to be reloaded from the WDOG_CNT register.

If the watchdog is enabled with a zero value loaded to the counter and the WDRO bit was cleared, the WDRO bit of the watchdog control register is set immediately and the counter remains at zero without further decrements. If, however, the WDRO bit was set by the time the watchdog is enabled, the counter decrements to 0xFFFF FFFF and continues operation.

Software can disable the watchdog timer only by writing a 0xAD value to the WDEN field in the WDOG_CTL register.

Register Definitions

The watchdog timer is controlled by three registers.

Watchdog Count (WDOG_CNT) Register

The WDOG_CNT register, shown in Figure 12-2, holds the 32-bit unsigned count value. The WDOG_CNT register must always be accessed with 32-bit read/writes.

A valid write to the WDOG_CNT register also preloads the watchdog counter. For added safety, the WDOG_CNT register can be updated only when the watchdog timer is disabled. A write to the WDOG_CNT register while the timer is enabled does not modify the contents of this register.

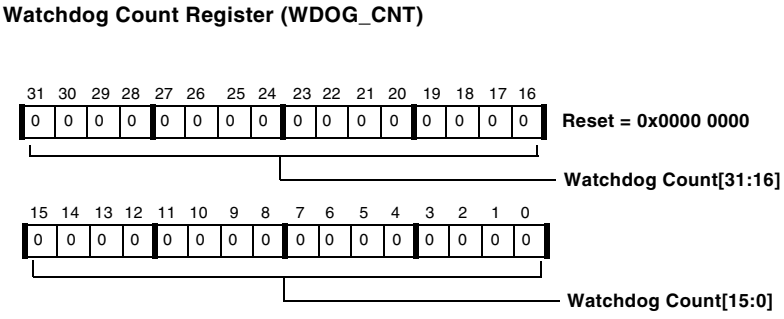


Figure 12-2. Watchdog Count Register

Watchdog Status (WDOG_STAT) Register

The 32-bit WDOG_STAT register, shown in Figure 12-3, contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, but not rolled over yet, writes to the WDOG_STAT register load it with the value in WDOG_CNT.

Enabling the watchdog timer does not automatically reload WDOG_STAT from WDOG_CNT.

The WDOG_STAT register is a 32-bit unsigned system MMR that must be accessed with 32-bit reads and writes.

Watchdog Status Register (WDOG_STAT)

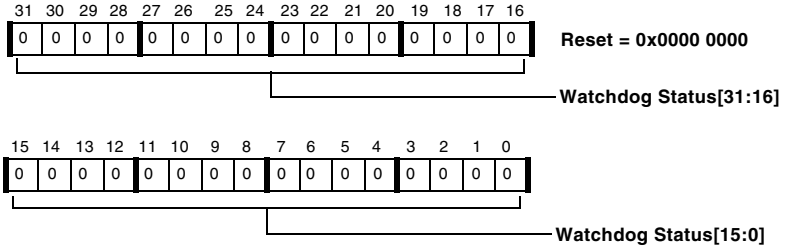


Figure 12-3. Watchdog Status Register

Watchdog Control (WDOG_CTL) Register

The WDOG_CTL register, shown in Figure 12-4, is a 16-bit system MMR used to control the watchdog timer.

The watchdog event (WDEV[1:0]) bit field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the SIC_IMASK register that holds the watchdog timer mask bit should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The watchdog enable (WDEN[7:0]) bit field is used to enable and disable the watchdog timer. Writing any value other than the disable key (0xAD) into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the watchdog has expired by interrogating the WDRO status bit of the WDOG_CTL register. This is a sticky bit that is

set whenever the watchdog timer count reaches 0. It can be cleared only by writing a “1” to the bit when the watchdog has been disabled first.

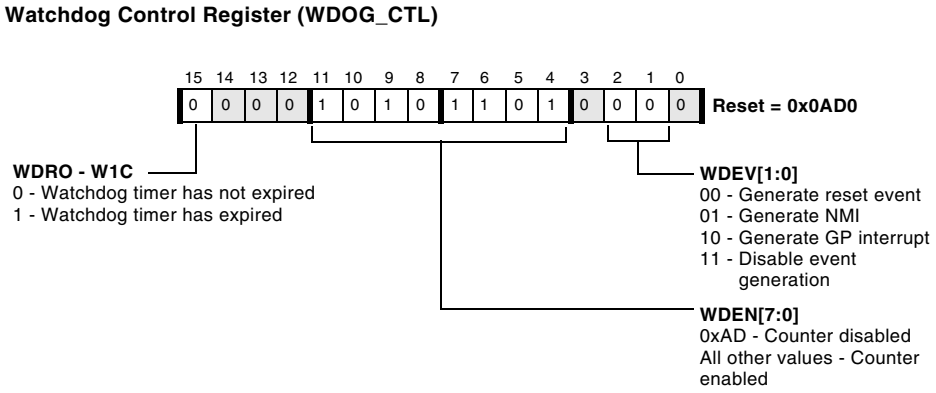


Figure 12-4. Watchdog Control Register

Programming Examples

Listing 12-1 shows how to configure the watchdog timer so that it resets the chip when it expires. At startup, the code evaluates whether the recent reset event has been caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 12-1. Watchdog Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
#define WDOGPERIOD 0x00200000

.section L1_code;
.global _reset;
_reset:
  ...
/* optionally, test whether reset was caused by watchdog */
```

```
    p0.h=hi(SWRST);
    p0.l=lo(SWRST);
    r6 = w[p0] (z);
    CC = bittst(r6, bitpos(RESET_WDOG));
    if !CC jump _reset.no_watchdog_reset;

/* optionally, warn at system level or host device here */

_reset.no_watchdog_reset:
/* optionally, set NOBOOT bit to avoid reboot in case */
    p0.h=hi(SYSCR);
    p0.l=lo(SYSCR);
    r0 = w[p0](z);
    bitset(r0,bitpos(NOBOOT));
    w[p0] = r0;


/* start watchdog timer, reset if expires */
    p0.h = hi(WDOG_CNT);
    p0.l = lo(WDOG_CNT);
    r0.h = hi(WDOGPERIOD);
    r0.l = lo(WDOGPERIOD);
    [p0] = r0;
    p0.l = lo(WDOG_CTL);
    r0.l = WDEN | WDEV_RESET;
    w[p0] = r0;
    ...
    jump _main;
_reset.end:
```

The subroutine shown in Listing 12-2 can be called by software to service the watchdog. Note that the value written to the `WDOG_STAT` register does not matter.

Listing 12-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Listing 12-3 is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 12-3. Watchdog Restarted by Interrupt Service Routine

```
isr_watchdog:
    [--sp] = astat;
    [--sp] = (p5:5, r7:7);
    p5.h = hi(WDOG_CTL);
    p5.l = lo(WDOG_CTL);
    r7.l = WDDIS;
    w[p5] = r7;
    bitset(r7, bitpos(WDRO));
    w[p5] = r7;
    r7 = [p5 + WDOG_CNT - WDOG_CTL];
    [p5 + WDOG_CNT - WDOG_CTL] = r7;
    r7.l = WDEN | WDEV_GPI;
    w[p5] = r7;
    (p5:5, r7:7) = [sp++];
    astat = [sp++];
    rti;
isr_watchdog.end:
```

Unique Information for the ADSP-BF50x Processor

None.

13 GENERAL-PURPOSE COUNTER

This chapter describes the general-purpose up/down counter. The counter provides support for manually controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial encoders. Following the overview and list of key features is a description of the operating modes.

This chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of GP counters for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For GP counter interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the GP counters is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each GP counter, refer to Chapter A, "System MMR Assignments".

GP counter behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the

section “Unique Information for the ADSP-BF50x Processor” on page 13-37.

Overview

The purpose of this interface is to convert pulses from incremental position encoders into data that is representative of the actual position. This is done by integrating (counting) pulses on one or two inputs. Since integration provides relative position, some devices also feature a zero position input (zero marker) that can be used to establish a reference point to verify that the acquired position does not drift over time.

In addition, the incremental position information can be used to determine speed, if the time intervals are measured.

The GP counter provides flexible ways to establish position information. When used in conjunction with the GP timer block, the GP counter allows for the acquisition of coherent position/time-stamp information that enables speed calculation.

Features

The GP counter includes the following features:

- 32-bit up/down counter
- Quadrature encoder mode (Gray code)
- Binary encoder mode
- Alternative frequency-direction mode
- Timed direction and up/down counting modes
- Zero marker/push button support

- Capture event timing in association with general purpose timer
- Boundary comparison and boundary setting features
- Input pin noise filtering (debouncing)
- Flexible error detection/signaling

Interface Overview

A block diagram of the GP counter is shown in Figure 13-1. There are two input pins, the count up and direction (CUD) pin and the count down and gate (CDG) pin, that accept various forms of incremental inputs and are processed by the 32-bit counter. The third input, count zero marker (CZM), is the zero marker input. The module interfaces to the processor by way of the peripheral access bus (PAB) and can optionally generate an interrupt request through the IRQ line. There is also an output that can be used by the timer module to generate time-stamps on certain events.

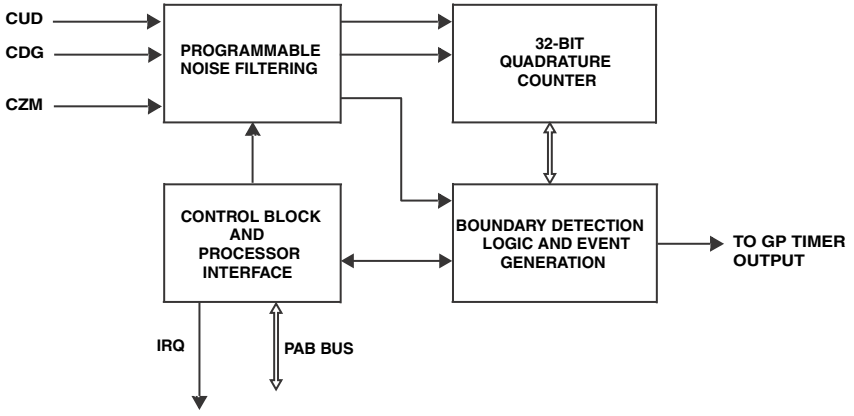


Figure 13-1. Block Diagram of the GP Counter Interface

Description of Operation

The GP counter has five modes of operation that are described in this section.

With the exception of the timed direction mode, the GP counter can operate with the GP timer block to capture additional timing information (time-stamps) associated with events detected by this block.

The third input (CZM) may be used as a zero marker or to sense the pressing of a push button. Refer to "Zero Marker (Push Button) Operation" on page 13-9 for more details.

The three input pins may be filtered (debounced) before being evaluated by the GP counter. Refer to "Input Noise Filtering (Debouncing)" on page 13-7 for more details.

The GP counter also features a flexible boundary comparison. In all of the operating modes, the counter can be compared to an upper and lower limit. A variety of actions can be taken when these limits are reached. Refer to "Boundary Comparison Modes" on page 13-10 for more details.

Quadrature Encoder Mode

In this mode, the CUD:CDG inputs expect a quadrature-encoded signal that is interpreted as a 2-bit Gray code. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The CNT_COUNTER register contains the number of transitions that have occurred. Refer to Table 13-1 for more details.

Optionally, an interrupt is generated if both inputs change within one SCLK cycle. Such transitions are not allowed by Gray coding. Therefore, the CNT_COUNTER register remains unchanged and an error condition is signaled.

Table 13-1. Quadrature Events and Counting Mechanism

CNT_COUNTER Register Value	–4	–3	–2	–1	0	+1	+2	+3	+4
CDG:CUD Inputs	00	01	11	10	00	01	11	10	00

It is possible to reverse the count direction of the Gray coded signal. This can be achieved by enabling the polarity inverter of either the CUD pin or the CDG pin. Inverting both pins will not alter the behavior. This feature can be enabled with the CDGINV and CUDINV bits in the CNT_CONFIG register.

As an example, if the CDG:CUD inputs are 00 respectively and the next transition is to 01, this would normally increment the counter as is shown in Table 13-1. If the CUD polarity is inverted this generates a received input of 01 followed by 00. This will result in a decrement of the counter, altering the behavior of the connected hardware.

Binary Encoder Mode

This mode is almost identical to the previous mode, with the exception that the CUD:CDG inputs expect a binary-encoded signal. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The CNT_COUNTER register contains the number of transitions that have occurred. Refer to Table 13-2.

Optionally, an interrupt is generated if the detected code steps by more than 1 (in binary arithmetic) within one SCLK cycle. Such transitions are considered erroneous. Therefore, the CNT_COUNTER register remains unchanged and an error condition is signaled.

Table 13-2. Binary Events and Counting Mechanism

CNT_COUNTER Register Value	–4	–3	–2	–1	0	+1	+2	+3	+4
CDG:CUD Inputs	00	01	10	11	00	01	10	11	00

Reversing the CUD and CDG pin polarity has a different effect for the binary encoder mode than for the quadrature encoder mode. Inverting the polarity of the CUD pin only, or inverting both the CUD and CDG pins, will result in reversing the count direction.

Up/Down Counter Mode

In this mode, the counter is incremented or decremented at every active edge of the input pins.

If an active edge is detected at the CUD input, the counter increments. The active edge can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will increment the counter. If this bit is set, a falling edge will increment the counter.

If an active edge is detected at the CDG input, the counter decrements. The active edge can be selected by the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will decrement the counter. If this bit is set, a falling edge will decrement the counter.

If simultaneous edges occur on pin CDG and pin CUD, the counter remains unchanged and both up-count and down-count events are signaled in the CNT_STATUS register.

Direction Counter Mode

In this mode, the counter is incremented or decremented at every active edge of the CDG input pin.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment, a low input will decrement. If this bit is set, the polarity is inverted.

If an active edge is detected at the CDG input, the counter value changes by one in the selected direction. The active edge can be selected by the

CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will decrement the counter. If this bit is set, a falling edge will decrement the counter.

Timed Direction Mode

In this mode, the counter is incremented or decremented at each SCLK cycle.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment the counter, a low input will decrement it. If this bit is set, the polarity is inverted.

The CDG pin can be used to gate the clock. The polarity can be selected by the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a high CDG input will enable the counter, a low input will stop it. If this bit is set, the polarity is inverted.

Functional Description

The following sections describe the various functions in more detail.

Input Noise Filtering (Debouncing)

In all modes, the three input pins can be filtered to present clean signals to the GP counter logic. This filtering can be enabled or disabled by the DEBE bit in the CNT_CONFIG register. Figure 13-2 shows the filtering operation for the CUD pin.

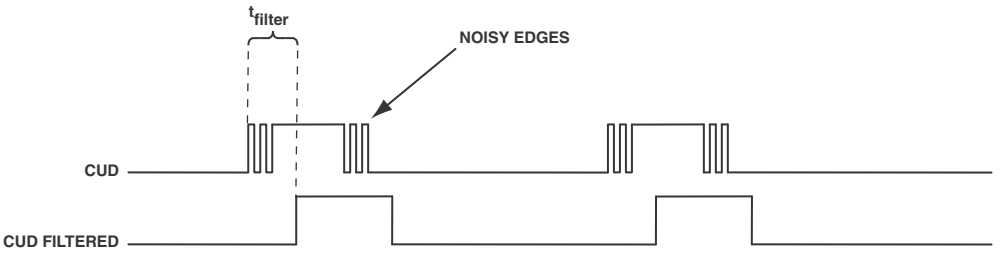


Figure 13-2. Programmable Noise Filtering

The filtering mechanism is implemented using counters for each pin. The counter for each pin is initialized from the DPRESCALE field of the CNT_DEBOUNCE register. When a transition is detected on a pin, the corresponding counter starts counting up to the programmed number of SCLK cycles. The state of the pin is latched after time t_{filter} and passed on to the GP counter logic.

The 5-bit DPRESCALE field in the CNT_DEBOUNCE register programs the desired number of cycles and therefore the debouncing time. The number of SCLK cycles for each pin can be selected in 18 steps ranging from 1 × 128 SCLK periods to 131072 × 128 SCLK periods (see Figure 13-9 on page 13-24).

The time t_{filter} is determined, given SCLK and the DPRESCALE value contained in the CNT_DEBOUNCE register, by the following formula:

$$t_{filter} = 128 \times (2^{DPRESCALE} \div SCLK)$$

where DPRESCALE can contain values from 0 (minimum filtering) to 17 (maximum filtering).

Assuming an SCLK frequency of 133 MHz, the filter time range is shown by the following equations:

DPRESCALE = 0b0000

$$t_{filter} = 128*1*7.5ns = 960ns = (approx.)\ 1\mu s$$

DPRESCALE = 0b10001

$t_{filter} = 128*(131072)*7.5ns = 125829\mu s = (approx.)\ 126ms$

Zero Marker (Push Button) Operation

The CZM input pin can be used to sense the zero marker output of a rotary device or to detect the pressing of a push button. There are four programming schemes which are functional in all counter modes:

- **Push button mode**–This mode is enabled by setting the CZMIE bit in the CNT_IMASK register. An active edge at the CZM input will set the CZMII bit in the CNT_STATUS register. If enabled at the system interrupt controller, this will generate an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register (rising edge if cleared, falling edge if set to one).
- **Zero-marker-zeros-counter mode**–This mode is enabled by setting the ZMZC bit in the CNT_CONFIG register. An active level at the CZM input clears the CNT_COUNTER register and holds it until the CZM pin is deactivated. In addition, if enabled by the CZMZIE bit in the CNT_IMASK register, it will set the CZMZII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this will generate an interrupt request. The active level is selected by the CZMINV bit in the CNT_CONFIG register (active high if cleared, active low if set to one).
- **Zero-marker-error mode**–This mode is used to detect discrepancies between counter value and the zero marker output of certain rotary encoder devices. It is enabled by setting the CZMEIE bit in the CNT_IMASK register. When an active edge is detected at the CZM input pin, the four LSBs of the CNT_COUNTER register are compared to zero. If they are not zero, a mismatch is signaled by way of the CZMEII bit in the CNT_STATUS register. If enabled by the peripheral

interrupt controller, this will generate an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register: (rising edge if cleared, falling edge if set to one).

- **Zero-once mode**–This mode is used to perform an initial reset of the counter value when an active zero marker is detected. After that, the zero marker is ignored (the counter is not reset anymore). This mode is enabled by setting the W1ZMONCE bit in the CNT_COMMAND register. The CNT_COUNTER register and the W1ZMONCE bit are cleared on the next active edge on the CZM pin. Thus, the W1ZMONCE bit can be read to check whether the event has already occurred, if desired. The active edge of the CZM pin is selected by the CZMINV bit in the CNT_CONFIG register (rising edge if cleared, falling edge if set to one).

Boundary Comparison Modes

The GP counter includes two boundary registers, CNT_MIN (lower) and CNT_MAX (upper). The counter value is compared to the lower and upper boundary. Depending on which mode is selected, different actions are taken if the count value reaches either of the boundary values.

There are four boundary modes:

- **Boundary-compare mode**–The two boundary registers are simply compared to the CNT_COUNTER register. If, after incrementing, CNT_COUNTER equals CNT_MAX then the MAXCII bit in the CNT_STATUS register is set. If the MAXCIE bit in the CNT_IMASK register is set, an interrupt request is generated. Similarly if, after decrementing, CNT_COUNTER equals CNT_MIN then the MINCII status bit is set. If the MINCIE bit in the CNT_IMASK register is set, an interrupt request is generated. The MAXCII and MINCII bits are not set if the CNT_MAX and CNT_MIN registers are updated by software.

- **Boundary-zero mode**–This mode is similar to the boundary-compare mode. In addition to setting the status bits and requesting interrupts, the counter value in the CNT_COUNTER register is also set to zero.
- **Boundary auto-extend mode**–In this mode, the boundary registers are modified by hardware whenever the counter value reaches either of them. The CNT_MAX register is loaded with the current CNT_COUNTER value if the latter increments beyond the CNT_MAX value. Similarly, the CNT_MIN register is loaded with the CNT_COUNTER value if the latter decrements below the CNT_MIN value. This mode may be used to keep track of the widest angle the wheel ever reported, even if the software did not serve interrupts. At startup, the application software should set both boundary registers to the initial CNT_COUNTER value. The MAXCII and MINCII status bits are still set when the counter value matches the boundary register.
- **Boundary-capture mode**–In this mode, the CNT_COUNTER value is latched into the CNT_MIN register at one detected edge of the CZM input pin, and latched into CNT_MAX at the opposite edge. If the CZMINV bit in the CNT_CONFIG register is cleared, a rising edge captures into CNT_MIN and a falling edge into CNT_MAX. If the CZMINV bit is set, the edges are inverted. The MAXCII and MINCII status bits report the capture event.

The comparison is performed with signed arithmetic. The boundary registers and the counter value are all treated as signed integer values.

Control and Signaling Events

Eleven events can be signaled to the processor using status information and optional interrupt requests. The interrupts are enabled by the respective bits in the CNT_IMASK register. Dedicated bits in the CNT_STATUS register report events. When an interrupt from the GP counter is

acknowledged, the application software is responsible for correct interpretation of the events. It is recommended to logically AND the content of the CNT_IMASK and CNT_STATUS registers to identify pending interrupts. Interrupt requests are cleared by write-one-to-clear (W1C) operations to the CNT_STATUS register. Hardware does not clear the status bits automatically, unless the counter module is disabled.

Illegal Gray/Binary Code Events

When the illegal transitions described in "Quadrature Encoder Mode" on page 13-4 or "Binary Encoder Mode" on page 13-5 occur, the ICII bit in the CNT_STATUS register is set. If enabled by the ICIE bit in the CNT_IMASK register, an interrupt request is generated. The ICIE bit should only be set in the quadrature encoder or binary encoder modes.

Up/Down Count Events

The UCII bit in the CNT_STATUS register indicates whether the counter has been incremented. Similarly, the DCII bit reports decrements. The two events are independent. For instance, if the counter first increments by one and then decrements by two, both bits remain set, even though the resulting counter value shows a decrement by one. In up/down counter mode, hardware may detect simultaneous active edges on the CUD and CDG inputs. In that case, the CNT_COUNTER remains unchanged, but both the UCII and DCII bits are set.

Interrupt requests for these events may be enabled through the UCIE and DCIE bits. This feature should be used carefully when the counter is clocked at high rates. This is especially critical when the counter operates in DIR_TMR mode, as interrupts would be generated every SCLK cycle.

These events can also be used for additional push buttons, if GP counter features are not needed. When up/down counter mode is enabled, these count events can be used to report interrupts from push buttons that connect to the CUD and CDG inputs.

Zero-Count Events

The CZEROII status bit indicates that the CNT_COUNTER has reached a value equal to 0x0000 0000 after an increment or decrement. This bit is not set when the counter value is set to zero by a write to CNT_COUNTER or by setting the W1LCNT_ZERO bit in the CNT_COMMAND register. If enabled by the CZEROIE bit, an interrupt request is generated.

Overflow Events

There are two status bits that indicate whether the signed counter register has overflowed from a positive to a negative value or vice versa.

The COV31II bit reports that the 32-bit CNT_COUNT register has either incremented from 0x7FFF FFFF to 0x8000 0000, or decremented from 0x8000 0000 to 0x7FFF FFFF. If enabled by the COV31IE bit, an interrupt request is generated.

Similarly, in applications where only the lower 16 bits of the counter are of interest, the COV15II status bit reports counter transitions from 0xXXXX 7FFF to 0xXXXX 8000, or from 0xXXXX 8000 to 0xXXXX 7FFF. If enabled by the COV15IE bit, an interrupt request is generated.

Boundary Match Events

The MINCII and MAXCII status bits report boundary events as described in "Boundary Comparison Modes" on page 13-10. These bits are not set if the CNT_COUNTER, CNT_MAX or CNT_MIN registers are updated by software or the CNT_COMMAND register is written to.

The MINCIE and MAXCIE bits in the CNT_IMASK register enable interrupt generation on boundary events.

Zero Marker Events

There are three status bits CZMII, CZMEII and CZMZII associated with zero marker events, as described in "Zero Marker (Push Button) Operation" on page 13-9. Each of these events can optionally generate an interrupt request, if enabled by the corresponding CZMIE, CZMEIE and CZMZIE bits in the CNT_IMASK register.

Capturing Timing Information

To calculate speed, many applications may wish to measure the time between two count events—in addition to accurately counting encoder pulses. For more accuracy, particularly at very low speeds, it is also necessary to obtain the time that has elapsed since the last count event. This additional information allows for estimating how much the GP counter has advanced since the last counter event.

For this purpose, the GP counter has an internal signal that connects to the alternate capture input (TACIx) of one of the GP timers. It is functional in all modes, with the exception of the timed direction mode. Refer to the "Internal Interfaces" section of Chapter 9, "General-Purpose Ports" for information regarding which GP timer(s) are associated with which GP counter module(s) for your device.

In order to use the timing measurements, the associated GP timer must be used in the WDTH_CAP mode. The alternate capture input is selected by setting the TIN_SEL bit in the GP timer's TIMER_CONFIG register. For more information about the GP timers and their operating modes, refer to the *General-Purpose Timer* chapter.

Capturing Time Interval Between Successive Counter Events

When the only timing information of interest is the interval between successive count events, the associated timer should be programmed in WDTH_CAP mode with PULSE_HI = 1, PERIOD_CNT = 1 and TIN_SEL = 1.

Typically, this information is sufficient if the speed of GP counter events is known not to reach very low values. Figure 13-3 shows the operation of the GP counter and the GP timer in this mode. TO generates a pulse every time a count event occurs. The GP timer will update its TIMER_PERIOD register with the period (measured from rising edge to rising edge) of the TO signal. The TIMER_PERIOD register is updated at every rising edge of the TO signal and contains the number of system clock (SCLK) cycles that have elapsed since the previous rising edge.

Incidentally, the TIMER_WIDTH register is also updated at the same time, but is generally of no interest in this mode of operation. If no reads of the CNT_COUNTER register occur between counter events, the TIMER_WIDTH register only contains the width of the TO pulse. If a read of CNT_COUNTER has occurred between events, the TIMER_WIDTH register will contain the time between the read of CNT_COUNTER and the next event.

This mode can also be used with PULSE_HI = 0. In this case, the period of TO is measured between falling edges. It will result in the same values as in the previous case, only the latching occurs one SCLK cycle later.

Capturing Counter Interval and CNT_COUNTER Read Timing

It is possible to also capture the time elapsed since the last count event. In this mode, the associated timer should be programmed in WDTH_CAP mode with PULSE_HI = 0, PERIOD_CNT = 0 and TIN_SEL = 1. Typically, this additional information is used to estimate the advancement of the GP counter since the last count event, when the speed is very low. Figure 13-4 shows the operation of the GP counter module and the GP timer module in this mode. TO generates a pulse every time a count event occurs. In addition, when the processor reads the CNT_COUNTER register, the TO signal presents a pulse which is extended (high) until the next count event. The GP timer will update its TIMER_PERIOD register with the period (measured from falling edge to falling edge, because PULSE_HI = 0) of the TO signal. The TIMER_WIDTH register is updated with the pulse width (the portion where

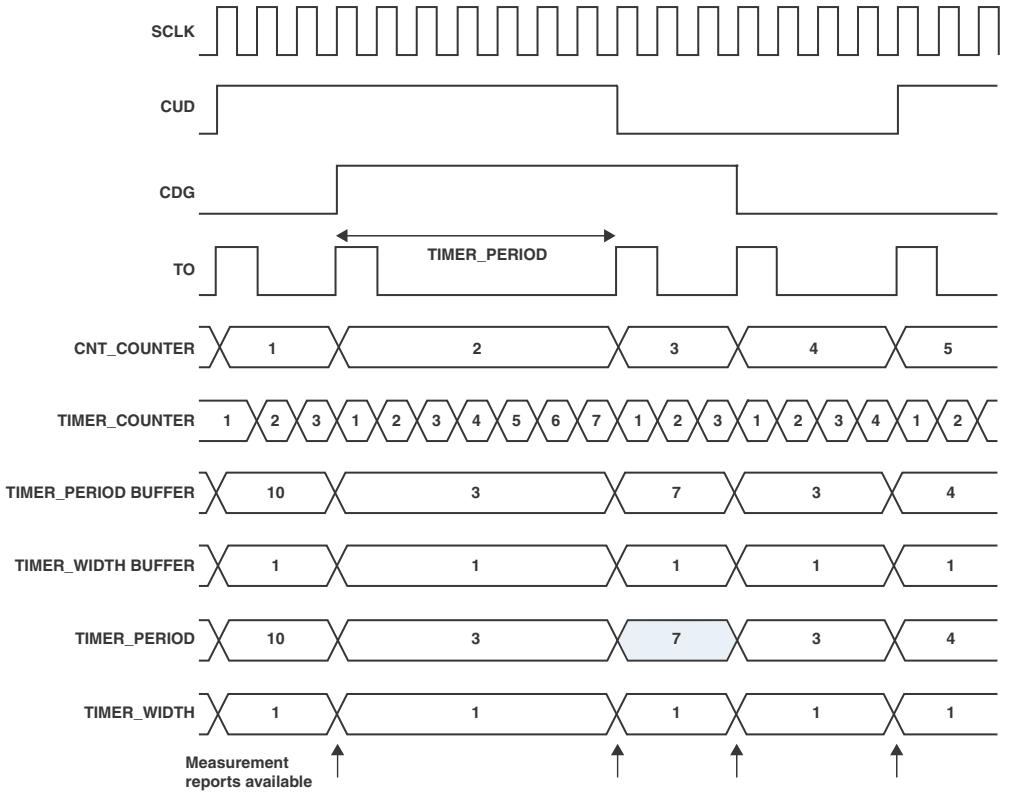


Figure 13-3. Operation With GP Timer Module

TO is low, again because PULSE_HI = 0). Both registers are updated at every rising edge of the TO signal (because PERIOD_CNT = 0). Therefore, the TIMER_PERIOD register contains the period between the last two count events and the TIMER_WIDTH register contains the time since the last count event and the read of the CNT_COUNTER register, both measured in number of SCLK cycles.

The result is that when reading the CNT_COUNTER register, the two time measurements are also latched and the user has a coherent triplet of information to calculate speed and position.

Restrictions apply to the use of the TO signal in terms of speed. Therefore, the user must take care to not operate at very high count events. For instance, if `CNT_COUNTER` is incremented/decremented every `SCLK` cycle (timed direction mode), the TO signal is incorrect.

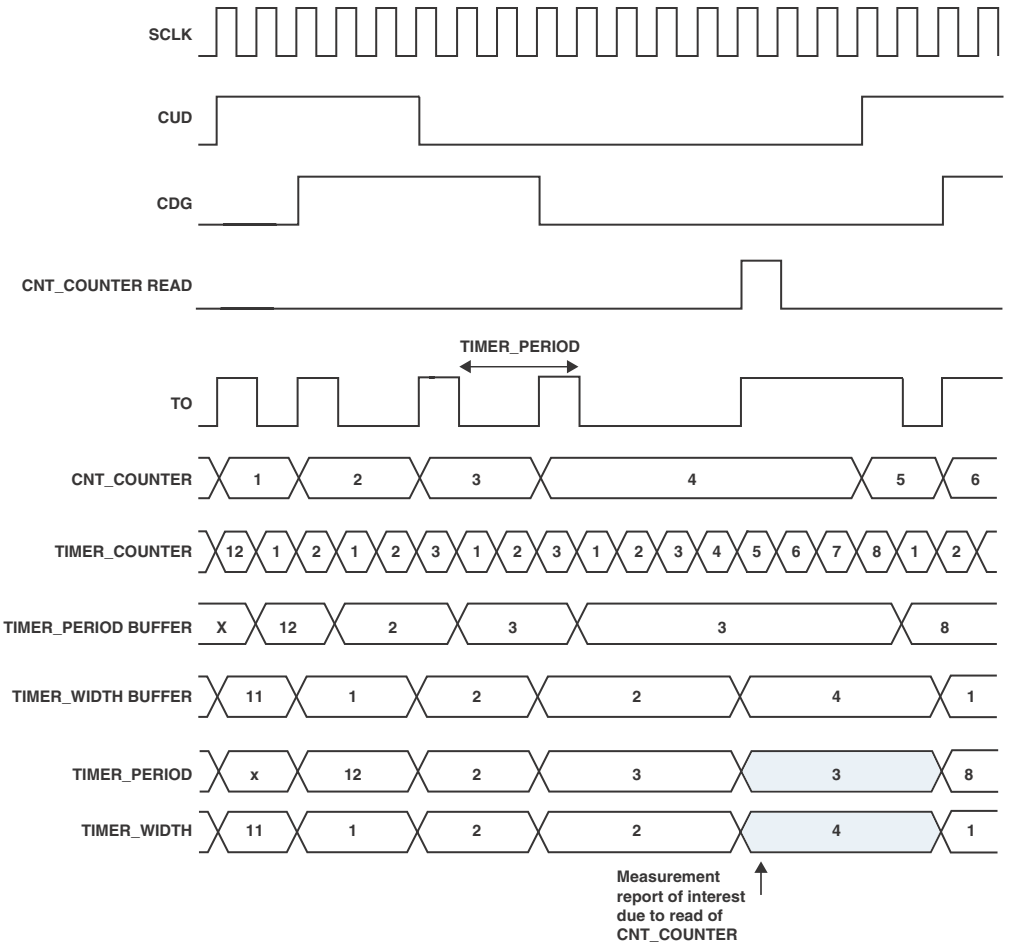


Figure 13-4. Capturing Counter Interval

Programming Model

In a typical application, the user will initialize the GP counter for the desired mode, without enabling it. Normally the events of interest will be processed using interrupts rather than polling the status bit. In that case, clear all status bits and activate the generation of interrupt requests with the `CNT_IMASK` register. Set up the system interrupt controller and core interrupts. If timing information is required, set up the relevant GP timer in `WDTH_CAP` mode with the settings described in the "Capturing Timing Information" on page 13-14. Then, enable the interrupts and the peripheral itself.

Registers

The GP counter interface has eight memory-mapped registers (MMRs) that regulate its operation. Descriptions and bit diagrams for MMRs is provided in the sections that follow.

Counter Module Register Overview

Refer to Table 13-3 for an overview of all MMRs associated with the GP counter interface.

Table 13-3. Counter Module Register Overview

Register Name	Width	PAB Operation	Reset Value
CNT_CONFIG	16 bits	R/W	0x0000
CNT_IMASK	16 bits	R/W	0x0000
CNT_STATUS	16 bits	R/W1C	0x0000
CNT_COMMAND	16 bits	R/W1A	0x0000
CNT_DEBOUNCE	16 bits	R/W	0x0000
CNT_COUNTER	32 bits	R/W (16/32 bits)	0x0000 0000

Table 13-3. Counter Module Register Overview (Cont'd)

Register Name	Width	PAB Operation	Reset Value
CNT_MAX	32 bits	R/W (16/32 bits)	0x0000 0000
CNT_MIN	32 bits	R/W (16/32 bits)	0x0000 0000

Counter Configuration Register (CNT_CONFIG)

This register (Figure 13-5) is used to configure counter modes and input pins, as well as to enable the peripheral. It can be accessed at any time with 16-bit read and write operations.

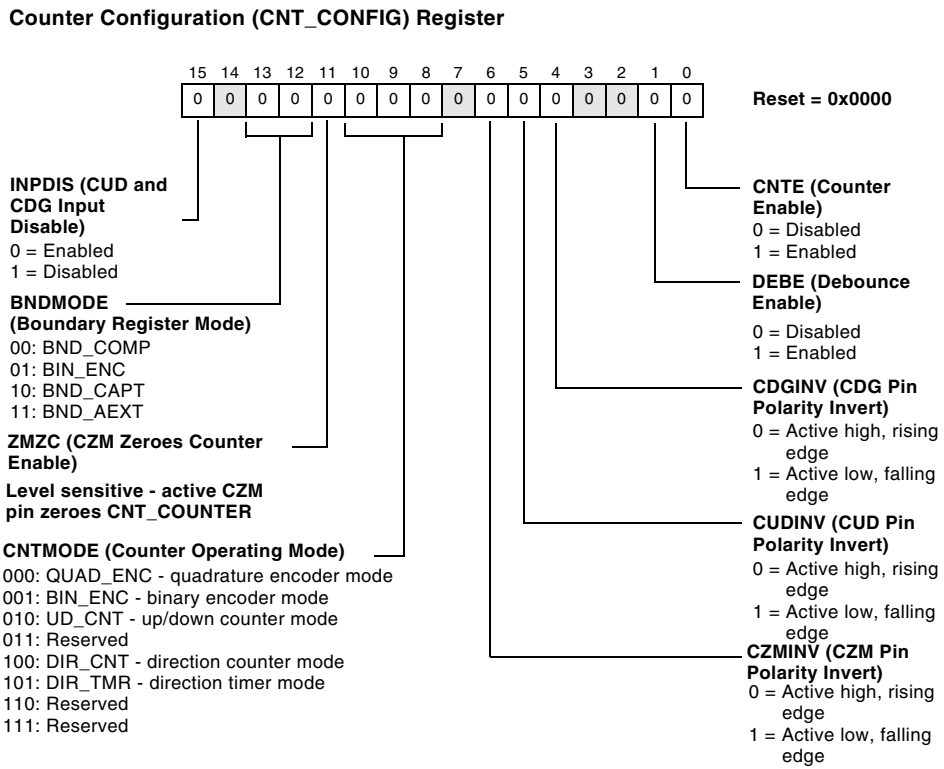


Figure 13-5. Counter Configuration Register

Counter Interrupt Mask Register (CNT_IMASK)

This register (Figure 13-6) is used to enable interrupt request generation from each of the eleven events. It can be accessed at any time with 16-bit read and write operations. For explanations of the register bits, refer to the "Control and Signaling Events" on page 13-11.

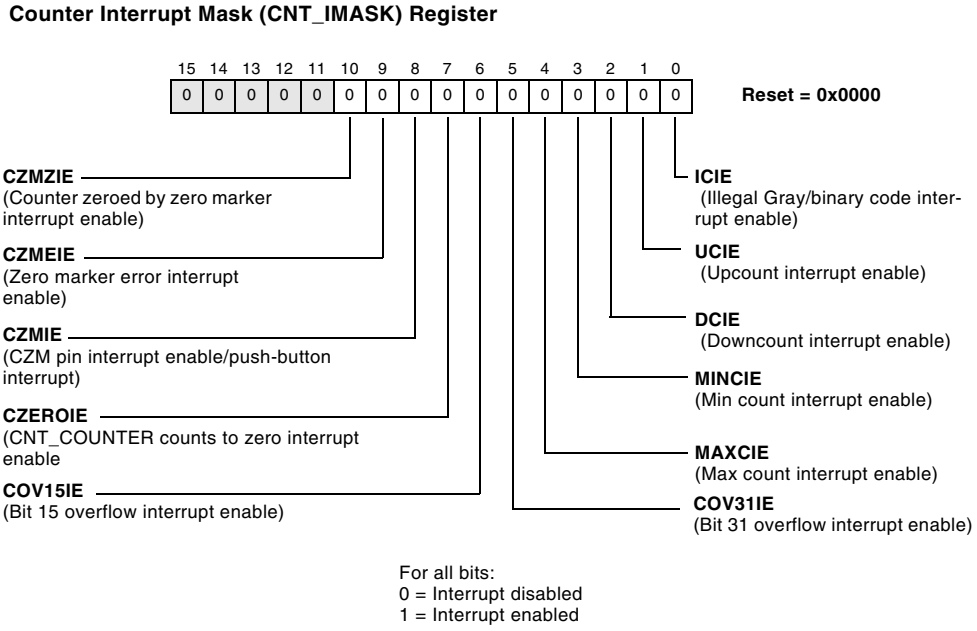


Figure 13-6. Counter Interrupt Mask Register

Counter Status Register (CNT_STATUS)

This register (Figure 13-7) provides status information for each of the eleven events where 0 = no interrupt pending and 1 = interrupt pending. When an event is detected, the corresponding bit in this register is set. It remains set until either software writes a "1" to the bit (write-1-to-clear) or the GP counter is disabled. For explanations of the register bits, refer to the "Control and Signaling Events" on page 13-11.

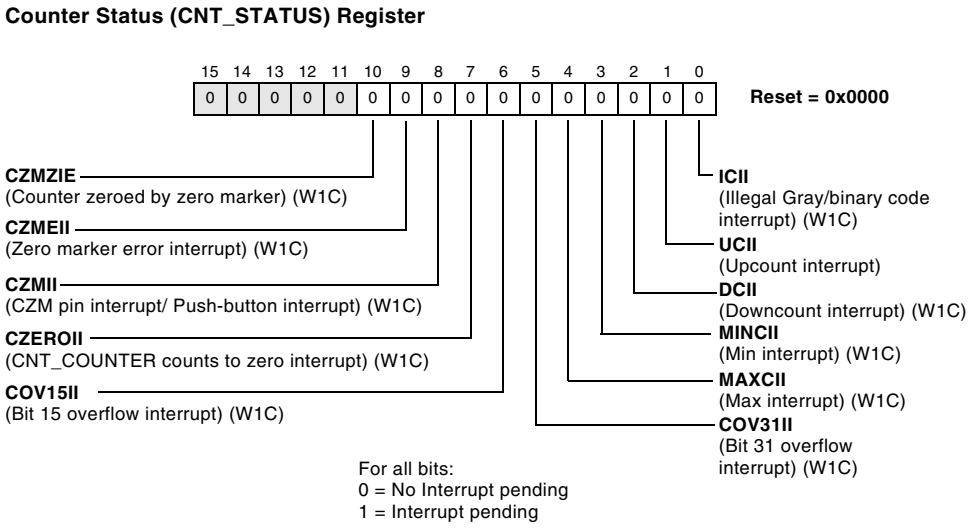


Figure 13-7. Counter Status Register

Counter Command Register (CNT_COMMAND)

The CNT_COMMAND register (shown in Figure 13-8 on page 13-23) configures the GP counter, enabling operations such as zeroing a counter register and copying or swapping boundary registers. These actions are taken by writing a "one" to the appropriate bit.

Read operations from this register will not return meaningful values, with the exception of the W1ZONCE bit, where a "1" indicates that the bit has been set by software before, but no zero marker event has been detected on the CZM pin yet. Refer to the "Zero Marker (Push Button) Operation" on page 13-9 for more details.

The CNT_COUNTER, CNT_MIN and CNT_MAX registers can be initialized to zero by writing a "one" to the W1LCNT_ZERO, W1LMIN_ZERO and W1LMAX_ZERO fields. In addition to clearing registers, CNT_COMMAND allows the boundary registers to be modified in a number of ways. The current counter value in CNT_COUNT can be captured and loaded into either of the two boundary

registers CNT_MAX and CNT_MIN to create new boundary limits. This is performed by setting the W1LMAX_CNT and W1LMIN_CNT bits. Alternatively, the counter can be loaded from CNT_MAX or CNT_MIN via the W1LCNT_MAX and W1LCNT_MIN bits. It is also possible to transfer the current CNT_MAX value into CNT_MIN (or vice versa) through the W1LMIN_MAX and W1LMAX_MIN bits. The final supported operation is the ability to only have the zero marker clear the CNT_COUNT register once, as described in "Zero Marker (Push Button) Operation" on page 13-9.

It is possible for multiple actions to be performed simultaneously by setting multiple bits in the CNT_COMMAND register. However, there are restrictions. The bits associated with each command have been grouped together such that all bits that involve a write to the CNT_COUNTER register are located within bits 3:0 of the CNT_COMMAND register. All commands that involve a write to the CNT_MIN register are located within bits 7:4 of the CNT_COMMAND register, and all commands that involve a write to the CNT_MAX register are located within bits 11:8 of the CNT_COMMAND register.

A maximum of three commands can be issued at any one time, excluding the W1ZMONCE command. Note that (W1LCNT_MIN, W1LCNT_MAX and W1LCNT_ZERO) have to be used exclusively. Never set more than one of them at the same time. The same rule applies for (W1LMAX_MIN, W1LMAX_CNT and W1LMAX_ZERO) and for (W1LMIN_MAX, W1LMIN_CNT, and W1LMIN_ZERO).

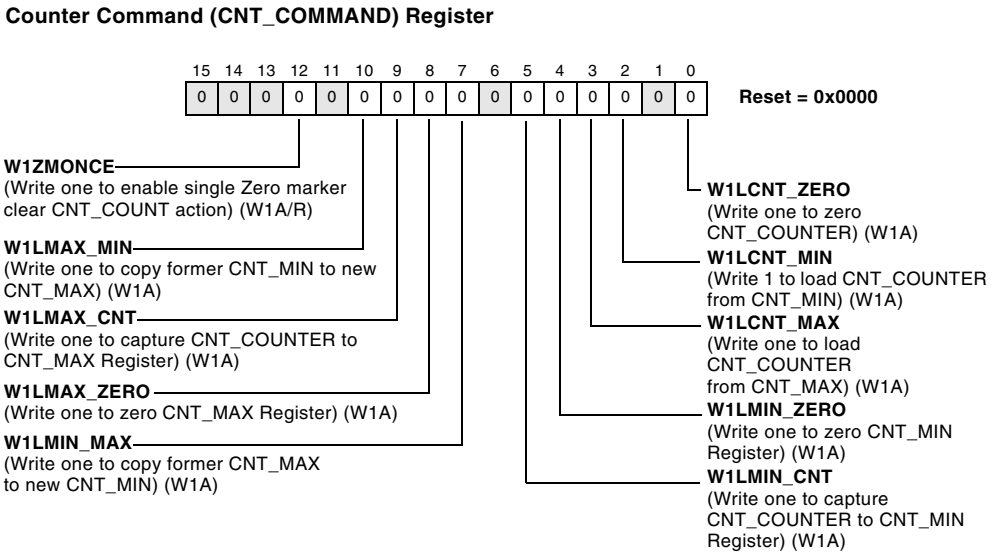


Figure 13-8. Counter Command Register

Counter Debounce Register (CNT_DEBOUNCE)

This register (Figure 13-9) is used to select the noise filtering characteristic of the three input pins (see "Input Noise Filtering (Debouncing)" on page 13-7). Bits [4:0] determine the filter time. The register can be accessed at any time with 16-bit read and write operations.

$$t_{filter} = 128 \times (2^{DPRESCALE} \div SCLK)$$

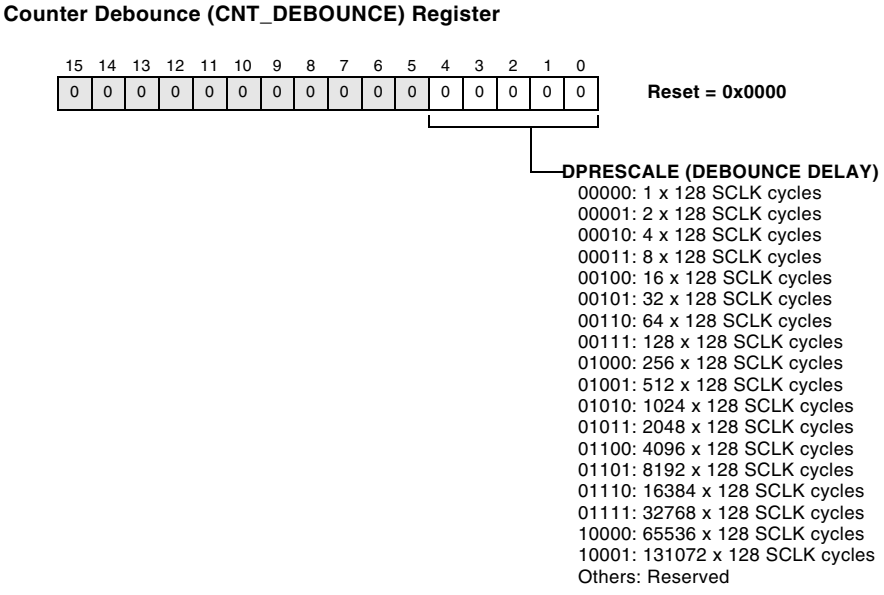


Figure 13-9. Counter Debounce Register

Counter Count Value Register (CNT_COUNTER)

This register (Figure 13-10) holds the 32-bit, twos-complement, count value. It can be read and written at any time. Hardware ensures that reads and write are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows use of the GP counter as a 16-bit counter if sufficient for the application.

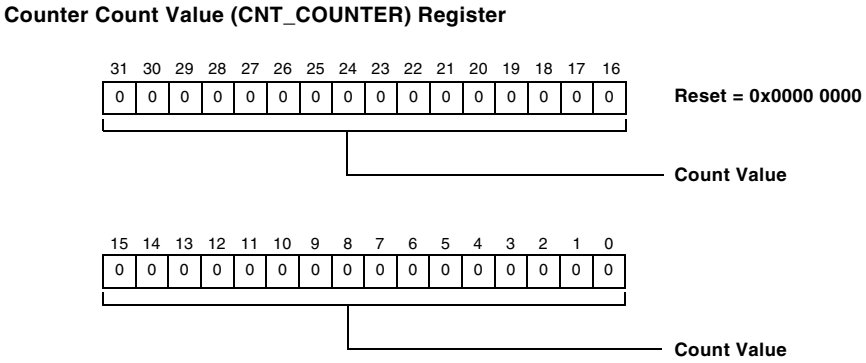


Figure 13-10. Counter Count Value Register

Counter Boundary Registers (CNT_MIN and CNT_MAX)

These registers (Figure 13-11 and Figure 13-12) hold the 32-bit, twos-complement, lower and upper boundary values. They can be read and written at any time. Hardware ensures that reads and write are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows for using the GP counter as a 16-bit counter if sufficient for the application.

Counter Maximal Count (CNT_MAX) Register

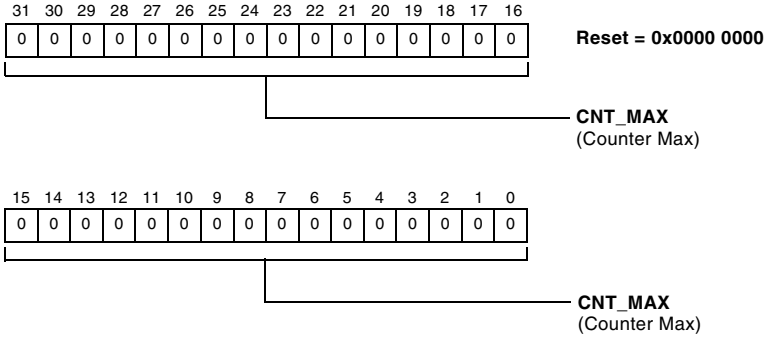


Figure 13-11. Counter Maximal Count Register

Counter Minimal Count (CNT_MIN) Register

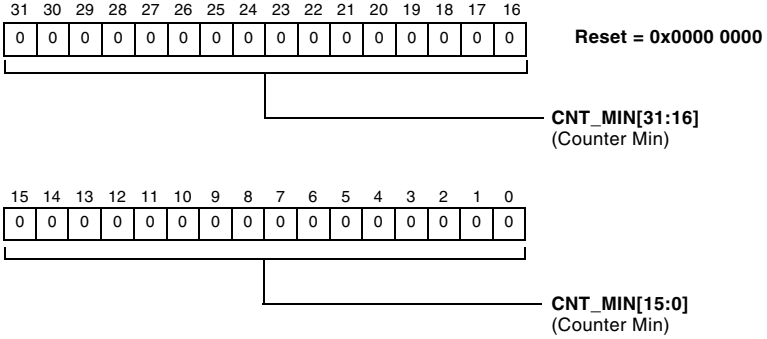


Figure 13-12. Counter Minimal Count Register

Programming Examples

Listing 13-1 illustrates how to initialize the GP counter for various modes. The required interrupts are first unmasked. The GP counter is then configured for the required mode of operation. Note that at this point we do not yet enable the counter. Finally, some GP counter MMRs are cleared, as well as any interrupts that may be pending in the CNT_STATUS register.

Listing 13-1. Initializing the GP Counter

```
/* Setup Counter Interrupts */
P5.H = hi(CNT_IMASK);
P5.L = lo(CNT_IMASK);
R5 = nCZMZIE   /* Counter zeroed by zero marker interrupt */
    | CZMEIE   /* Zero marker error interrupt */
    | CZMIE    /* CZM pin interrupt (push-button) */
    | CZEROIE /* Counts to zero interrupt */
    | nCOV15IE /* Counter bit 15 overflow interrupt */
    | nCOV31IE /* Counter bit 31 overflow interrupt */
    | MAXCIE   /* Max count interrupt */
    | MINCIE   /* Min count interrupt */
    | DCIE     /* Downcount interrupt */
    | UCIE     /* Upcount interrupt */
    | ICIE (z);    /* Illegal gray/binary code interrupt */
w[P5] = R5;

/* Configure the GP Counter mode of operation */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = nINPDIS           /* Enable CUD and CDG inputs */
    | BNDMODE_COMP     /* Boundary compare mode */
    | nZMZC            /* Disable Zero Counter Enable */
    | CNTMODE_QUADENC /* Quadrature Encoder Mode */
    | CZMINV           /* Polarity of CZM pin */
```

```
    | nCUDINV          /* Polarity of CUD pin */
    | nCDGINV          /* Polarity of CDG Pin */
    | nDEBE            /* Disable the debounce */
    | nCNTE (z);       /* Disable the counter */
w[P5] = R5;

/* Zero the CNT_COUNT, CNT_MIN and CNT_MAX registers
This is optional as after reset they are default to zero */
P5.H = hi(CNT_COMMAND);
P5.L = lo(CNT_COMMAND);
R5 = W1LCNT_ZERO | W1LMIN_ZERO | W1LMAX_ZERO (z);
w[P5] = R5;

/* Clear any identified interrupts */
P5.H = hi(CNT_STATUS);
P5.L = lo(CNT_STATUS);
R5.L = ICII     /* Illegal Gray/Binary Code Interrupt Identifier
*/
       | UCII   /* Up count Interrupt Identifier */
       | DCII   /* Down count Interrupt Identifier */
       | MINCII /* Min Count Interrupt Identifier */
       | MAXCII /* Max Count Interrupt Identifier */
       | COV31II /* Bit 31 Overflow Interrupt Identifier */
       | COV15II /* Bit 15 Overflow Interrupt Identifier */
       | CZEROII /* Count to Zero Interrupt Identifier */
       | CZMII   /* CZM Pin Interrupt Identifier */
       | CZMEII /* CZM Error Interrupt Identifier */
       | CZMZII; /* CZM Zeroes Counter Interrupt Identifier */
w[P5] = R5;
```

Listing 13-2 illustrates how to set up the peripheral and core interrupts for the GP counter. This example assumes the counter interrupts are generated on IRQ27, which is assumed to be mapped to the IVG11 interrupt. Finally, the system and peripheral interrupts are unmasked, and then the

GP counter is enabled. This example can be easily tailored to processors with different SIC register mappings.

Listing 13-2. Setting Up the Interrupts for the GP Counter

```
/* Assign the CNT interrupt to IVG11 */
P5.H = hi(SIC_IAR3);
P5.L = lo(SIC_IAR3);
R6.H = hi(0xFFFF4FFF);
R6.L = lo(0xFFFF4FFF);
R7.H = hi(0x00000000);
R7.L = lo(0x00000000);
R5 = [P5];
R5 = R5 & R6; /* zero the counter interrupt field */
R5 = R5 | R7; /* set Counter interrupt to required priority */
[P5] = R5;

/* Set up the interrupt vector for the counter */
R5.H = hi(_IVG11_handler);
R5.L = lo(_IVG11_handler);
P5.H = hi(EVT11);
P5.L = lo(EVT11);
[P5] = R5;

/* Unmask IVG11 interrupt in the IMASK register */
P5.H = hi(IMASK);
P5.L = lo(IMASK);
R5 = [P5];
bitset(R5, bitpos(EVT_IVG11));
[P5] = R5;

/* Unmask interrupt 27 generated by the counter */
P5.H = hi(SIC_IMASK0);
P5.L = lo(SIC_IMASK0);
```

```
R5 = [P5];
bitset(R5, bitpos(IRQ_CNT));
[P5] = R5;

/* Enable the counter */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = w[P5](z);
bitset(R5, bitpos(CNTE));
w[P5] = R5.L;
```

Using the same assumptions from the previous example, Listing 13-3 illustrates a sample interrupt handler that is responsible for servicing the GP counter interrupts. On entry to the handler, the SIC_ISR0 register is interrogated to determine if the counter is waiting for an interrupt to be serviced. If so, the handler responsible for processing all counter interrupts is called.

Listing 13-3. Sample Interrupt Handler for GP Counter Interrupts

```
_IVG11_handler:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = ASTAT;
    [--SP] = (R7:0, P5:0);

    /* Was it a counter interrupt? */
    P5.H = hi(SIC_ISR0);
    P5.L = lo(SIC_ISR0);
    R5 = [P5];
    CC = bittst(R5, bitpos(IRQ_CNT));
    IF !CC JUMP _IVG11_handler.completed;
    CALL _IVG11_handler.counter;

    _IVG11_handler.completed:
```

```
    SSYNC;
    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    ASTAT = [SP++];
    RETS = [SP++];
    RTI;    /* Exit the interrupt service routine */
_IVG11_handler.end:

_IVG11_handler.counter:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = (R7:0, P5:0);

    /* Determine what counter interrupts we wish to service */
    R5 = w[P5](z);
    P5.H = hi(CNT_IMASK);
    P5.L = lo(CNT_IMASK);
    R5 = w[P5](z);

    P5.H = hi(CNT_STATUS);
    P5.L = lo(CNT_STATUS);
    R6 = w[P5](z);
    R5 = R5 & R6;

/* Interrupt handlers for all GP counter interrupts */
_IVG11_handler.counter.illegal_code:
    CC = bittst(R5, bitpos(ICII));
    IF !CC JUMP _IVG11_handler.counter.up_count;

    /* Clear the serviced request */
    R6 = ICII (z);
    w[P5] = R6;
```

```
    /* insert illegal code handler here */

_IVG11_handler.counter.illegal_code.end:


_IVG11_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG11_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

_IVG11_handler.counter.up_count.end:

_IVG11_handler.counter.down_count:
    CC = bittst(R5, bitpos(DCII));
    IF !CC JUMP _IVG11_handler.counter.min_count;

    /* Clear the serviced request */
    R6 = DCII (z);
    w[P5] = R6;

    /* insert down count handler here */

_IVG11_handler.counter.down_count.end:

_IVG11_handler.counter.min_count:
    CC = bittst(R5, bitpos(MINCII));
    IF !CC JUMP _IVG11_handler.counter.max_count;

    /* Clear the serviced request */
```

```
    R6 = MINCII (z);
    w[P5] = R6;

    /* insert min count handler here */

_IVG11_handler.counter.min_count.end:


_IVG11_handler.counter.max_count:
    CC = bittst(R5, bitpos(MAXCII));
    IF !CC JUMP _IVG11_handler.counter.b31_overflow;

    /* Clear the serviced request */
    R6 = MAXCII (z);
    w[P5] = R6;

    /* insert max count handler here */

_IVG11_handler.counter.max_count.end:

_IVG11_handler.counter.b31_overflow:
    CC = bittst(R5, bitpos(COV31II));
    IF !CC JUMP _IVG11_handler.counter.b15_overflow;

    /* Clear the serviced request */
    R6 = COV31II (z);
    w[P5] = R6;

    /* insert bit 31 overflow handler here */

_IVG11_handler.counter.b31_overflow.end:

_IVG11_handler.counter.b15_overflow:
    CC = bittst(R5, bitpos(COV15II));
```

```
    IF !CC JUMP _IVG11_handler.counter.count_to_zero;

    /* Clear the serviced request */
    R6 = COV15II (z);
    w[P5] = R6;

    /* insert bit 15 overflow handler here */

_IVG11_handler.counter.b15_overflow.end:

_IVG11_handler.counter.count_to_zero:
    CC = bittst(R5, bitpos(CZEROII));
    IF !CC JUMP _IVG11_handler.counter.czm;

    /* Clear the serviced request */
    R6 = CZEROII (z);
    w[P5] = R6;

    /* insert count to zero handler here */

_IVG11_handler.counter.count_to_zero.end:

_IVG11_handler.counter.czm:
    CC = bittst(R5, bitpos(CZMII));
    IF !CC JUMP _IVG11_handler.counter.czm_error;

    /* Clear the serviced request */
    R6 = CZMII (z);
    w[P5] = R6;

    /* insert czm handler here */

_IVG11_handler.counter.czm.end:
```

```
_IVG11_handler.counter.czm_error:
    CC = bittst(R5, bitpos(CZMEII));
    IF !CC JUMP _IVG11_handler.counter.czm_zeroes_counter;

    /* Clear the serviced request */
    R6 = CZMEII (z);
    w[P5] = R6;

    /* insert czm error handler here */

_IVG11_handler.counter.czm_error.end:


_IVG11_handler.counter.czm_zeroes_counter:
    CC = bittst(R5, bitpos(CZMZII));
    IF !CC JUMP _IVG11_handler.counter.all_serviced;

    /* Clear the serviced request */
    R6 = CZMZII (z);
    w[P5] = R6;

    /* insert czm zeroes counter handler here */

_IVG11_handler.counter.czm_zeroes_counter.end:

_IVG11_handler.counter.all_serviced:

    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    RETS = [SP++];
    RTS;
_IVG11_handler.counter.end:
```

Listing 13-4 shows how to set up timer 7 (as an example) to capture the period of counter events. Refer to the “Internal Interfaces” section of

Chapter 9, "General-Purpose Ports" for information regarding which GP timer(s) are associated with which GP counter module(s) for your device. The timer is configured for WDTH_CAP mode, and the period between the last two successive counter events is read from within the up count interrupt handler that was provided in Listing 13-3 on page 13-30.

Listing 13-4. Setting Up Timer 7 for Counter Event Period Capture

```
/* configure the timer for WDTH_CAP mode */
P5.H = hi(TIMER7_CONFIG);
P5.l = lo(TIMER7_CONFIG);
R5 = PULSE_HI | PERIOD_CNT | TIN_SEL | WDTH_CAP (z);
w[P5] = R5.l;

/* Enable Timer 7
P5.H = hi(TIMER_ENABLE0);
P5.L = lo(TIMER_ENABLE0);
R5 = TIMEN7 (z);
w[P5] = R5.L;

...

_IVG11_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG11_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

    /* Read the period between the last two successive events */
    P5.H = hi(TIMER7_PERIOD);
```

```
    P5.L = lo(TIMER7_PERIOD);
    R5 = [P5];

    P5.H = hi(_event_period);
    P5.L = lo(_event_period);
    [P5] = R5;
_IVG11_handler.counter.up_count.end:
```

Unique Information for the ADSP-BF50x Processor

None.

14 PWM CONTROLLER

This chapter describes the PWM controller module. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model discussion and consolidated register definitions.

Specific Information for the ADSP-BF50x

For details regarding the number of PWMs for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For PWM Controller interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how the PWM Controller is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for the PWM Controller, refer to Chapter A, "System MMR Assignments".

Overview

The PWM controller is a flexible, programmable, three-phase PWM waveform generator that can be programmed to generate the required switching patterns to drive a three-phase voltage source inverter for ac

induction motor (ACIM) or permanent magnet synchronous motor (PMSM) control.

In addition, the PWM block contains functions that considerably simplify the generation of the required PWM switching patterns for control of electronically commutated motors (ECMs) or brushless dc motors (BDCMs).

Programming the PWM_SRMODE bit of the PWM_CTRL register to 0 enables a special mode used for switched reluctance motors (SRMs). Figure 14-1 shows a block diagram that represents the main functional blocks of the PWM Controller.

The following six blocks control the generation of the six output PWM signals (PWM_AH, PWM_AL, PWM_BH, PWM_BL, PWM_CH, and PWM_CL):

- **Three-Phase PWM Timing Unit**. As the core of the PWM Controller, this block generates three pairs of complemented, center-based PWM signals and PWM_SYNC coordination.
- **Dead Time Control Unit**. This block inserts emergency dead time after the "ideal" PWM output pair, including crossover, is generated.
- **Output Control Unit**. This block permits the redirection of the outputs of the Three-Phase Timing Unit for each channel to the high-side or the low-side output. In addition, the Output Control Unit allows individual enabling/disabling of each of the six PWM output signals.
- **Gate Drive Unit**. This block provides the correct polarity output PWM signals, based on the state of the PWM_POLARITY bit of the PWM_CTRL register. The Gate Drive Unit also permits the generation of the high-frequency chopping waveform and its subsequent mixing with the PWM output signals.

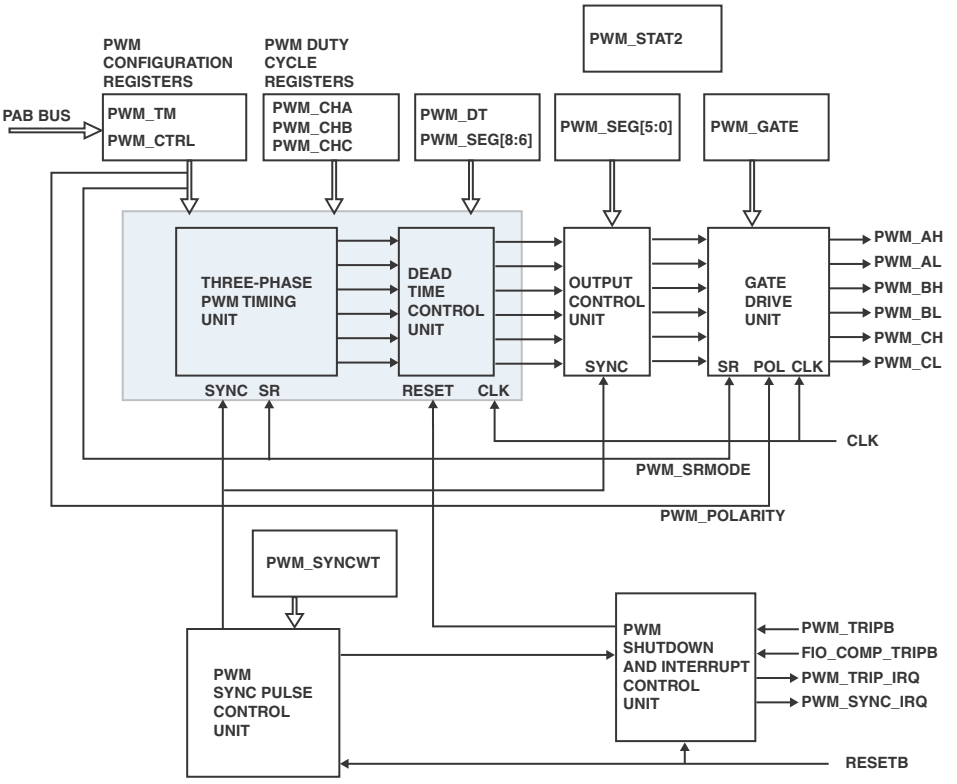


Figure 14-1. PWM Controller Block Diagram

- **PWM Shutdown & Interrupt Control Unit**. This block takes care of the various PWM shutdown modes (via the $\overline{\texttt{PWM\_TRIP}}$ pin and the `PWM_CTRL` register). This unit generates the correct reset signal for the Three-Phase PWM Timing Unit and interrupt signals for the Interrupt Control Unit
- **PWM Sync Pulse Control Unit**. This block generates the internal PWM synchronization pulse and also controls whether an external `PWM_SYNC` pulse is used.

The PWM Controller is driven by a clock, whose period is t_{SCLK}. The PWM generator produces three pairs (PWM_AH, PWM_AL, PWM_BH, PWM_BL, PWM_CH, and PWM_CL) of PWM signals on the six PWM output pins. There are three high-side drive signals (PWM_AH, PWM_BH, and PWM_CH) and three low-side drive signals (PWM_AL, PWM_BL, and PWM_CL). The polarity of the generated PWM signals may be programmed by the PWM_POLARITY bit of the PWM_CTRL register to generate active high or active low PWM patterns. The switching frequency and dead time of the generated PWM patterns are programmable via the PWM_TM and PWM_DT registers. In addition, three duty-cycle control registers (PWM_CHA, PWM_CHB, and PWM_CHC) directly control the duty cycles of the three pairs of PWM signals.

Each of the six PWM output signals can be enabled or disabled via separate output enable bits of the PWM_SEG register. In addition, three control bits of the PWM_SEG register permit independent crossover of the two signals of a PWM pair for easy control of ECMs or BDCMs. In crossover mode, the PWM signal destined for the high-side switch is diverted to the complementary low-side output, and the signal destined for the low-side switch is diverted to the corresponding high-side output signal for ECM or BDCM modes of operation. A typical configuration for these types of motors is shown in Figure 14-2.

In common three-phase inverters, it is necessary to insert a so-called “dead time” between turning off one switch and turning on the other switch in the same leg, to prevent shoot-through. This dead time is inserted by an emergency dead-time insertion circuit, which enforces a dead time defined by the PWM_DT register between the high- and low-side drive signals of each PWM channel. This ensures that the correct dead time occurs at the power inverter.

In many applications, there is a need to provide an isolation barrier in the gate-drive circuits that turn on the power devices of the inverter. In general, there are two common isolation techniques: optical isolation using opto-isolators, and transformer isolation using pulse transformers. The PWM Controller permits the mixing of the output PWM signals with a

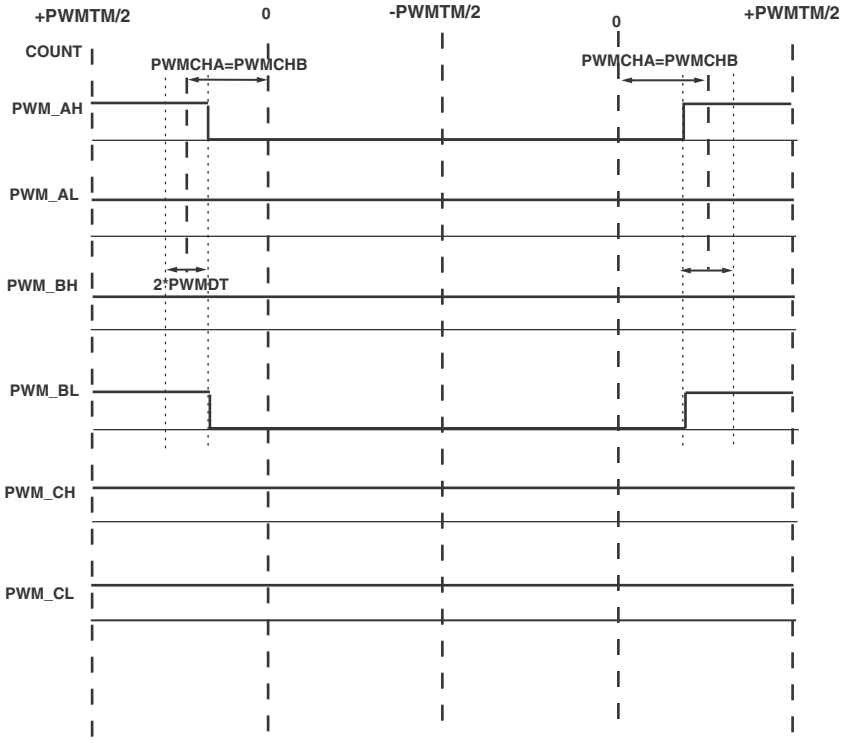


Figure 14-2. Active Low PWM Signals for ECM Control

high-frequency chopping signal, which provides an simple interface to pulse transformers. The features of gate-drive-chopping mode are controlled by the PWM_GATE register. An 8-bit value (GDCLK) within the PWM_GATE register directly controls the chopping frequency. In addition, high-frequency chopping can be independently enabled for the high- and low-side outputs using separate control bits in the PWM_GATE register. In addition, all PWM outputs require sufficient sink and source capability to directly drive most opto-isolators.

The PWM generator is capable of operating in two distinct modes:

- **Single-Update Mode**. In single-update mode, duty cycle values are programmable only once per PWM period; resultant PWM patterns are symmetrical about the mid-point of the PWM period.
- **Double-Update Mode**. In double-update mode, a second updating of the PWM registers is implemented at the midpoint of the PWM period. In double-update mode, it is possible to produce asymmetrical PWM patterns that produce lower harmonic distortion in three-phase PWM inverters. This technique also permits closed-loop controllers to change the average voltage applied to the machine windings at a faster rate, thus permitting faster closed-loop bandwidths to be achieved.

The operating mode of the PWM block (single- or double-update mode) is selected by the PWM_DBL bit in the PWM_CTRL register. Setting PWM_DBL to 1 selects double-update mode, and 0 selects single-update mode.

The PWM generator can provide an internal synchronization pulse on the PWM_SYNC pin that is synchronized to the PWM switching frequency. In single-update mode, a PWM_SYNC pulse is produced at the start of each PWM period. In double-update mode, an additional PWM_SYNC pulse is also produced at the midpoint of each PWM period. The width of the PWM_SYNC pulse is programmable through the PWM_SYNCWT register.

The PWM generator can also accept an external synchronization pulse on the PWM_SYNC pin. External synchronization is selected by setting the PWM_EXTSYNC bit in the PWM_CTRL register. The PWM_SYNC input timing can be synchronized to the internal system clock, which is selected by setting the PWM_SYNCSEL bit of the PWM_CTRL register. If the external synchronization pulse from the chip pin is asynchronous to the internal system clock (typical case), the external PWM_SYNC is considered asynchronous and should be synchronized. If the PWM_SYNC is actually received from another PWM on the same chip controlled by the same system clock, the PWM_SYNC can usually be considered synchronous. Synchronization logic will add

latency and jitter from the external sync pulse to the actual PWM outputs. If the same asynchronous external sync pulse is received by two independent PWM Controllers, synchronization of PWM_SYNC is also done independently and the jitter between the PWM Controllers will not be in unison. The size of the sync pulse on PWM_SYNC must be greater than two system clock periods.

The produced PWM output signals can be shut off via:

- **Hardware**. A dedicated asynchronous PWM shutdown pin ($\overline{\text{PWM\_TRIP}}$) that when brought low (provided it is not disabled by the PWMTRIP_DSBL bit of the PWM_CTRL register) instantaneously places all six PWM outputs in the "off" state (as determined by the state of the PWM_POLARITY bit of the PWM_CTRL register). This hardware shutdown mechanism is asynchronous so that the associated PWM disable circuitry does not go through any clocked logic, thereby ensuring correct PWM shutdown even in the event of a loss of the processor system clock. A trip shutdown in hardware resets the PWM_EN bit in the PWM_CTRL register, but all the other programmable registers maintain their current state.
- **Software**. The PWM system may be shut down in software by disabling the PWM_ENABLE bit in the PWM_CTRL register.

On many processors, the PWM pins are multiplexed with other functionality. Because they can be in a high-impedance state before the PORTx_MUX registers are programmed to select the PWM functionality, there should be external pull-down logic for the $\overline{\text{PWM\_TRIP}}$ pin in these cases. For these and other questions about pin multiplexing, see the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

The PWM unit is capable of generating two different interrupt types. One interrupt (PWM_SYNCINT) is generated on the occurrence of a rising edge on the PWM_SYNC pulse, which is internally generated. The other interrupt (PWM_TRIPINT) is generated on the occurrence of $\overline{\text{PWM\_TRIP}}$, the PWM

shut-down action. Both interrupts are generated only when the corresponding enable bits (PWMSYNCINT_EN and PWMTRIPINT_EN) are set in the PWM_CTRL register.

The PWM_STAT register provides status information about the PWM system. In particular, the state of the $\overline{\text{PWM\_TRIP}}$ pin (PWM_TRIP bit), PWM_POLARITY (PWM_POL bit), and PWM_SRMODE (PWM_SR bit) are available, as well as a status bit (PWM_PHASE) that indicates whether operation is in the first half or the second half of the PWM period. The PWM_STAT register also reflects the status of the PWM_SYNCINT and PWM_TRIPINT interrupts, which are set if enabled in the PWM_CTRL register. The latter two bits are sticky; hence, the interrupt service routine must write-1-to-clear (W1C) these bits.

General Operation

Typically, the PWM_SYNCINT interrupt is used to periodically execute an interrupt service routine (ISR) to update the three PWM channel duties, according to a control algorithm based on expected motor operation and sampled data of the existing motor operation. PWM_SYNC can also trigger the ADC to sample data for use during the ISR. During processor boot, the PWM Controller is initialized and program flow enters a wait loop. When a PWM_SYNCINT interrupt occurs, the ADC samples data, the data is algorithmically interpreted, and then the new PWM channel duties are calculated and written to the PWM registers. More sophisticated implementations include different start-up, run-time, and shut-down algorithms to determine PWM channel duties, based on expected behavior and further features.

During initialization, the PWM_TM register is written to define the PWM period, and the PWM_CHA, PWM_CHB and PWM_CHC registers are written to define the initial channel pulse widths. The PWM_SYNCWT, PWM_GATE, PWM_SEG, PWM_CHAL, PWM_CHBL and PWM_CHCL registers are written, depending on the system configuration and modes. The PWM_STAT register can be

read to determine polarity, and whether switched reluctance (SR) mode (PWM_SR bit) is enabled, and whether an external trip situation is preventing the correct start-up of the PWM Controller. An active external trip event must be resolved prior to PWM startup. The PWM_CTRL register is then written to define the major operating mode and to enable the PWM outputs and PWM sync pulse.

During the PWM_SYNCINT interrupt-driven control loop, only the PWM_CHx duty values are updated typically. The PWM_SEG register may also be updated for other system implementations requiring output crossover.

During an external trip event (if not disabled), the PWM outputs will be turned off (that is, set to the opposite of the "on" polarity configured by the PWM_POLARITY bit of the PWM_CTRL register), and the PWM sync pulse will continue to operate if already enabled. A PWM_TRIPINT interrupt will occur if unmasked, notifying the software of this event. To handle cases where clock signal integrity is an issue, external trips will turn off the PWM outputs, with or without clocks.

Functional Description

This section describes the function of the following PWM features:

Three-Phase PWM Timing Unit and Dead Time Control Unit

The 16-bit Three-Phase PWM Timing Unit is the core of the PWM Controller and produces three pairs of pulse-width modulated signals with high resolution and minimal processor overhead. The outputs of this unit are such that a low level is interpreted as a command to turn on (active-low) the associated power device. Three configuration registers (PWM_CTRL, PWM_TM, and PWM_DT) determine the fundamental characteristics of the PWM outputs. These registers, in conjunction with the three 16-bit duty cycle registers (PWM_CHA, PWM_CHB, and PWM_CHC), control the output of the Three-Phase PWM Timing Unit.

PWM Switching Frequency (PWM_TM) Register

The 16-bit read/write PWM period register (PWM_TM) controls the PWM switching frequency. The fundamental timing unit of the PWM

Controller is t_{SCLK}. Therefore, for a 100 MHz system clock (SCLK), f_{SCLK}, the fundamental time increment (t_{SCLK}) is 10 ns. The value written to the PWM_TM register is effectively the number of t_{SCLK} clock increments in half a PWM period. The required PWM_TM value as a function of the desired PWM switching frequency (f_{PWM}) is given by:

$$PWM\_TM = \frac{f_{SCLK}}{2 \times f_{PWM}}$$

Therefore, the PWM switching period (T_s) can be written as:

$$T_S = 2 \times PWM\_TM \times t_{SCLK}$$

For example, for an f_{SCLK} of 100 MHz and a desired PWM switching frequency (f_{PWM}) of 10 kHz (T_s = 100 µs), the correct value to load into the PWM_TM register is:

$$PWM\_TM = \frac{100 \times 10^6}{2 \times 10 \times 10^3} = 5000$$

The largest value that can be written to the 16-bit PWM_TM register is 0xFFFF = 65,535, which, at an f_{SCLK} of 100 MHz, corresponds to a minimum PWM switching frequency of:

$$f_{PWM(min)} = \frac{100 \times 10^6}{2 \times 65535} = 762Hz$$

PWM_TM values of 0 and 1 are not defined and should not be used when the PWM outputs or PWM sync is enabled.

PWM Switching Dead Time (PWM_DT) Register

The second important parameter that must be set up in the initial configuration of the PWM Controller is the switching dead time. This is a short delay introduced between turning off one PWM signal (for example, AH) and turning on the complementary signal (for example, AL). This short time delay permits the power switch being turned off (AH in this case) to completely recover its blocking capability before the complementary switch is turned on. This time delay prevents a potentially destructive short-circuit condition from developing across the dc link capacitor of a typical voltage source inverter.

The 10-bit, read/write PWM_DT register controls the dead time. This register controls the dead time inserted into the three pairs of PWM output signals. Dead time (T_d) is related to the value in the PWM_DT register by:

$$T_d = PWM\_DT \times 2 \times t_{SCLK}$$

Therefore, a PWM_DT value of 0x00A introduces a 200 ns delay (for a SCLK of 100 MHz) between turning off any PWM signal (for example, AH) and then turning on its complementary signal (for example, AL). The length of the dead time can therefore be programmed in increments of $2t_{SCLK}$ (or 20 ns for an SCLK of 100 MHz). The PWM_DT register is a 10-bit register whose maximum value of 0x3FF (1023 decimal) corresponds to a maximum programmed dead time of:

$$T_{d(max)} = 1023 \times 2 \times t_{SCLK} = 1023 \times 2 \times 10 \times 10^{-9} = 20.5\mu s$$

for an f_{SCLK} rate of 100 MHz. The dead time can be programmed to be zero by writing 0 to the PWM_DT register.

PWM Operating Mode (PWM_CTRL and PWM_STAT) Registers

The PWM Controller can operate in two distinct modes: single-update mode and double-update mode. The mode is determined by the state of PWM_DBL bit of the PWM_CTRL register. When this bit is cleared, the PWM Controller operates in single-update mode. Setting the PWM_DBL bit places the PWM Controller in double-update mode. Following a peripheral reset or power on, the PWM_DBL bit is cleared; thus, PWM Controller operation defaults to single-update mode.

In single-update mode, a PWM_SYNC pulse is produced during each PWM period. The rising edge of this signal marks the start of a new PWM cycle and is used to latch new values from the PWM configuration registers (PWM_TM, PWM_DT, and PWM_SYNCWT), and the PWM duty cycle registers (PWM_CHA, PWM_CHB, PWM_CHC, PWM_CHAL, PWM_CHBL, and PWM_CHCL) into the Three-Phase PWM Timing Unit. In addition, the PWM_SEG register is also latched into the Output Control Unit on the rising edge of the PWM_SYNC pulse. In effect, this means that the characteristics and resultant duty cycles of the PWM signals can be updated only once per PWM period at the start of each cycle. This results in PWM patterns that are symmetrical about the midpoint of the switching period.

In double-update mode, an additional PWM_SYNC pulse is produced at the midpoint of each PWM period. The rising edge of this second PWM_SYNC pulse is again used to latch new values of the PWM configuration registers, duty cycle registers, and the PWM_SEG register. As a result, it is possible to alter both the characteristics (switching frequency, dead time, and PWM_SYNC pulse width) and the output duty cycles at the midpoint of each PWM cycle. Consequently, it is possible to produce PWM switching patterns that are no longer symmetrical about the midpoint of the period (asymmetrical PWM patterns).

In double-update mode, it may be necessary to know whether operation at any point in time is in the first or second half of the PWM cycle. This

information is provided by the PWM_PHASE bit of the PWM_STAT register, which is cleared during operation in the first half of each PWM period (between the rising edge of the original PWM_SYNC pulse and the rising edge of the second PWM_SYNC pulse introduced in double-update mode). The PWM_PHASE bit is set during operation in the second half of each PWM period. This status bit allows determination of the particular half-cycle during implementation of the PWM_SYNC interrupt service routine.

The advantage of double-update mode is that the PWM process can produce lower harmonic voltages, and faster control bandwidths are possible. However, for a given PWM switching frequency, PWM_SYNC pulses occur at twice the rate in double-update mode. Since new duty cycle values are computed in each PWM_SYNCINT interrupt service routine, double-update mode places a larger computational burden on the processor. Alternatively, the same PWM update rate may be maintained at half the switching frequency, yielding lower switching losses.

The PWM_STAT2 status register is provided for software simulation. This register contains the output values of all the three pairs of PWM signals (PWM_AH, PWM_AL, PWM_BH, PWM_BL, PWM_CH, and PWM_CL).

PWM Duty Cycle (PWM_CHA, PWM_CHB, and PWM_CHC) Registers

Three 16-bit read/write duty cycle registers (PWM_CHA, PWM_CHB, and PWM_CHC) control the duty cycles of the six PWM output signals on the PWM_AH, PWM_AL, PWM_BH, PWM_BL, PWM_CH, and PWM_CL pins when not in switched reluctance mode. The two's complement integer value in the PWM_CHA register controls the duty cycle of the signals on the PWM_AH and PWM_AL outputs; in PWM_CHB, it controls the duty cycle of the signals on PWM_BH and PWM_BL; in PWM_CHC, it controls the duty cycle of the signals on PWM_CH and PWM_CL. The duty cycle registers are programmed in two's complement integer counts of the fundamental time unit (t_{SCLK}) and define the desired on-time of the high-side PWM signal produced by the Three-Phase PWM Timing Unit over half the PWM period.

Each duty cycle register range is from (–PWMTM/2 – PWMDT) to (+PWMTM/2 + PWMDT), which, by definition, is scaled such that a value of 0 represents a 50% PWM duty cycle.

The switching signals produced by the Three-Phase PWM Timing Unit are also adjusted to incorporate the programmed dead time value in the PWM_DT register by programming active low polarity in PWM_CTRL. The Three-Phase PWM Timing Unit produces active-low signals to turn on the associated power device.

Figure 14-3 shows a typical pair of PWM outputs (in this case, for PWM_AH and PWM_AL) from the Three-Phase PWM Timing Unit for operation in single-update mode. All illustrated time values indicate the integer value in the associated register and can be converted to time by multiplying by the fundamental time increment (t_{SCLK}) and comparing to the two's complement counter.

Notice that the switching patterns are perfectly symmetrical about the midpoint of the switching period in single-update mode, since the same values of PWM_CHA, PWM_TM, and PWM_DT are used to define the signals in both half cycles of the period.

As implied by Figure 14-3, the programmed duty cycles are adjusted to incorporate the desired dead time into the resultant pair of PWM signals by moving the switching instants of both PWM signals (PWM_AH and PWM_AL) away from the instant set by the PWM_CHA register. Both switching edges are moved by an equal amount (PWMDT * t_{SCLK}) to preserve the symmetrical output patterns. Figure 14-3 shows the PWM_SYNC pulse whose rising edge denotes the beginning of the switching period and whose width is set by the PWM_SYNCWT register. Also shown is the PWM_PHASE bit of the PWM_STAT register, which indicates whether operation is in the first half cycle or second half cycle of the PWM period.

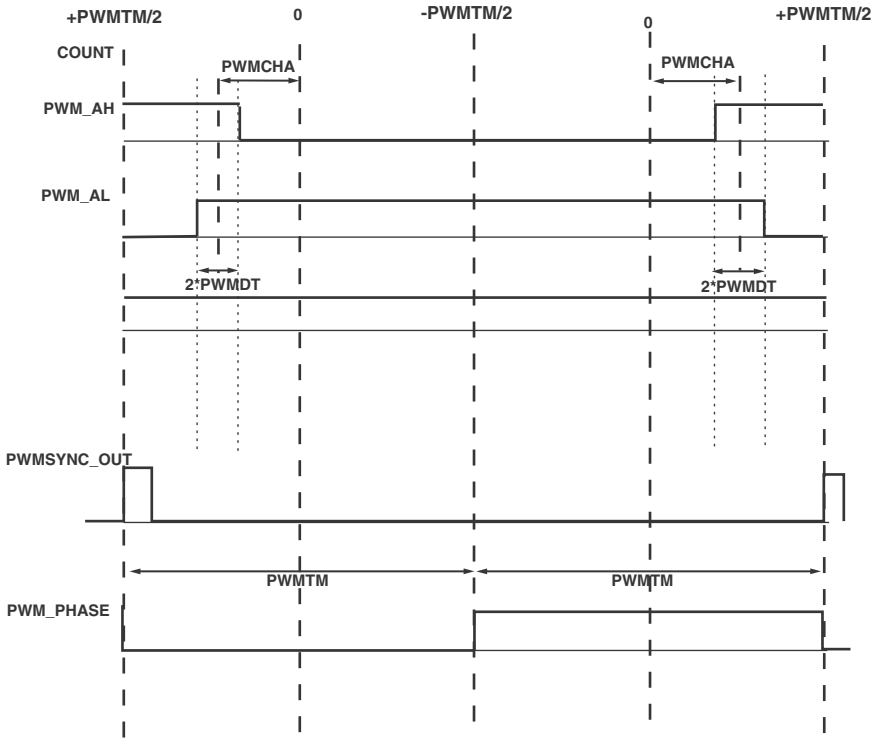


Figure 14-3. Typical PWM Outputs of Three-Phase Timing Unit in Single-Update Mode (Active-Low Waveforms)

The resultant on-times (active low) of the PWM signals over the full PWM period (two half periods) produced by the Three-Phase PWM Timing Unit and illustrated in Figure 14-3, may be written as:

$$T_{AH} = (PWMTM + 2 \times (PWMCHA - PWMDT)) \times t_{SCLK}$$

$$Range\ of\ T_{AH} = [0\ ,\ 2 \times PWMTM \times t_{SCLK}]$$

$$T_{AL} = (PWMTM - 2 \times (PWMCHA + PWMDT)) \times t_{SCLK}$$

$$Range\ of\ T_{AL} = [0\ ,\ 2 \times PWMTM \times t_{SCLK}]$$

and the corresponding duty cycles are:

$$d_{AH} = \frac{T_{AH}}{T_S} = \frac{1}{2} + \frac{PWMCHA - PWMDT}{PWMTM}$$

$$d_{AL} = \frac{T_{AL}}{T_S} = \frac{1}{2} - \frac{PWMCHA + PWMDT}{PWMTM}$$

Obviously, negative values of T_{AH} and T_{AL} are not permitted, and the minimal permissible value is zero (corresponding to a 0% duty cycle). In a similar fashion, the maximal value is T_s, which is the PWM switching period that corresponds to a 100% duty cycle.

Figure 14-4 shows the output signals from the Three-Phase PWM Timing Unit in double-update mode. This figure illustrates a completely general case in which the switching frequency, dead time, and duty cycle are changed in the second half of the PWM period. Of course, the same value for any or all of these quantities may be used in both halves of the PWM cycle. However, it can be seen that there is no guarantee that a symmetrical PWM signal will be produced by the Three-Phase PWM Timing Unit in double-update mode. Additionally, it is seen that the dead time is inserted into the PWM signals similarly to single-update mode.

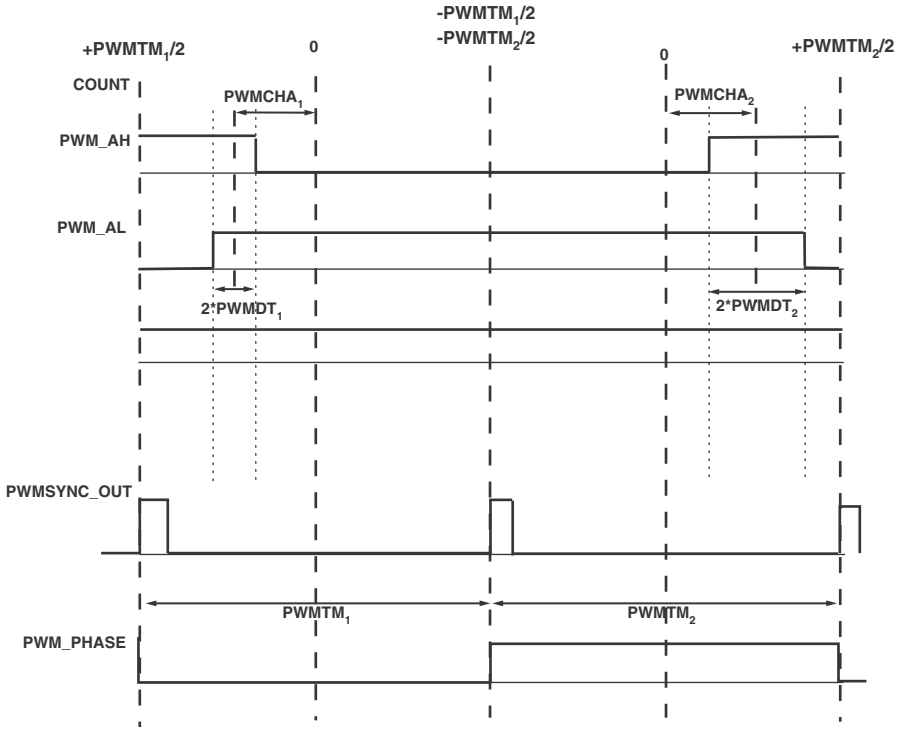


Figure 14-4. Typical PWM Outputs of Three-Phase Timing Unit in Double-Update Mode (Active Low Waveforms)

In general, the on-times (active low) of the PWM signals over the full PWM period in double-update mode can be defined as:

$$T_{AH} = \left(\frac{PWMTM_1}{2} + \frac{PWMTM_2}{2} + PWMCHA_1 + PWMCHA_2 - PWMDT_1 - PWMDT_2\right) \times t_{SCLK}$$

$$T_{AL} = \left(\frac{PWMTM_1}{2} + \frac{PWMTM_2}{2} - PWMCHA_1 - PWMCHA_2 - PWMDT_1 - PWMDT_2\right) \times t_{SCLK}$$

$$T_S = (PWMTM_1 + PWMTM_2) \times t_{SCLK}$$

where subscript 1 refers to the value of that register during the first half cycle and subscript 2 refers to the value during the second half cycle. The corresponding duty cycles are:

$$d_{AH} = \frac{T_{AH}}{T_S} = \frac{1}{2} + \frac{(PWMCHA_1 + PWMCHA_2 - PWMDT_1 - PWMDT_2)}{(PWMTM_1 + PWMTM_2)}$$

$$= \frac{T_{AL}}{T_S} = \frac{1}{2} - \frac{(PWMCHA_1 + PWMCHA_2 + PWMDT_1 + PWMDT}{(PWMTM_1 + PWMTM_2)}$$

since for the completely general case in double-update mode, the switching period is given by:

$$T_S = (PWMTM_1 + PWMTM_2) \times t_{SCLK}$$

Again, the values of T_{AH} and T_{AL} are constrained to lie between zero and T_s. Similar PWM signals to those illustrated in Figure 14-2 on page 14-5 and in Figure 14-3 on page 14-16 can be produced on the BH, BL, CH, and CL outputs by programming the PWM_CHB and PWM_CHC registers in a manner identical to that described for PWM_CHA.

Special Consideration for PWM Operation in Over-Modulation

The Three-Phase PWM Timing Unit can produce PWM signals with variable duty-cycle values at the PWM output pins. At the extremities of the modulation process, both 0% and 100% modulation (termed *full off mode* and *full on mode*, respectively) are possible. In between, for other duty cycle values, the operation is termed *normal modulation*.

- **Full On Mode**. The PWM for any pair of PWM signals is said to operate in full on mode when the desired high side output of the Three-Phase PWM Timing Unit is in the "on" state (low or high as specified by PWM_POLARITY bit of the PWM_CTRL register) between successive PWM_SYNC rising edges. This state may be entered by virtue of the commanded duty cycle values (in conjunction with the PWM_DT register).
- **Full Off Mode**. The PWM for any pair of PWM signals is said to operate in full off mode when the desired high side output of the Three-Phase PWM Timing Unit is in the "off" state (high or low as specified by the PWM_POLARITY bit of the PWM_CTRL register) between successive PWM_SYNC pulses. This state may be entered by virtue of the commanded duty cycle values (in conjunction with the PWM_DT register).
- **Normal Modulation**. The PWM for any pair of PWM signals is said to operate in normal modulation when the desired output duty cycle is other than 0% or 100% between successive PWM_SYNC pulses.

Certain situations exist whereby it is necessary to transition into or out of full on mode or full off mode in order to insert additional "emergency dead time" delays to prevent potential shoot-through conditions in the inverter. Crossover usage also can potentially cause outputs to violate shoot-through condition criteria, as described in "Crossover Feature" on page 14-25. These transitions are detected automatically and, if

appropriate for safety, emergency dead-time is inserted to prevent shoot-through conditions.

The insertion of additional emergency dead time into one of the PWM signals of a given pair during these transitions is necessary only when both PWM signals are required to toggle within a dead time of each other. The additional emergency dead time delay is inserted into the PWM signal that is toggling into the “on” state. In effect, the turn on (if turning on during this dead time region) of this signal is delayed by an amount ($2*PWM\_DT*t_{SCLK}$) from the rising edge of the opposite output. After this delay, the PWM signal is allowed to turn on, provided the desired output is still scheduled to be in the on state after the emergency dead time delay.

Figure 14-5 illustrates two examples of such transitions. In the top half (marked A) of Figure 14-5, no special action (dead time) is needed when transitioning from normal modulation to full on mode at the half cycle boundary in double-update mode. However, in the bottom half (marked B) of Figure 14-5, when transitioning from normal modulation into full off mode at the same boundary, it can be seen that an additional emergency dead time is necessary (inserted by the PWM Controller). Clearly, this inserted dead time is different from the normal dead time as it is impossible to move one of the switching events back in time, because this would take it into the previous modulation cycle. Therefore, the entire emergency dead time is inserted by delaying the turn on of the appropriate signal by the full amount.

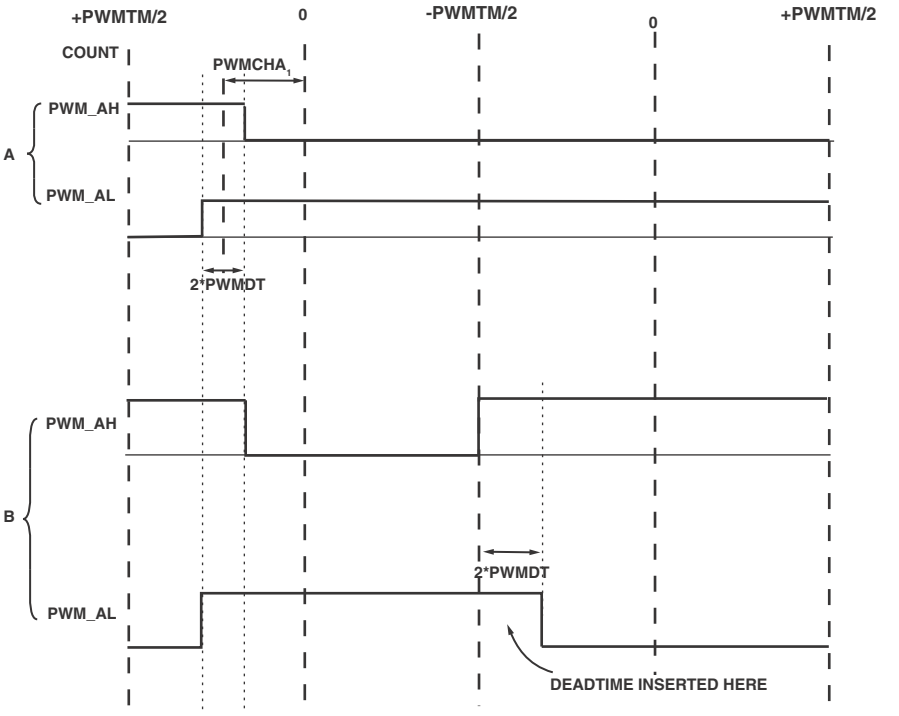


Figure 14-5. Examples of Transitioning from Normal Modulation to Full On Mode (A) or Full Off Mode (B)

Three-Phase PWM Timing Unit Operation

The internal operation of the PWM Controller is controlled by the Three-Phase PWM Timing Unit, which is clocked at the system clock rate with period t_{SCLK}. The operation of the Three-Phase PWM Timing Unit over one full PWM period is illustrated in Figure 14-6.

During the first half cycle (when the PWM_PHASE bit of the PWM_STAT register is cleared), the Three-Phase PWM Timing Unit decrements from +PWMTM/2 to -PWMTM/2 using a two's complement count. Then the

count direction changes, and the unit increments from -PWMTM/2 to the +PWMTM/2 value.

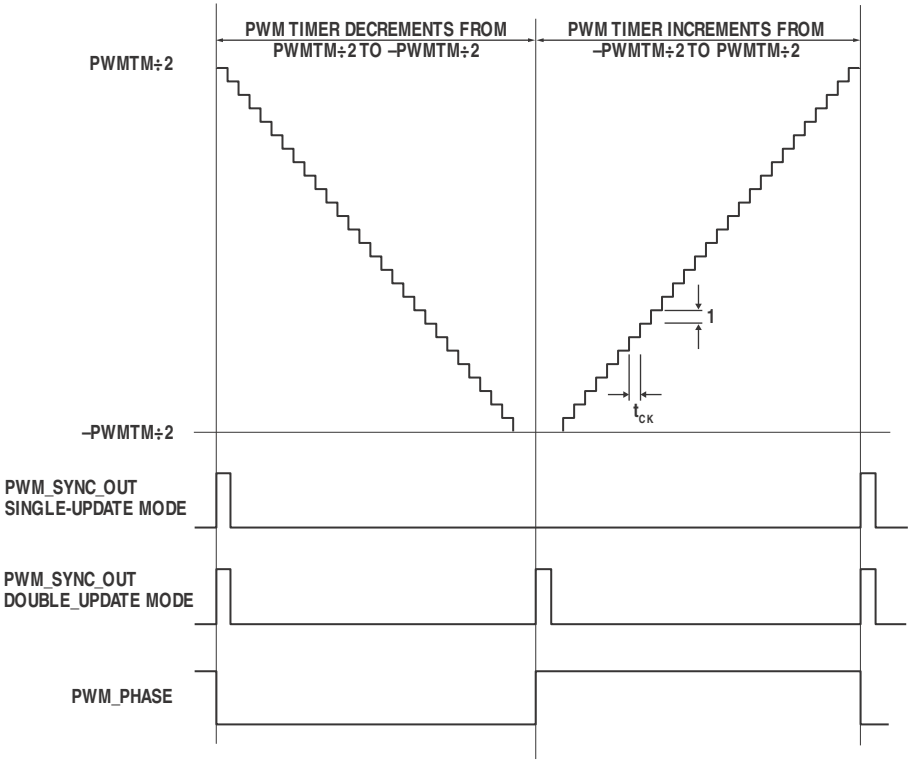


Figure 14-6. Operation of Internal PWM Timer

Figure 14-6 also shows the PWM SYNC pulses during single-update mode and double-update mode. Clearly, an additional PWM SYNC pulse is generated at the midpoint of the PWM cycle in double-update mode. If the value of the PWM_TM register is altered at the midpoint in double-update mode, the duration of the second half period (when the PWM_PHASE bit of the PWM_STAT register is set) may differ from that of the first half cycle. PWM_TM is double-buffered; a change in one half of the PWM switching period will only take effect in the next half period.

Effective PWM Accuracy

The PWM Controller has 16-bit resolution, but accuracy depends on the PWM period. In single-update mode, the same values of PWM_CHA, PWM_CHB, and PWM_CHC define the on-times in both half cycles of the PWM period. As a result, the effective accuracy of the PWM generation process is $2 \times t_{SCLK}$ (20 ns for a 100 MHz f_{SCLK}). Incrementing one of the duty cycle registers by 1 changes the resultant on-time of the associated PWM signals by t_{SCLK} in each half period ($2 \times t_{SCLK}$ for the full period). In double-update mode, improved accuracy is possible since different values of the duty cycles registers are used to specify the on-times in both the first half and second half of the PWM period. As a result, it is possible to adjust the on-time over the entire period in increments of t_{SCLK}. This corresponds to an effective PWM accuracy of t_{SCLK} in double-update mode (10 ns for a 100 MHz f_{SCLK}). The minimum achievable PWM switching frequency at a given PWM accuracy is shown in Table 14-1 for SCLK = 100 MHz.

Table 14-1. Minimum Achievable PWM Frequency versus Bit Resolution for SCLK = 100 MHz

Resolution (bits)	PWM Frequency (kHz) in Single-Update Mode	PWM Frequency (kHz) in Double-Update Mode
8	195.3	390.6
9	97.7	195.3
10	48.8	97.7
11	24.4	48.8
12	12.2	24.4
13	6.1	12.2
14	3.05	6.1

Switched Reluctance Mode

A general-purpose mode utilizing independent edge placement of upper and lower signals of each of the three PWM channels is incorporated into the Three-Phase PWM Timing Unit. This mode is provided for SR motor operation and is described in detail in "Switched Reluctance (SR) Mode" on page 14-31.

Output Control Unit

The operation of the Output Control Unit is controlled by the 9-bit read/write PWM_SEG register (on page 14-45) that controls two distinct features that are useful in the control of ECMs or BDCMs.

Crossover Feature

The PWM_SEG register contains three crossover bits—one for each pair of PWM outputs. Setting the AHAL_XOVR bit of the PWM_SEG register enables crossover mode for the AH/AL pair of PWM signals, setting BHBL_XOVR enables crossover on the BH/BL pair, and setting CHCL_XOVR enables crossover on the CH/CL pair. If crossover mode is enabled for any pair of PWM signals, the high-side PWM signal (for example, AH) from the Three-Phase PWM Timing Unit is diverted to the associated low-side output of the Output Control Unit so that the signal ultimately appears at the AL pin. The corresponding low-side output of the Three-Phase PWM Timing Unit is also diverted to the complementary high-side output of the Output Control Unit so that the signal appears at the AH pin. Following a reset, the three crossover bits are cleared, disabling crossover mode on all three pairs of PWM signals. Even though crossover is considered an output control feature, dead time insertion occurs after crossover transitions (as necessary to eliminate shoot-through safety issues).

Mode Bits (POLARITY and SRMODE)

PWM_POLARITY and PWM_SRMODE are programmable bits of the PWM_CTRL register.

The incorrect programming of these two mode-select signals can have destructive consequences on the external power inverter connected to the PWM unit. Since PWM_POLARITY and PWM_SRMODE are software programmable bits, accidental power inverter shoot-through current may occur from incorrect programming.

Output Enable Function

The PWM_SEG register also contains six bits (bits 0 to 5) that can be used to individually enable or disable each of the six PWM outputs. The PWM signal of the AL pin is enabled by clearing the AL_EN bit of the PWM_SEG register, the AH_EN bit controls AH, the BL_EN bit controls BL, the BH_EN bit controls BH, the CL_EN bit controls CL, and the CH_EN bit controls the CH output. If the associated bit of the PWM_SEG register is set, the corresponding PWM output is disabled irrespective of the value of the corresponding duty cycle register. This PWM output signal will remain in the off state as long as the corresponding enable/disable bit of the PWM_SEG register is set. This output enable function is implemented after the crossover function. Following a reset, all six enable bits of the PWM_SEG register are cleared so that all PWM outputs are enabled by default. In a manner identical to the duty cycle registers, the PWM_SEG register is latched on the rising edge of the PWM_SYNC signal so that changes to this register only become effective at the start of each PWM cycle in single-update mode. In double-update mode, the PWM_SEG register can also be updated at the midpoint of the PWM cycle.

Brushless DC Motor (Electronically Commutated Motor) Control

In the control of an electronically commutated motor (ECM), only two inverter legs are switched at any time. Often, the high-side device in one leg must be switched on at the same time as the low-side driver in a second leg. Therefore, by programming identical duty cycles values for two PWM channels (for example, PWM_CHA = PWM_CHB) and setting the BHBL_XOVR bit of the PWM_SEG register to crossover the BH/BL pair if PWM signals, it is possible to turn on the high-side switch of phase A and the low-side switch of phase B at the same time.

In ECM control, usually the third inverter leg (phase C in this example) is disabled for a number of PWM cycles. This is implemented by disabling the CH and CL outputs by setting the CH_EN and CL_EN bits of the PWM_SEG register.

This is illustrated in Figure 14-7 where it can be seen that both the AH and BL signals are identical (since PWM_CHA = PWM_CHB and the crossover bit for phase B is set). In addition, the other four signals (AL, BH, CH, and CL) are disabled by setting the appropriate enable/disable bits of the PWM_SEG register.

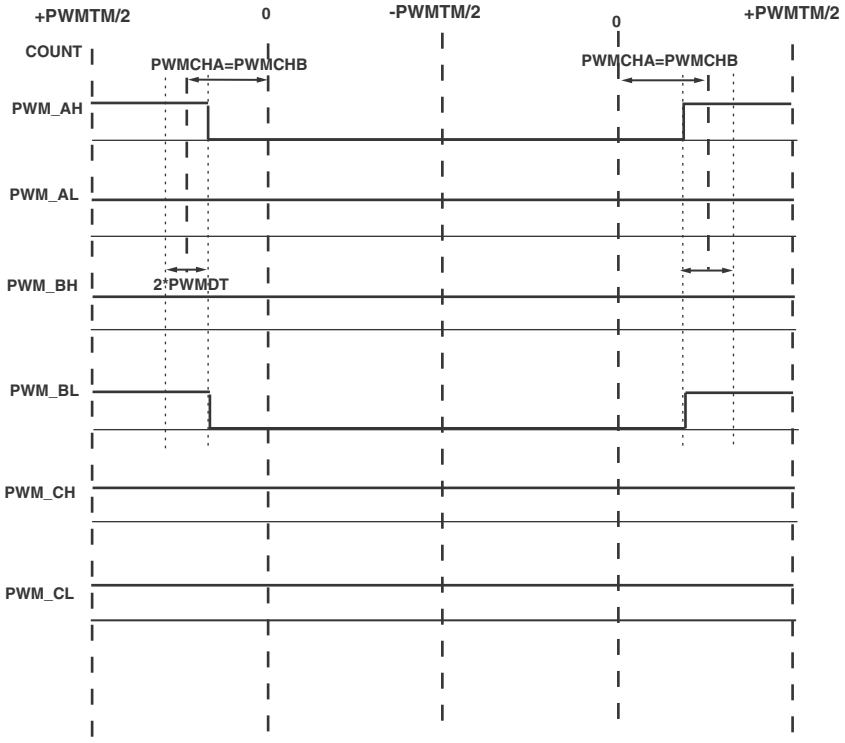


Figure 14-7. Example of Active Low Signals for ECM Control

For the situation illustrated in Figure 14-7, an appropriate value for the PWM_SEG register is 0x00A7. In normal ECM operation, each inverter leg is disabled for certain lengths of time, such that the PWM_SEG register is changed, based upon the position of the rotor shaft (motor commutation).

Gate Drive Unit

The Gate Drive Unit is described in the following sections:

- "High-Frequency Chopping"
- "PWM Polarity Control" on page 14-30

High-Frequency Chopping

The Gate Drive Unit of the PWM Controller simplifies the design of isolated gate drive circuits for PWM inverters. If a transformer-coupled power device gate drive amplifier is used, the active PWM signal must be chopped at a high frequency. The 10-bit read/write PWM_GATE register allows you to specify this high-frequency chopping mode.

Chopped active PWM signals may be required for high-side drivers only, for low-side drivers only, or for both high-side and low-side switches. Therefore, independent control of this mode for both high- and low-side switches is included with two separate control bits (CHOPHI and CHOPLO) in the PWM_GATE register.

Typical PWM output signals with high-frequency chopping enabled on both high- and low-side signals are shown in Figure 14-8. Chopping the high-side PWM outputs (AH, BH, and CH) is enabled by setting the CHOPHI bit of the PWM_GATE register; chopping the low-side PWM outputs (AL, BL, and CL) is enabled by setting the CHOPLO bit of the PWMGATE register. The high-frequency chopping frequency is controlled by the 8-bit word placed in bits 0 to 7 (GDCLK) of the PWM_GATE register. The period of this high-frequency carrier is:

$$T_{chop} = [4 \times (GDCLK + 1)] \times t_{SCLK}$$

and the chopping frequency is therefore an integral subdivision of the system clock frequency:

$$f_{chop} = \frac{f_{SCLK}}{[4 \times (GDCLK + 1)]}$$

The GDCLK value may range from 0 to 255, which corresponds to a programmable chopping frequency rate from 97.7 kHz to 25 MHz for a 100 MHz f_{SCLK} rate. The gate drive features must be programmed before operation of the PWM Controller and typically are not changed during normal operation of the PWM Controller. Following a reset, all bits of the PWM_GATE register are cleared so that high-frequency chopping is disabled, by default.

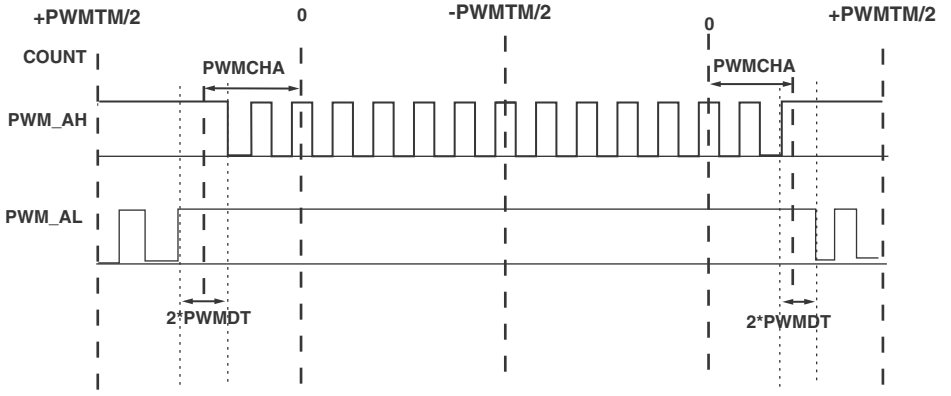


Figure 14-8. Example of Active Low PWM Signals for Gate Chopping

PWM Polarity Control

The polarity of the PWM signals produced at output pins AH to CL can be programmed via the PWM_POLARITY bit of the PWM_CTRL register. Setting

this bit to 0 selects active low PWM outputs, such that a low level is interpreted as a command to turn on the associated power device. Conversely, setting the PWM_POLARITY bit to 1 selects active high PWM outputs, such that a high level at the PWM outputs turns on the associated power devices. The status of the polarity may be read from the PWM_POL bit of the PWM_STAT register, where a zero indicates a measured low level at the PWM_POLARITY bit.

Output Control Feature Precedence

It is important to understand the order in which output control features are applied to the PWM signal. The following hierarchy indicates the order (from most important to least important) in which signal features are applied to the PWM output signal.

1. Channel duty generation
2. Channel crossover
3. Low-side invert
4. Output enable
5. Emergency dead time insertion
6. Active signal chopping
7. Polarity

Switched Reluctance (SR) Mode

The PWM Controller provides a switched reluctance (SR) mode that is enabled by setting the PWM_SRMODE bit in the PWM_CTRL register to 0. This mode is not enabled by default. The state of this switched reluctance mode may be read from the PWM_SR bit of the PWM_STAT register. If the PWM_SRMODE bit is high (such that SR mode is disabled) the PWM_SR bit of the PWM_STAT register is set (indicating that the mode is disabled).

Conversely, if the PWM_SRMODE bit is low and SR mode is enabled, the PWM_SR bit of PWM_STAT register is cleared.

Since this is a software programmable bit, be careful not to write it to an active state in a non-SR mode system and cause shoot-through at the power inverters, possibly leading to an unsafe situation.

In the typical power converter configuration for switched or variable reluctance motors, the motor winding is connected between the two power switches of a given inverter leg. Therefore, to allow for a complete circuit in the motor winding, it is necessary to turn on both switches at the same time.

SR mode provides four mode types: hard chop, alternate chop, soft chop-bottom on, and soft chop-top on (see Table 14-2). Three registers (PWM_CHAL, PWM_CHBL, and PWM_CHCL) are used to define edge placement of the low side of the channel. The PWM_DT register, which is not used, is internally forced to 0 by hardware when SR mode is active. The four switched reluctance (SR) chop modes are specified via three bits (PWM_SR_LSI_A, PWM_SR_LSI_B, and PWM_SR_LSI_C) of the PWM_LSI register, full on mode, and full off mode.

The PWM_CHA and PWM_CHAL registers are programmed independently; PWM_CHA specifies edge placement for the high side of the channel, and PWM_CHAL specifies edge placement for the low side of the channel. Similarly, the PWM_CHB and PWM_CHBL pair, and the PWM_CHC and PWM_CHCL pair, respectively, specify high-side and low-side edge placement.

Figure 14-9 shows the four SR mode types as active-high PWM output signals, and Table 14-2 describes the four mode types.

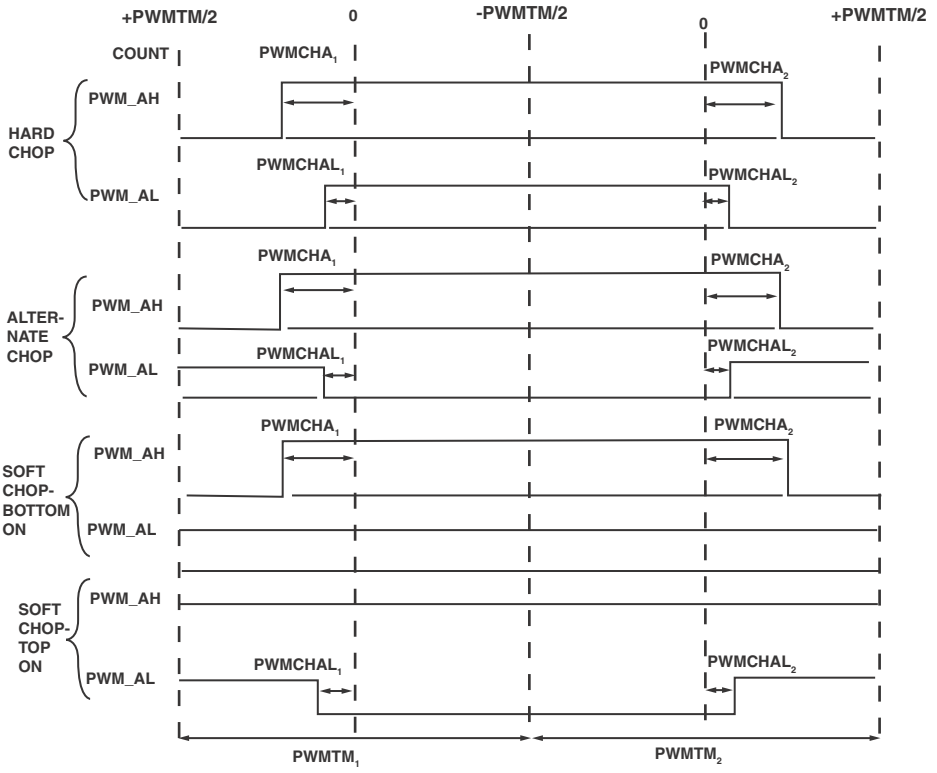


Figure 14-9. Four SR Mode Types

Table 14-2. Switched Reduction Mode (SR Mode) Types

Mode	Description
Hard chop	Contains independently programmed rising edges of channels' high and low signals in the same PWM half cycle, and both contain independently programmed falling edges in the next PWM half cycle. The PWM_CHA duty register is used for the high channel, and the PWM_CHAL duty register is used for the low channel. A similar structure is present for the B and C channels.
Alternate chop	Similar to normal PWM operation, but the PWM channel high and low signal edges are opposite and are independently programmed. The PWM_CHA duty register is used for the high channel, and the PWM_CHAL duty register is used for the low channel. A similar structure is present for the B and C channels. The PWM_CTRL and PWM_LSI registers are used to independently invert the low side of each PWM channel. The low-side invert is the only difference between hard chop mode and alternate chop mode
Soft chop-bottom on	Utilizes a 100% duty on the low side of the channel. Similar to hard chop mode, the PWM_CHA duty register is used for the high channel and the PWM_CHAL duty register is used for the low channel. A similar structure is present for the B and C channels.
Soft chop-top on	Utilizes a 100% duty on the high side of the channel. Similar to hard chop mode, the PWM_CHA duty register is used for the high channel and the PWM_CHAL duty register is used for the low channel. A similar structure is present for the B and C channels.

PWM Sync Operation

The PWM sync can be internally generated as a function of the PWM_TM and PWM_SYNCWT register values, or the PWM sync can be input externally. Multiple PWM configurations can be established, each of which can operate with its own independent PWM sync (or from its own external PWM sync signal or a shared external PWM sync signal). The external PWM sync can be synchronous to the internal clock, as in the case of a primary PWM Controller generating an internal PWM_SYNC signal that drives a secondary PWM Controller's PWM_SYNC pin. The external PWM sync can also be asynchronous to the internal clock, as is typically the case of an off-chip PWM_SYNC signal used to drive each PWM Controller's PWM_SYNC pin.

Internal PWM SYNC Generation

The PWM Controller produces an output PWM synchronization pulse at a rate equal to the PWM switching frequency in single-update mode and at twice the PWM frequency in double-update mode. This pulse is available for external use at the PWM_SYNC pin. The width of this PWM SYNC pulse is programmable by the 10-bit read/write PWM_SYNCWT register. The width of the PWM SYNC pulse (TPWM_SYNC) is given by:

$$T_{PWMSYNC} = t_{SCLK} \times (PWMSYNCWT + 1)$$

so that the width of the pulse is programmable from t_{SCLK} to 1024*t_{SCLK} (corresponding to 10 ns to 10.24 µs for an f_{SCLK} rate of 100 MHz). Following a reset, the PWM_SYNCWT register contains 0x3FF (1023 decimal) so that the default PWM_SYNC width is 10.24 µs, again for an f_{SCLK} of 100 MHz.

External PWM SYNC Generation

By setting the PWM_EXTSYNC bit of the PWM_CTRL register, the PWM is set up in a mode to expect an external PWM SYNC on the PWM_SYNC pin. The external sync should be synchronized by setting the PWM_SYNCSEL bit of the PWM_CTRL register to 0, which assumes the selected external PWM SYNC is asynchronous.

The external PWM SYNC period is expected to be an integer multiple of the internal PWM SYNC period. When the rising edge of the external PWM_SYNC is detected, the PWM Controller is restarted at the beginning of the PWM cycle. If the external PWM SYNC period is not an integer multiple of the internal PWM SYNC, the behavior of the PWM channel outputs will be clipping. Note that a small amount of jitter inherent in the synchronization logic cannot be avoided when the external PWM SYNC is synchronized.

The latency from PWM_SYNC to the effect in PWM outputs is 3 SCLK cycles in synchronous mode and 5 SCLK cycles in asynchronous mode.

In external sync pulse mode, do not allow changes in PWM_SYNCSEL (which selects between asynchronous/synchronous external sync pulse) ± 10 SCLK cycles of the toggling of an external sync pulse. If this rule is not followed, unexpected behavior may occur.

PWM Shutdown and Interrupt Control Unit

In the event of an external fault condition, it is essential that the PWM Controller be shut down instantaneously in a safe fashion. A falling edge on the $\overline{\text{PWM\_TRIP}}$ pin (assuming it is not disabled by the PWM_TRIP_DSBL bit of the PWM_CTRL register) provides an instantaneous, asynchronous (independent of the processor clock) shutdown of the PWM controller. All six PWM outputs are placed in the off state (as defined by the PWM_POLARITY bit of the PWM_CTRL register). However, the PWM_SYNC pulse occurs if it was previously enabled, and the associated interrupt is also not stopped.

The processor's $\overline{\text{PWM\_TRIP}}$ signal should have an external pull-down resistor; if the pin becomes disconnected, the PWM Controller will be disabled. The state of the $\overline{\text{PWM\_TRIP}}$ pin can be read from the PWM_TRIP bit of the PWMSTAT register.

On the occurrence of a PWM shutdown command (or from a signal on the $\overline{\text{PWM\_TRIP}}$ pin), a PWM_TRIP interrupt will be generated if enabled. In addition, if PWM_SYNC_EN is enabled in the PWM_CTRL register, the PWM_SYNC pulse will continue to appear at the output pin. Following a PWM shutdown, the PWM can be re-enabled (by a PWM_TRIP interrupt service routine, for example) by writing to the PWM_EN bit of the PWM_CTRL register. The PWM Controller will restart in a manner identical to that prior to the PWM shutdown, provided that the external fault has been cleared and $\overline{\text{PWM\_TRIP}}$ returned to a high level. That is, except for the PWM_EN bit in the PWM_CTRL register, all PWM registers retain their values during the PWM shutdown.

The dead time counters will be reset when a trip occurs, and the user is expected to restart the PWM only after waiting the required dead time. If restarting a PWM immediately after trip, for high dead time period cases, the dead time will not be met.

Do not allow changes in the PWM_TRIP_DSBL bit of the PWM_CTRL register (which is to select between trip enable and disable) ± 10 SCLK cycles of the toggling of an external trip pulse. If this rule is not followed, unexpected behavior may occur.

Between the time that the PWM_EN bit is written to 0 and the time the waveforms are disabled, the latency is 2 SCLK cycles. After enabling the PWM_EN bit, output waveforms will begin to appear from the next PWM pulse.

PWM Registers

Descriptions and bit diagrams for each of the PWM memory-mapped registers (MMRs) are provided in the following sections.

Table 14-3. PWM Registers

Name	Description
PWM_CTRL	PWM control register on page 14-38
PWM_STAT	PWM status register on page 14-40
PWM_TM	PWM period register on page 14-41
PWM_DT	PWM dead time register on page 14-42
PWM_GATE	PWM chopping control on page 14-42
PWM_CHA	PWM channel A duty control on page 14-43
PWM_CHB	PWM channel B duty control on page 14-43
PWM_CHC	PWM channel C duty control on page 14-43
PWM_SEG	PWM crossover and output enable on page 14-45

Table 14-3. PWM Registers (Cont'd)

Name	Description
PWM_SYNCWT	PWM sync pulse width control on page 14-47
PWM_CHAL	PWM channel AL duty control (SR mode only) on page 14-47
PWM_CHBL	PWM channel BL duty control (SR mode only) on page 14-47
PWM_CHCL	PWM channel CL duty control (SR mode only) on page 14-47
PWM_LSI	PWM low side invert (SR mode only) on page 14-49
PWM_STAT2	PWM simulation status register on page 14-49

PWM Control (PWM_CTRL) Register

The PWM_CTRL register is used for configuration of the PWM block. Bit diagrams and descriptions are provided in Figure 14-10 and Table 14-4.

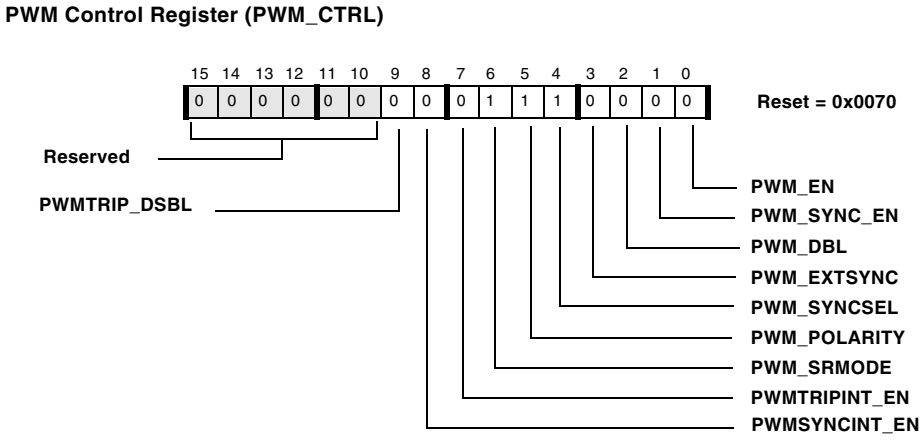


Figure 14-10. PWM Control Register

Table 14-4. PWM_CTRL Register

Bit	Name	Function	Type	Default
0	PWM_EN Hardware modifiable bit.	PWM enable 0 = disabled 1 = enabled reset by $\overline{\text{PWM\_TRIP}}$	RW	0
1	PWM_SYNC_EN	PWM sync enable 0 = disabled 1 = enabled	RW	0
2	PWM_DBL	Double-update mode 0 = single-update mode 1 = double-update mode	RW	0
3	PWM_EXTSYNC	External sync 0 = internal sync 1 = external sync	RW	0
4	PWM_SYNCSEL	External sync select 0 = asynchronous 1 = synchronous	RW	1
5	PWM_POLARITY	PWM output polarity 1 = active high 0 = active low	RW	1
6	PWM_SRMODE	PWM SR Mode 0 = enabled 1 = disabled	RW	1
7	PWMTRIPINT_EN	Interrupt enable for trip 1 = enabled 0 = disabled	RW	0
8	PWMSYNCINT_EN	Interrupt enable for sync 1 = enabled 0 = disabled	RW	0
9	PWMTRIP_DSBL	Disable for trip input 1 = disabled 0 = enabled	RW	0
15:10	Reserved			0

PWM Status (PWM_STAT) Register

The PWM_STAT register provides status information regarding PWM operation. Bit diagrams and descriptions are provided in Figure 14-11 and Table 14-5.

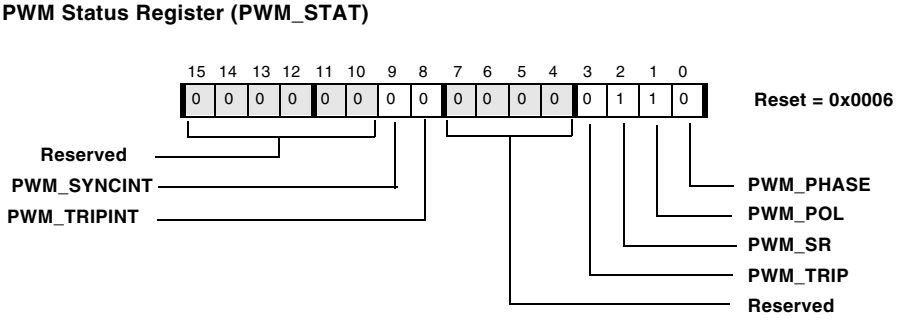


Figure 14-11. PWM Status Register

Table 14-5. PWM_STAT Register

Bit	Name	Function	Type	Default
0	PWM_PHASE	PWM phase 0 = first half 1 = second half	RO	0
1	PWM_POL	PWM polarity 1 = active high 0 = active low	RO	1
2	PWM_SR	PWM SR mode 0 = active 1 = inactive	RO	1
3	PWM_TRIP	PWM trip	RO	0
7:4	Reserved			0
8	PWM_TRIPINT	PWM trip interrupt (via hardware pin or software)	R/W1C	0

Table 14-5. PWM_STAT Register (Cont'd)

Bit	Name	Function	Type	Default
9	PWM_SYNCINT	PWM sync interrupt	R/W1C	0
15:10	Reserved			0

PWM Period (PWM_TM) Register

The `PWM_TM` register controls the switching frequency of the generated PWM patterns. Bit diagrams and descriptions are provided in Figure 14-12 and Table 14-6.

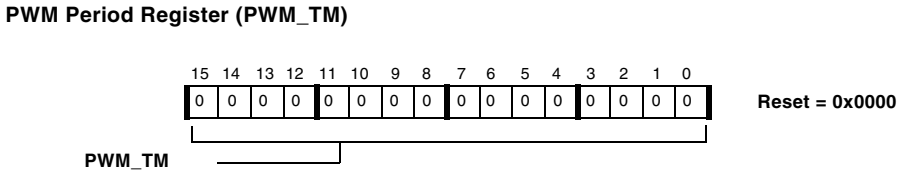


Figure 14-12. PWM Period Register

Table 14-6. PWM_TM Register

Bit	Name	Function	Type	Default
15:0	PWM_TM	PWM period (unsigned)	RW	0

PWM Dead Time (PWM_DT) Register

The PWM_DT register controls the dead time interval of the generated PWM patterns. Bit diagrams and descriptions are provided in Figure 14-13 and Table 14-7.

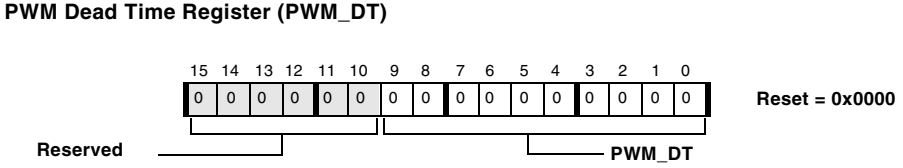


Figure 14-13. PWM Dead Time Register

Table 14-7. PWM_DT Register

Bit	Name	Function	Type	Default
9:0	PWM_DT	PWM dead time (unsigned)	RW	0
15:10	Reserved			0

PWM Chopping Control (PWM_GATE) Register

The PWM controller permits the mixing of the output PWM signals with a high-frequency chopping signal. The features of gate-drive-chopping mode are controlled

by the PWM_GATE register. Bit diagrams and descriptions are provided in Figure 14-14 and Table 14-8.

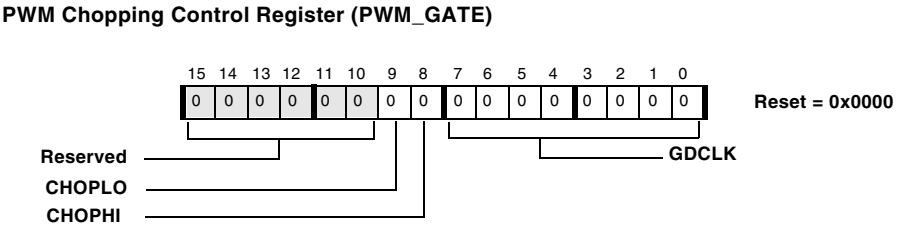


Figure 14-14. PWM Chopping Control Register

Table 14-8. PWM_GATE Register

Bit	Name	Function	Type	Default
7:0	GDCLK	PWM gate chopping period (unsigned)	RW	0
8	CHOPHI	Gate chopping enable high side	RW	0
9	CHOPLO	Gate chopping enable low side	RW	0
15:10	Reserved			0

PWM Channel A, B, C Duty Control (PWM_CHA, PWM_CHB, PWM_CHC) Registers

The three duty-cycle control registers (`PWM_CHA`, `PWM_CHB`, and `PWM_CHC`) directly control the duty cycles of the three pairs of PWM signals. Bit diagrams and descriptions for each are provided in Figure 14-15 through Figure 14-17, and Table 14-9 through Table 14-11.

PWM Channel A Duty Control Register (PWM_CHA)

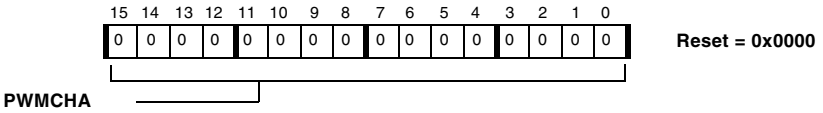


Figure 14-15. PWM Channel A Duty Control Register

Table 14-9. PWM_CHA Register

Bit	Name	Function	Type	Default
15:0	PWMCHA	Channel A duty (two's complement)	RW	0

PWM Channel B Duty Control Register (PWM_CHB)

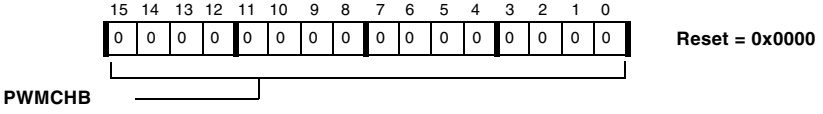


Figure 14-16. PWM Channel B Duty Control Register

Table 14-10. PWM_CHB Register

Bit	Name	Function	Type	Default
15:0	PWMCHB	Channel B duty (two's complement)	RW	0

PWM Channel C Duty Control Register (PWM_CHC)

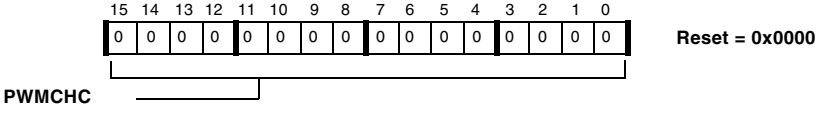


Figure 14-17. PWM Channel C Duty Control Register

Table 14-11. PWM_CHC Register

Bit	Name	Function	Type	Default
15:0	PWMCHC	Channel C duty (two's complement)	RW	0

PWM Crossover and Output Enable (PWM_SEG) Register

The PWM_SEG register controls output enabling of the high-side and low-side PWM outputs, and it also permits configuration of crossover mode for each output pair. Bit diagrams and descriptions are provided in Figure 14-18 and Table 14-12.

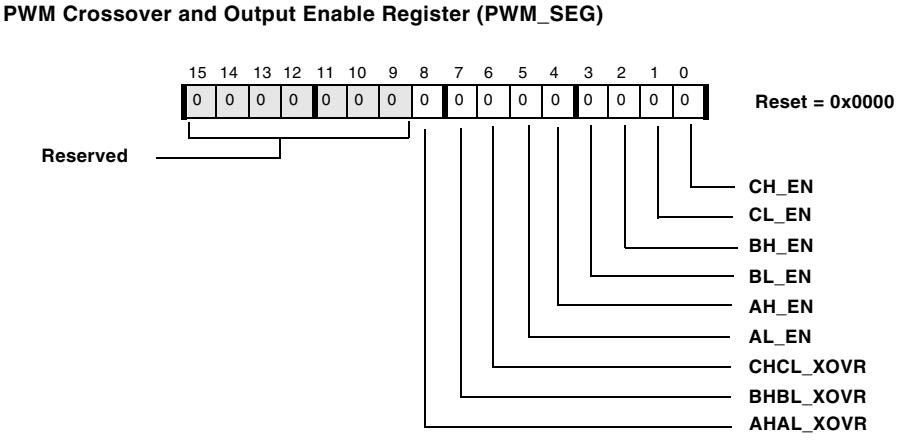


Figure 14-18. PWM Crossover and Output Enable Register

Table 14-12. PWM_SEG Register

Bit	Name	Function	Type	Default
0	CH_EN	CH output enable 1 = disabled 0 = enabled	RW	0
1	CL_EN	CL output enable 1 = disabled 0 = enabled	RW	0
2	BH_EN	BH output enable 1 = disabled 0 = enabled	RW	0
3	BL_EN	BL output enable 1 = disabled 0 = enabled	RW	0
4	AH_EN	AH output enable 1 = disabled 0 = enabled	RW	0
5	AL_EN	AL output enable 1 = disabled 0 = enabled	RW	0
6	CHCL_XOVR	Channel C output crossover 1 = XOVR 0 = not XOVR	RW	0
7	BHBL_XOVR	Channel B output crossover 1 = XOVR 0 = not XOVR	RW	0
8	AHAL_XOVR	Channel A output crossover 1 = XOVR 0 = not XOVR	RW	0
15:9	Reserved			0

PWM Sync Pulse Width Control (PWM_SYNCWT) Register

The PWM_SYNCWT register allows programming of the PWM_SYNC pulse width. Bit diagrams and descriptions are provided in Figure 14-19 and Table 14-13.

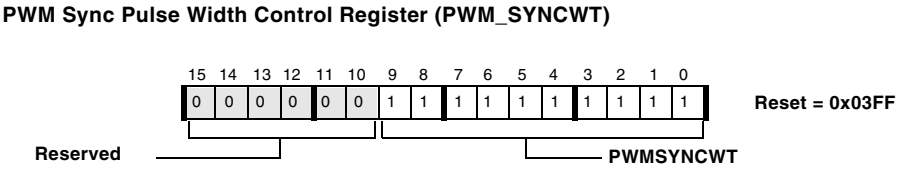


Figure 14-19. PWM Sync Pulse Width Control Register

Table 14-13. PWM_SYNCWT Register

Bit	Name	Function	Type	Default
9:0	PWMSYNCWT	PWM sync pulse width (unsigned)	RW	0x03FF
15:10	Reserved			0

PWM Channel AL, BL, CL Duty Control (PWM_CHAL, PWM_CHBL, PWM_CHCL) Registers

These registers are used to program duty cycle for a low-side channel in SR (switched reluctance) mode only. Bit diagrams and descriptions for each register are provided in Figure 14-20 through Figure 14-22, and Table 14-14 through Table 14-16.

PWM Channel AL Duty Control Register (PWM_CHAL)

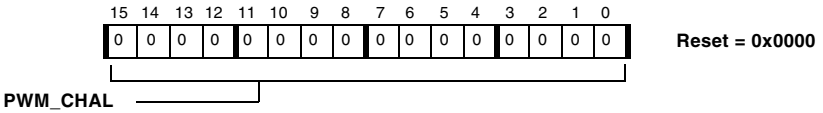


Figure 14-20. PWM Channel AL Duty Control Register

Table 14-14. PWM_CHAL Register

Bit	Name	Function	Type	Default
15:0	PWMCHAL	Channel A duty (two's complement)	RW	0

PWM Channel BL Duty Control Register (PWM_CHBL)

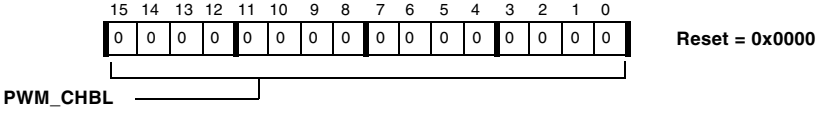


Figure 14-21. PWM Channel BL Duty Control Register

Table 14-15. PWM_CHBL Register

Bit	Name	Function	Type	Default
15:0	PWMCHBL	Channel B duty (two's complement)	RW	0

PWM Channel CL Duty Control Register (PWM_CHCL)

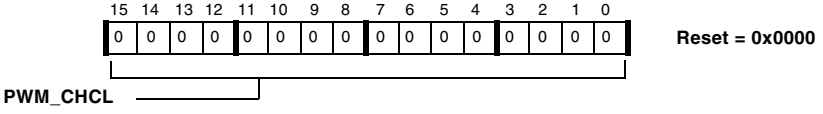


Figure 14-22. PWM Channel CL Duty Control Register

Table 14-16. PWM_CHCL Register

Bit	Name	Function	Type	Default
15:0	PWM_CHCL	Channel C duty (two's complement)	RW	0

PWM Low Side Invert (PWM_LSI) Register

The PWM_LSI register is used for specifying switched reluctance (SR) chop modes. Bit diagrams and descriptions are provided in Figure 14-23 and Table 14-17.

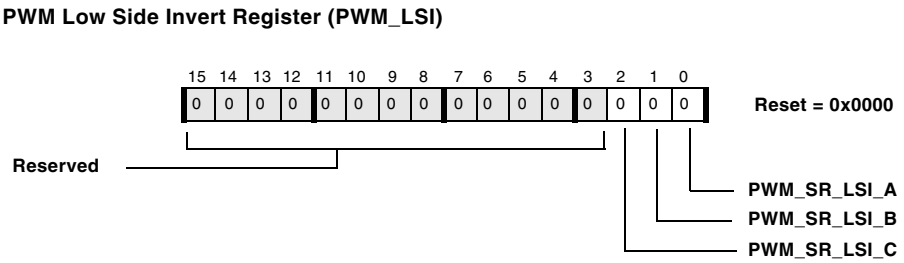


Figure 14-23. PWM Low Side Invert Register

Table 14-17. PWM_LSI Register

Bit	Name	Function	Type	Default
0	PWM_SR_LSI_A	PWM SR mode low side invert channel A	RW	0
1	PWM_SR_LSI_B	PWM SR mode low side invert channel B	RW	0
2	PWM_SR_LSI_C	PWM SR mode low side invert channel C	RW	0
15:3	Reserved			0

PWM Simulation Status (PWM_STAT2) Register

The PWM_STAT2 register provides a way to observe the status of the PWM high-side and low-side output channels via software. This can be useful for

debug operation. Bit diagrams and descriptions are provided in Figure 14-24 and Table 14-18.

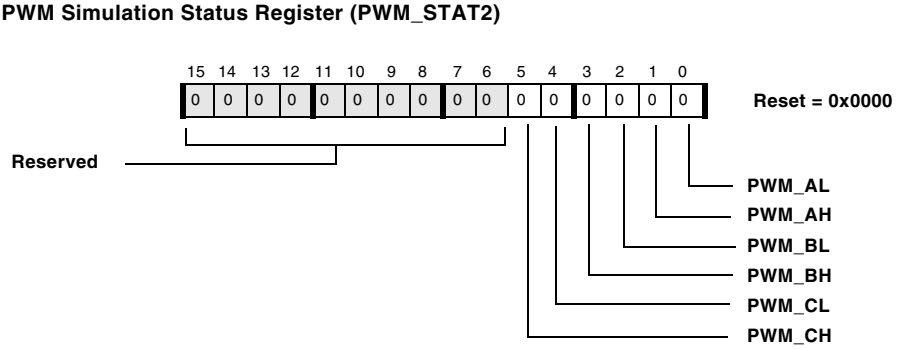


Figure 14-24. PWM Simulation Status Register

Table 14-18. PWM_STAT2 Register

Bit	Name	Function	Type	Default
0	PWM_AL	PWM_AL output signal for S/W observation	RO	0
1	PWM_AH	PWM_AH output signal for S/W observation	RO	0
2	PWM_BL	PWM_BL output signal for S/W observation	RO	0
3	PWM_BH	PWM_BH output signal for S/W observation	RO	0
4	PWM_CL	PWM_CL output signal for S/W observation	RO	0
5	PWM_CH	PWM_CH output signal for S/W observation	RO	0
15:6	Reserved			0

Unique Information for the ADSP-BF50x Processor

None.

15 UART PORT CONTROLLERS

This chapter describes the universal asynchronous receiver/transmitter (UART) modules and includes the following sections:

Overview

The ADSP-BF50x Blackfin processors feature multiple separate and identical UART modules.

ADSP-BF50x processors feature two UARTs, referred to as UART0 and UART1.

The UART modules are full-duplex peripherals compatible with PC-style industry-standard UARTs, sometimes called Serial Controller Interfaces (SCI). The UARTs convert data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, bit rate, and parity generation options.

Features

Each UART includes these features:

- 5 – 8 data bits
- 1 or 2 stop bits (1 1/2 in 5-bit mode)
- Even, odd, and sticky parity bit options
- Additional 4-stage receive FIFO with programmable threshold interrupt
- Flexible transmit and receive interrupt timings
- 3 interrupt outputs for reception, transmission, and status
- Independent DMA operation for receive and transmit
- Programmable automatic RTS/CTS hardware flow control on UART1
- False start bit detection
- SIR IrDA operation mode
- Internal loop back
- Improved bit rate granularity

The UARTs are logically compliant to EIA-232E, EIA-422, EIA-485 and LIN standards, but usually require external transceiver devices to meet electrical requirements. In IrDA® (Infrared Data Association) mode, the UARTs meet the half-duplex IrDA SIR (9.6/115.2 Kbps rate) protocol.

Interface Overview

Figure 15-1 shows a simplified block diagram of one UARTx module and how it interconnects to the Blackfin architecture and to the outside world.

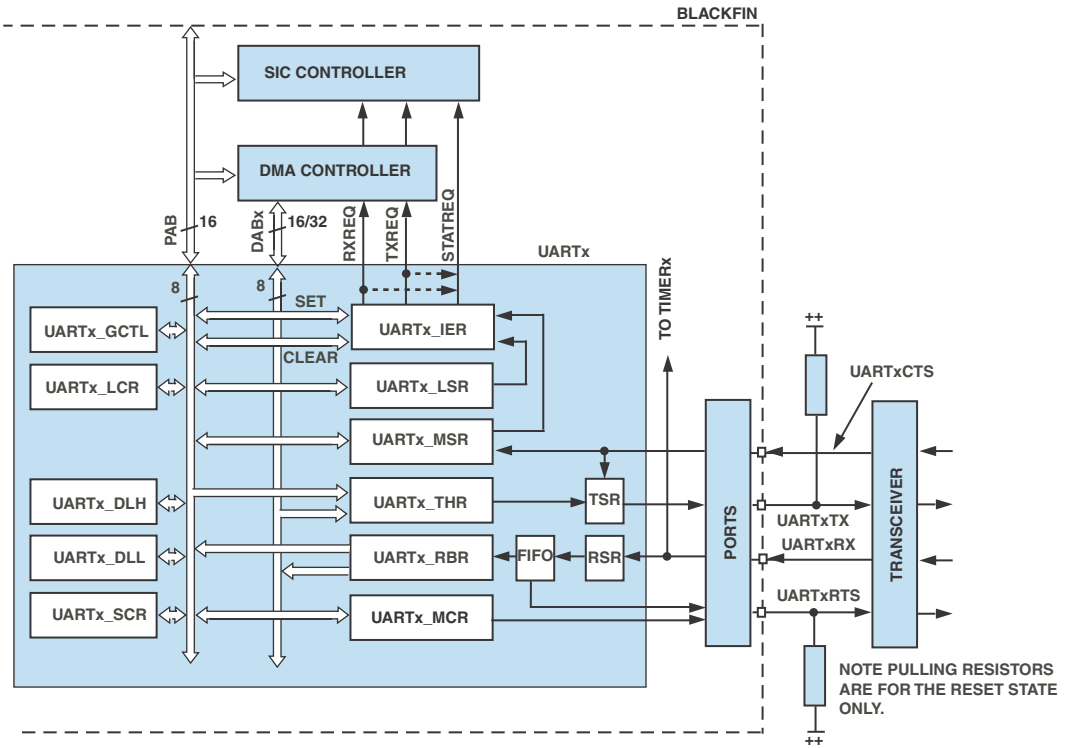


Figure 15-1. UART Block Diagram

External Interface

Each UART features an RX and a TX pin available through general-purpose ports. These two pins usually connect to an external transceiver device that meets the electrical requirements of full duplex (for example,

EIA-232, EIA-422, 4-wire EIA-485) or half duplex (for example, 2-wire EIA-485, LIN) standards. Additionally, each UART features a pair of UARTxCTS (clear to send, input) and UARTxRTS (request to send, output) signals for hardware flow control.

All UART signals are multiplexed and compete with other functions at pin level. Table 15-1 shows where the signal can be found and how they are enabled in the port control.

Table 15-1. UART Signals

Signal	Pin	Port Control	Autobaud Timer
UART0 TX	PF1 (or PG13)	PORTF_MUX[3:2] = b#01 PORTF_FER[1] = 1 (or PORTG_MUX[13:12] = b#00 PORTG_FER[13] = 1)	-
UART0 RX	PF0 (or PG12)	PORTF_MUX[1:0] = b#01 PORTF_FER[0] = 1 (or PORTG_MUX[13:12] = b#00 PORTG_FER[12] = 1)	Timer 6 (TMR6) (or Timer 2 (TACI2))
UART0 RTS	PG14	PORTG_MUX[15:14] = b#00 PORTG_FER[14] = 1	-
UART0 CTS	PG15	PORTG_MUX[15:14] = b#00 PORTG_FER[15] = 1	-
UART1 TX	PF6 (or PG3)	PORTF_MUX[7:6] = b#00 PORTF_FER[6] = 1 (or PORTG_MUX[7:6] = b#10 PORTG_FER[3] = 1)	-
UART1 RX	PF7 (or PG0)	PORTF_MUX[7:6] = b#00 PORTF_FER[7] = 1 (or PORTG_MUX[1:0] = b#10 PORTG_FER[0] = 1)	Timer 3 (TACI3) (or Timer 4 (TACI4))
UART1 RTS	PF8	PORTF_MUX[9:8] = b#00 PORTF_FER[8] = 1	-
UART1 CTS	PF9	PORTF_MUX[9:8] = b#00 PORTF_FER[9] = 1	-

Internal Interface

The UARTs are DMA-capable peripherals with support for separate TX and RX DMA master channels. They can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. For more information on DMA, see the *Direct Memory Access* chapter.

All UART registers are 8 bits wide. They connect to the PAB bus. The UARTx_RBR and UARTx_THR registers also connect to one of the DABx busses. While UART0 and UART1 connect to the DAB16 bus.

Each UART has three interrupt outputs. The transmit request and receive request outputs can function as DMA requests and connect to the DMA controller. Therefore, if the DMA is not enabled, the DMA controller simply forwards the request to the SIC controller. The status interrupt output connects directly to the SIC controller.

When no DMA channel is assigned, a UART has only one interrupt output. To modify, set the EGLSI bit in the UARTx_GCTL register to redirect transmit and receive requests to the status interrupt output.

Every UART's RX pin is also sensed by the alternative capture input (TACIx) of one of the general-purpose timers. Table 15-1 shows the assignment. In capture mode, the timers can be used to detect the bit rate of the received signal. See "Autobaud Detection" on page 15-20.

Description of Operation

The sections that follow describe the operation of the UART.

UART Transfer Protocol

UART communication follows an asynchronous serial protocol, consisting of individual data words. A word has 5 to 8 data bits.

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the line control register (UARTx_LCR). Data is always transmitted and received with the least significant bit (LSB) first.

Figure 15-2 shows a typical physical bitstream measured on one of the TX pins.

Aside from the standard UART functionality, the UART also supports serial data communication by way of infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

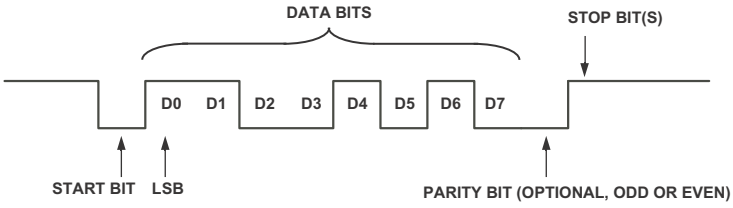


Figure 15-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

IrDA support is enabled by setting the IREN bit in the UARTx_GCTL register. The IrDA application requires external transceivers.

UART Transmit Operation

Receive and transmit paths operate completely independently except that the bit rate and the frame format are identical for both transfer directions.

Transmission is initiated by writes to the UARTx_THR register. If no former operation is pending, the data is immediately passed from the UARTx_THR register to the internal TSR register where it is shifted out at a bit rate characterized by the formula that follows with start, stop, and parity bits appended as defined by the UARTx_LCR register:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDB0)} \times Divisor}$$

The least significant bit (LSB) is always transmitted first. This is bit 0 of the value written to UARTx_THR.

Writes to the UARTx_THR register clear the THRE flag. Transfers of data from UARTx_THR to the transmit shift registers (TSR) set this status flag in UARTx_LSR again.

When enabled by the ETBEI bit in the UARTx_IER register, the THRE flag requests an interrupt on the dedicated TXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the TXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the SIC interrupt controller. If no DMA channel is assigned to the UART, the EGLSI bit in the UARTx_GCTL register can redirect the receive and transmit interrupts to the UART status interrupt alternatively.

The UARTx_THR register and the internal TSR register can be seen as a two-stage transmit buffer. When data is pending in either one of these registers, the TEMT flag is low. As soon as all data has left the TSR register, the

TEMT bit goes high again and indicates that all pending transmit operation has finished. At that time it is safe to disable the UCEN bit or to three-state off-chip line drivers. An interrupt can be generated by that time either through the status interrupt channel when the ETFI bit is set, or through the DMA controller when enabled by the EDTPTI bit.

UART Receive Operation

The receive operation uses the same data format as the transmit configuration, except that one valid stop bit is always sufficient, that is, the STB bit has no impact to the receiver.

The UART receiver is sensing the falling edges of the RX input. When an edge is detected, the receiver starts sampling the RX input according to the bit rate and the EDBO bit settings. The start bit is sampled close to its midpoint. If sampled low, a valid start condition is assumed. Otherwise, the detected falling edge is discarded.

After detection of the start bit, the received word is shifted into the internal shift register (RSR) at a bit rate characterized by the following formula:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDB0)} \times Divisor}$$

After the corresponding stop bit is received, the content of the RSR register is transferred through the 4-deep receive FIFO to the UARTx_RBR register, shown in Figure 15-13. Finally, the data ready (DR) bit and the status flags are updated in the UARTx_LSR register, to signal data reception, parity, and also error conditions, if required.

The receive FIFOs and the UARTx_RBR registers can be seen as a five-stage receive buffer. If the stop bit of the 6th word is received before software reads the UARTx_RBR register, an overrun error is reported. The overrun case protects data in the UARTx_RBR and receive FIFO from being overwritten by further data until the OE bit is cleared by software. The data in the

RSR register, however, is immediately destroyed as soon as the overrun occurs.

If enabled by the ERBFI bit in the UARTx_IER register, the DR flag requests an interrupt on the dedicated RXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the RXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the SIC interrupt controller. If no DMA channel is assigned to the UART, the EGLSI bit in the UARTx_GCTL register can redirect the receive and transmit interrupts to the UART status interrupt alternatively.

The state of the five-deep receiver buffer (including UARTx_RBR) can be monitored by the receiver FIFO count status (RFCS) bit in the UARTx_MSR register. The buffer's behavior is controlled by the receive FIFO interrupt threshold (RFIT) bit in the UARTx_MCR register. If RFIT is zero, the RFCS bit is set when the receive buffer holds two or more words. If RFIT is set, the RFCS bit is set when the receive buffer holds four or more words. The RFCS bit is cleared by hardware when core or DMA read the UARTx_RBR register and when the buffer is flushed below the level of two (RFIT=0) or four (RFIT=4). If the associated interrupt bit ERFCI is enabled, status interrupt is reported when the RFCS bit is set.

If errors are detected during reception, an interrupt can be requested to a the status interrupt output. This status interrupt request goes directly to the SIC interrupt controller. Status interrupt requests are enabled by the ELSI bit in the UARTx_IER_SET register. The following error situations are detected. Every error has an indicating bit in the UARTx_LSR register.

- Overrun error (OE bit)
- Parity error (PE bit)
- Framing error/Invalid stop bit (FE bit)
- Break indicator (BI bit)

The sampling clock is 16 times faster than the bit clock. The receiver over samples every bit 16 times and does a majority decision based on the mid three samples. This improves immunity against noise and hazards on the line. Spurious pulses of less than two times the sampling clock period are disregarded.

Normally, every incoming bit is sampled at exactly the 7th, 8th and 9th sample clock. If, however, the EDBO bit is set to 1 to achieve better bit rate granularity and accuracy as required at high operation speeds, the bits are one roughly sampled at 7/16th, 8/16th and 9/16th of their period. Hardware design should ensure that the incoming signal is stable between 6/16th and 10/16th of the nominal bit period.

Reception is started when a falling edge is detected on the UARTxRX input pin. The receiver attempts to see a start bit. The data is shifted into the internal RSR register. After the 9th sample of the first stop bit is processed, the received data is copied to the 5-stage receive buffer and the RSR recovers for further data.

The receiver samples data bits close to their midpoint. Because the receiver clock is usually asynchronous to the transmitter's data rate, the sampling point may drift relative to the center of the data bits. The sampling point is synchronized again with each start bit, so the error accumulates only over the length of a single word.

Hardware Flow Control

To prevent the UART transmitter from sending data while the receiving counterpart is not ready, a RTS/CTS hardware flow control mechanism is supported. The UARTxRTS (request to send) signal is an output that con-

nects to the communication's partner UARTxCTS (clear to send) input. If data transfer is bidirectional, the handshake is as shown in Figure 15-3.

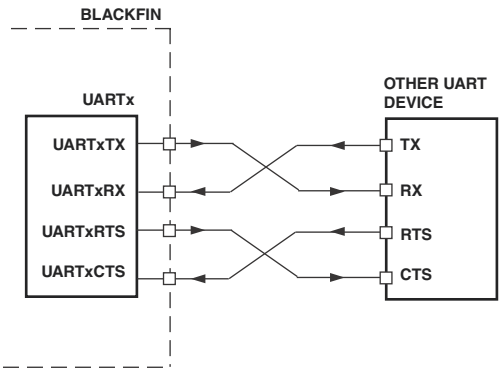


Figure 15-3. UART Hardware Flow

Regardless of whether working in DMA or non-DMA mode, the receiver can deassert the UARTxRTS signal to indicate that its receive buffer is getting full. Further data may cause an overrun error. Consequently, the transmitter pauses transmission when the UARTxCTS input is in deasserted state. On ADSP-BF50x processors, UART1 features a pair of RTS/CTS pins each. Automatic hardware flow control can be enabled individually for receiver and transmitter by the UARTx_MCR register's ARTS and ACTS bits.

The signals are usually active low, that is, transmission is halted when the pin state is high. The polarity of the UARTxCTS and UARTxRTS pins can be inverted by setting the FCPOL bit in the UARTx_MCR register. If ACTS is enabled, the UARTxCTS bit in the UARTx_MSR register holds the complement value (FCPOL=0) or the value (FCPOL=1) of the UARTxCTS input pin. In either case the UARTxCTS bit reads 1 when the external device is ready to receive data. The delta CTS (DCTS) bit is a sticky version of the UARTxCTS bit that is set high when the UARTxCTS bit transitions from 0 to 1. It can request a status interrupt and is cleared by software with a W1C opera-

tion. If the TX handshaking protocol is enabled (bit ACTS=1), the UART hardware pauses transmission if the UARTxCTS bit is zero. If the UARTxCTS input is deasserted, the transmitter still completes transmission of the data work currently held in the internal TSRx register, but does not continue with the data in UARTx_THR. If the UARTxCTS is asserted again, the transmitter resumes and loads the content of UARTx_THR into TSRx.

If the RX handshaking protocol is enabled (bit ARTS=1 in the UARTx_MCR register), the UARTxRTS output pin is toggled automatically by the receiver's hardware. The pin's assertion and deassertion timing is controlled by the receive FIFO RTS threshold (RFRT) bit in the UARTx_MCR register. If RFRT is cleared, the UARTxRTS pin is deasserted when the receive buffer already holds two words and a third start bit is detected. The UARTxRTS pin is asserted again when the buffer does not contain any more data than the word in the UARTx_RBR register. If RFRT is set, the UARTxRTS pin is deasserted when the receive buffer already holds four words and a fifth start bit is detected. The UARTxRTS is re-asserted when the buffer contains less than four words. Hardware guarantees minimal UARTxRTS deassertion pulse width of at least the number of data bits as defined by the WLS bit field in the UARTx_LCR register.

If ACTS=0, the TX handshaking protocol is disabled, and the UART transmits data as long as there is data to transmit, regardless of the value of UARTxCTS. With ACTS=0 software can pause on-going transmission by setting the XOFF bit in the UARTx_MCR register.

If ARTS=0, the UARTxRTS pin is not generated automatically by hardware. The UARTxRTS output can then still be manually controlled by the MRTS bit in the UARTx_MCR register.

On reset, when the UART is not yet enabled and the port multiplexing has not been programmed, the UARTxRTS pin is not driven. Some applications may require the UARTxRTS signal to be pulled to either state by a resistor during reset.

IrDA Transmit Operation

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted if the TPOLC bit is cleared, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3/16 clock periods in a 16-cycle UART clock period. The pulse is centered around the middle of the bit time, as shown in Figure 15-4. The final IrDA pulse is fed to the off-chip infrared driver.

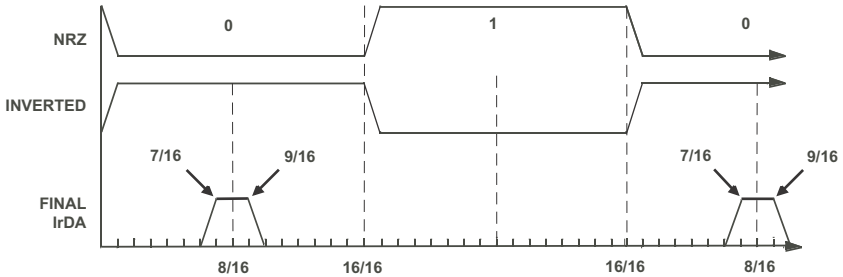


Figure 15-4. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 15-2 on page 15-19, the error terms associated with the bit rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receive Operation

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do

this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16x bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the IRPOL bit. Figure 15-5 gives examples of each polarity type.

- IRPOL = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- IRPOL = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

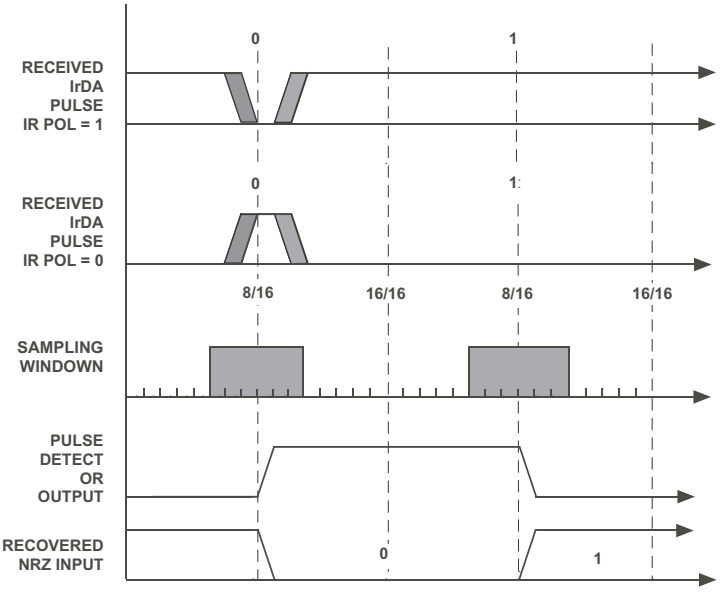


Figure 15-5. IrDA Receiver Pulse Detection

In the IrDA mode the EDBO bit is ignored. The sample frequency is always exactly 16 times the bit rate.

Interrupt Processing

Each UART module has three interrupt outputs. One is dedicated for transmission, one for reception, and the third is used to report status events. As shown in Figure 15-1 on page 15-3, the transmit and receive

requests are routed through the DMA controller. The status request goes directly to the SIC controller.

If the associated DMA channel is enabled, the request functions as a DMA request. If the DMA channel is disabled, it simply forwards the request to the SIC interrupt controller. Note that a DMA channel must be associated with the UART module to enable TX and RX interrupts. Otherwise, the transmit and receive requests cannot be forwarded. Refer to the description of the peripheral map registers in the "Direct Memory Access" chapter in the *ADSP-BF50x Blackfin Processor Hardware Reference (Volume 1 of 2)*.

On ADSP-BF50x processors not all UARTs have a DMA channel assigned by default. Even if disabled, a DMA channel is still required to forward the DMA requests to the SIC controller as interrupt requests (see Figure 15-1 on page 15-3). Also, if no DMA channel is assigned, the UART loses its normal receive and transmit interrupt functionality.

To operate in interrupt mode without assigned DMA channels, set the EGLSI bit in the UARTx_GCTL register. This setup redirects receive and transmit requests to the status interrupt output. The status interrupt goes directly to the SIC controller without being routed through the DMA controller.

Transmit interrupts are enabled by the ETBEI bit in the UARTx_IER_SET register. If set, the transmit request is asserted along with the THRE bit in the UART_LSR, indicating that the TX buffer is ready for new data.

Note that the THRE bit resets to 1. When the ETBEI bit is set in the UARTx_IER_SET register, the UART module immediately issues an interrupt or DMA request. This way, no special handling of the first character is required when transmission of a string is initiated. Simply set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UARTx_THR register in the normal manner. Accordingly, the ETBEI bit can be cleared in the UARTx_IER_CLEAR register

if the string transmission has completed. For more information, see "DMA Mode" on page 15-23.

The THRE bit is cleared by hardware when new data is written to the UARTx_THR register. These writes also clear the TX interrupt request. However, they also initiate further transmission. If software doesn't want to continue transmission, the TX request can alternatively be cleared by clearing the ETBEI bit in the UARTx_IER_CLEAR register.

Receive interrupts are enabled by the ERBFI bit in the UARTx_IER_SET register. If set, the receive request is asserted along with the DR bit in the UART_LSR register, indicating that new data is available in the UARTx_RBR register. When software reads the UARTx_RBR, hardware clears the DR bit again which in turn clears the receive interrupt request.

The UART status interrupt channels are used for multiple purposes:

- Line Status Interrupts
- Flow Control Interrupts
- Receive FIFO Threshold Interrupts
- Transmission Finished Interrupt

Line status interrupts are enabled by the ELSI bit in the UARTx_IER_SET register. If set, the status interrupt request is asserted with any of the BI, FE, PE or OE receive errors bits in the UART_LSR register. Refer to "UARTx_LSR Registers" on page 15-33 for details. The error bits in the UARTx_LSR register are cleared by W1C operation. Once all error conditions are cleared the interrupt request deasserts.

The receive FIFO count interrupt is enabled by the ERFCI bit in the UARTx_IER_SET register. If set, a status interrupt is generated when the RFCS is active. The RFCS bit indicates a receive buffer threshold level. If the RFIT bit in the UARTx_MCR register is cleared, software can safely read two words out of the UARTx_RBR register by the time the RFCS interrupt occurs. If the RFIT bit is set, software can safely read four words. The interrupt

and the RFCS bit clear when the UARTx_RBR is read sufficient times, so that the receive buffer drains below the threshold of two (RFIT=0) or four (RFIT=1). Because in DMA mode a status service routine may not be permitted to read UARTx_RBR, this interrupt is only recommended in non-DMA mode. In DMA mode, use this functionality for error recovery only.

The UARTxCTS interrupts are enabled by the EDSSI bit in the UARTx_IER_SET register. If active, a status interrupt is generated when the sticky SCTS bit in the UARTx_MSR register is set, indicating that the transmitter's UARTx-CTS input been re-asserted. A W1C operation to the SCTS bit clears the interrupt request.

A transmission finished interrupt is enabled by the ETFI bit in the UARTx_IER_SET register. If active, a status interrupt request is asserted when the TFI bit in the UARTx_LSR register is set. TFI is the sticky version of the TEMT bit, indicating that a byte that started transmission has completely finished. The interrupt request is cleared by a W1C operation to the TFI bit.

Bit Rate Generation

The UART clock is enabled by the UCEN bit in the UARTx_GCTL register.

The sample clock is characterized by the system clock (SCLK) and the 16-bit divisor. The divisor is split into the 8-bit UARTx_DLL and the UARTx_DLH registers. These registers form a 16-bit divisor.

By default every serial bit is over sampled 16 times. The bit clock is 1/16th of the sample clock. If not in IrDA mode the bit clock can equal the sample clock if the EDBO bit in the UARTx_GCTL register is set, so that the following applies:

$$BIT\ RATE = \frac{SCLK}{16^{(1-EDBO)} \times Divisor}$$

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

Table 15-2 provides example divide factors required to support most standard baud rates.

Table 15-2. UART Bit Rate Examples With 133 MHz SCLK

Bit Rate	Dfactor = 16			Dfactor = 1		
	DL	Actual	% Error	DL	Actual	% Error
2400	3464	2399.68	0.013	55417	2399.99	0.001
4800	1732	4799.36	0.013	27708	4800.06	0.001
9600	866	9598.73	0.013	13854	9600.12	0.001
19200	433	19197.46	0.013	6927	19200.23	0.001
38400	216	38483.80	0.218	3464	38394.92	0.013
57600	144	57725.69	0.218	2309	57600.69	0.001
115200	72	115451.39	0.218	1155	115151.52	0.042
921600	9	923611.11	0.218	144	923611.11	0.218
1500000	6	1385416.67	7.639	89	1494382.02	0.375
3000000	3	2770833.33	7.639	44	3022727.27	0.758
6250000	1	8312500.00	33.000	21	6333333.33	1.333

Careful selection of SCLK frequencies, that is, even multiples of desired bit rates, can result in lower error percentages.

Setting the bit clock equal to the sample clock (EDBO=1) improves bit rate granularity and enables the Blackfin bit clock to more closely match the bit rate of the communication partner. There is, however, a disadvantage—the power dissipation is higher. Also the sample points may not be that accurate. It is recommended to use EDBO=1 mode only when bit rate accuracy is not acceptable in EDBO=0 mode.

The EDBO=1 mode is not intended to increase operation speed beyond the electrical limitations of the asynchronous UART transfer protocol.

Autobaud Detection

At the chip level, the UART RX pins are routed to the alternate capture inputs (TACIx) of the general purpose timers. When working in WDTH_CAP mode these timers can be used to automatically detect the bit rate applied to the UARTxRX pin by an external device. For more information, see "General-Purpose Timers" on page 10-1.

The capture capabilities of the timers are often used to supervise the bit rate at runtime. If the Blackfin UART was talking to any device supplied by a weak clock oscillator that drifts over time, the Blackfin can re-adjust its UART bit rate dynamically as required.

Often, autobaud detection is used for initial bit rate negotiations. There, the Blackfin processor is most likely a slave device waiting for the host to send a predefined autobaud character as discussed below. This is exactly the scenario used for UART booting. In this scenario, it is recommended that the UART clock enable bit UCEN is not enabled while autobaud detection is performed to prevent the UART from starting reception with incorrect bit rate matching. Alternatively, the UART can be disconnected from the UARTxRX pin by setting the LOOP_ENA bit.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with the UART operation—all derived from SCLK—the pulse widths can be used to calculate the bit rate divider for the UART by using the following formula:

$$DIVISOR = \frac{\dot{TIMERx\_WIDTH}}{16^{(1-EDB0)} \times Number\ of\ captured\ UART\ bits}$$

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more

bits. Traditionally, a NULL character (ASCII 0x00) was used in autobaud detection, as shown in Figure 15-6.

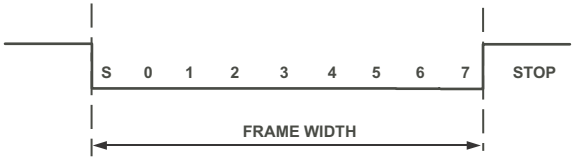


Figure 15-6. Autobaud Detection Character 0x00

Because the example frame in Figure 15-6 encloses 8 data bits and 1 start bit, apply the following formula:

$$DIVISOR = \frac{TIMERx\_WIDTH}{16^{(1-EDB0)} \times 9}$$

Real UARTxRX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character "@" (0x40) as the autobaud detection character and measure the period between two subsequent falling edges. As shown in Figure 15-7, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses 8 bits, apply the following:

- Divisor = `TIMERx_PERIOD` >> 7 if `EDB0` = 0
- Divisor = `TIMERx_PERIOD` >> 3 if `EDB0` = 1

An example is provided in Listing 15-2 on page 15-48.

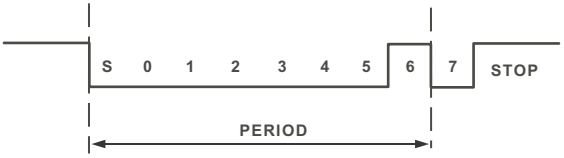


Figure 15-7. Autobaud Detection Character 0x40

Programming Model

The following sections describe a programming model for the UARTs.

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into UARTx_THR. Received data can be read from UARTx_RBR. The processor must write and read a limited number of characters at a time.

To prevent any loss of data and misalignments of the serial data stream, the UARTx_LSR register provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UARTx_THR is ready for new data and cleared when the processor loads new data into UARTx_THR. Writing UARTx_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UARTx_RBR. This flag is cleared automatically when the processor reads from UARTx_RBR. Reading UARTx_RBR when it is not full returns the previously received value. When UARTx_RBR is not read in time, an overrun condition protects the already received data from being overwritten by new data until the OE bit is cleared by software. Only the content of the RSR register can be overwritten in the overrun case.

The TEMT bit can be interrogated to see whether any transmission is ongoing. The TEMT bit's sticky counterpart TFI tells whether the transmit buffer has drained and can trigger a status interrupt, if required.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Since read operations from UARTx_LSR registers have no side effects, different software threads can interrogate these registers without mutual impacts. Polling the SIC_ISRx register without enabling the interrupts by SIC_MASKx is an alternate method of operation to consider. Software can write up to two words into the UARTx_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines (ISRs). Separate interrupt lines are provided for UART TX, UARTxRX, and UART status. The independent interrupts can be enabled individually by the UARTx_IER_SET and UARTx_IER_CLEAR register pair. The UCEN bit must be set to enable UART transmit interrupts.

The ISRs can evaluate the status bits in the UARTx_LSR and UARTx_MSR registers to determine the signalling interrupt source. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 15-15 on page 15-41.

To reduce interrupt frequency on the receive side in non-DMA mode, the ERFCI status interrupt may be used as an alternative to the regular ERBFI receive interrupt. Hardware ensure that at least two (if RFIT=0) or four (if RFIT=1) words are available in the receive buffer by the time the interrupt is requested.

DMA Mode

In this mode, separate receive (UARTxRX) and transmit (UARTxTX) DMA channels move data between the UART and memory. The software

does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through autobuffer mode.

DMA channels provide a 4-deep FIFO, resulting in total buffer capabilities of 6 words at the transmit and 9 words at the receive side receive sides. In DMA mode, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities. For more information, see "Direct Memory Access" on page 7-1.

DMA interrupt routines must explicitly write 1s to the corresponding DMAx_IRQ_STATUS registers to clear the latched request of the pending interrupt.

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UARTx_IER_SET register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. The UART's status interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

For transmit DMA, it is recommended to set the SYNC bit in the DMAx_CONFIG register. With this bit set, the interrupt generation is delayed until the entire DMA FIFO is drained to the UART module. The UART TX DMA interrupt service routine is allowed to disable the DMA or to clear the ETBEI control bit only when the SYNC bit is set, otherwise up to four data bytes might be lost.

When the ETBEI bit is set in the UARTx_IER_SET register, an initial transmit DMA request is issued immediately. It is common practice to clear the ETBEI bit by the DMA's service routine.

In DMA transmit mode, the ETBEI bit enables the peripheral request to the DMA FIFO. The strobe on the memory side is still enabled by the DMAEN bit. If the DMA count is less than the DMA FIFO depth, which is

4, then the DMA interrupt might be requested already before the ETBEI bit is set. If this is not wanted, set the SYNC bit in the DMAx_CONFIG register.

Regardless of the SYNC setting, the DMA stream has not left the UART transmitter completely at the time the interrupt is generated. Transmission may abort in the middle of the stream, causing data loss, if the UART clock was disabled without additional synchronization with the TEMT bit.

The ADSP-BF50x UART implementation provides new functionality to avoid expensive polling of the TEMT bit. The EDTPTI bit in the UARTx_IER_SET register enables the TEMT bit to trigger a DMA interrupt. To delay the DMA completion interrupt until the last data word of a STOP DMA has left the UART, keep the DMA's DI_EN bit cleared and set the EDTPTI bit instead. Then, the normal DMA completion interrupt is suppressed. Later, the TEMT event triggers a DMA interrupt after the DMA's last word has left the UART transmit buffers. If DI_EN and EDTPTI are set, when finishing STOP mode, the DMA requests two interrupts.

The UART's DMA supports 8-bit and 16-bit operation, but not 32-bit operation. Sign extension is not supported.

Mixing Modes

Especially on the transmit side, switching from DMA mode to non-DMA operation on the fly requires some thought. By default, the interrupt timing of the DMA is synchronized with the memory side of the DMA FIFOs. Normally, the UARTxTX DMA completion interrupt is generated after the last byte is copied from the memory into the DMA FIFO. The UARTxTX DMA interrupt service routine is not yet permitted to disable the DMA enable bit DMAEN. The interrupt is requested by the time the DMA_DONE bit is set. The DMA_RUN bit, however, remains set until the data has completely left the UARTxTX DMA FIFO.

Therefore, when planning to switch from DMA to non-DMA of operation, always set the SYNC bit in the DMAx_CONFIG word of the last descriptor or work unit before handing over control to non-DMA mode. Then, after

the interrupt occurs, software can write new data into the UARTx_THR register as soon as the THRE bit permits. If the SYNC bit cannot be set, software can poll the DMA_RUN bit instead. Using the EDTPTI bit can avoid expensive status bit polling, alternatively.

When switching from non-DMA to DMA operation, take care that the very first DMA request is issued properly. If the DMA is enabled while the UART is still transmitting, no precaution is required. If, however, the DMA is enabled after the TEMT bit became high, the ETBEI bit should be pulsed to initiate DMA transmission.

UART Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These memory-mapped registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled. Table 15-3 provides an overview of the UART registers.

Unlike on ADSP-BF52x processors, register addresses are not shared on ADSP-BF50x processors. Each register has its own MMR address. Consequently, the DLAB bit is not present on ADSP-BF50x processors' UARTx_LSR registers. Software must use 16-bit word load/store instructions to access these registers.

Furthermore, the interrupt processing differs from ADSP-BF52x processors. Error bits in status registers do not clear on register reads implicitly, rather they are cleared by write-1-to-clear (W1C) operations. The UARTx_IIR register is not present at all. The interrupt enable register has separate set and clear ports, so that separate receive, transmit, and status interrupt service routines can enable or set masks individually.

Transmit and receive channels are both buffered. The UARTx_THR registers buffer the transmit shift registers (TSR). The UARTx_RBR registers and an

additional 4-stage receive FIFO buffer the receive shift register (RSR). The shift registers are not directly accessible by software.

Table 15-3. ADSP-BF50x versus ADSP-BF52x UART Register

Name	ADSP-BF50x Address Offset	ADSP-BF52x Address Offset	Register Name
UARTx_DLL	0x00	0x00, DLAB=1	UART divisor latch low byte registers on page 15-43
UARTx_DLH	0x04	0x00, DLAB=1	UART divisor latch high byte registers on page 15-43
UARTx_GCTL	0x08	0x24	UART global control register on page 15-45
UARTx_LCR	0x0C	0x0C	UART line control registers on page 15-28
UARTx_MCR	0x10	0x10	UART modem control registers on page 15-31
UARTx_LSR	0x14	0x14	UART line status registers on page 15-33
UARTx_MSR	0x18	N/A	UART modem status registers on page 15-36
UARTx_SCR	0x1C	0x1C	UART scratch registers on page 15-44
UARTx_IER_SET	0x20	N/A	UART interrupt enable set registers on page 15-39
UARTx_IER_CLEAR	0x24	N/A	UART interrupt enable clear registers on page 15-39
UARTx_IER	N/A	0x04, DLAB=0	Interrupt Enable R/W register on page 15-28
UARTx_THR	0x28	0x00, DLAB=0	UART transmit hold registers on page 15-37
UARTx_RBR	0x2C	0x00, DLAB=0	UART receive buffer registers on page 15-38
UARTx_IIR	N/A	0x08	Interrupt Enable register on page 15-28

UARTx_LCR Registers

The line control (UARTx_LCR) registers, shown in Figure 15-8, control the format of received and transmitted character frames.

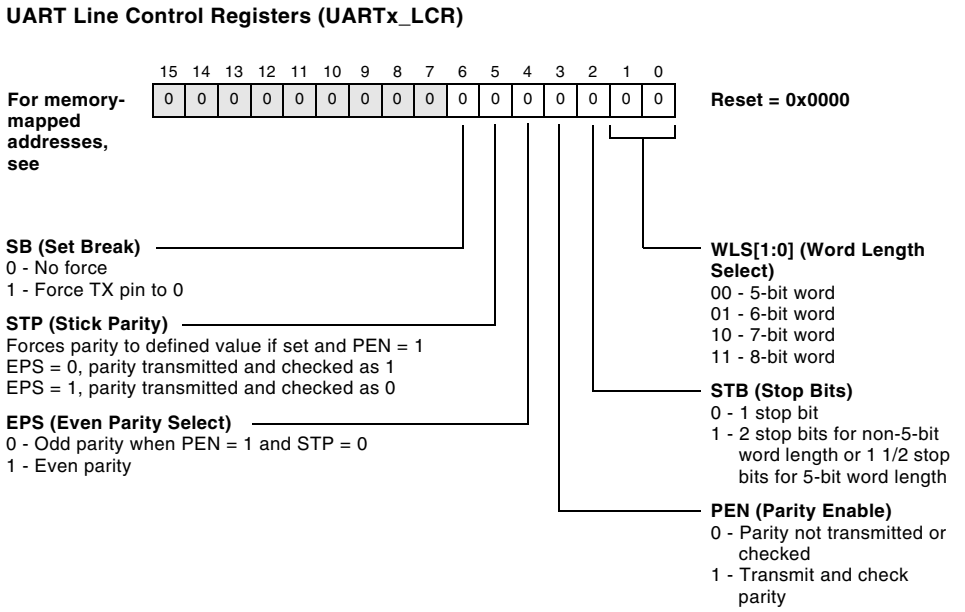


Figure 15-8. UART Line Control Registers

Table 15-4. UART Line Control Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_LCR	0xFFC0 040C
UART1_LCR	0xFFC0 200C

The 2-bit WLS field determines whether the transmitted and received UART word consists of 5, 6, 7 or 8 data bits.

The STB bit controls how many stop bits are appended to transmitted data. When STB=0, one stop bit is transmitted. If WLS is non zero, STB=1 instructs the transmitter to add one additional stop bit, two stop bits in total. If WLS=0 and 5-bit operation is chosen, STB=1 forces the transmitter to append one additional half bit, 1 1/2 stop bits in total. Note that this bit does not impact data reception—the receiver is always satisfied with one stop bit.

The PEN bit inserts one additional bit between the most significant data bit and the first stop bit. The polarity of this so-called parity bit depends on data and the STP and EPS control bits. Both transmitter and receiver calculate the parity value. The receiver compares the received parity bit with the expected value and issues a parity error if they don't match. If PEN is cleared, the STP and the EPS bits are ignored.

The STP bit controls whether the parity is generated by hardware based on the data bits or whether it is set to a fixed value. If STP=0 the hardware calculates the parity bit value based on the data bits. Then, the EPS bit determines whether odd or even parity mode is chosen. If EPS=0, odd parity is used. That means that the total count of logical-1 data bits including the parity bit must be an odd value. Even parity is chosen by STP=0 and EPS=1. Then, the count of logical-1 bits must be a even value. If the STP bit is set, then hardware parity calculation is disabled. In this case, the sent and received parity equals the inverted EPS bit. The example in Table 15-5 summarizes polarity behavior assuming 8-bit data words (WLS=3).

Table 15-5. UART Parity

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
0	x	x	x	x	None
1	0	0	0x60	0000 0110	1
1	0	0	0x57	1110 1010	0
1	0	1	0x60	0000 0110	0

Table 15-5. UART Parity (Cont'd)

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
1	0	1	0x57	1110 1010	1
1	1	0	x	x	1
1	1	0	x	x	1
1	1	1	x	x	0
1	1	1	x	x	0

If set, the SB bit forces the UARTxTX pin to low asynchronously, regardless of whether or not data is currently transmitted. It functions even when the UART clock is disabled. Since the UARTxTX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

UARTx_MCR Registers

The modem control (UARTx_MCR) registers control the UART port, as shown in Figure 15-9. Partial modem functionality is supported to allow for hardware flow control and loopback mode.

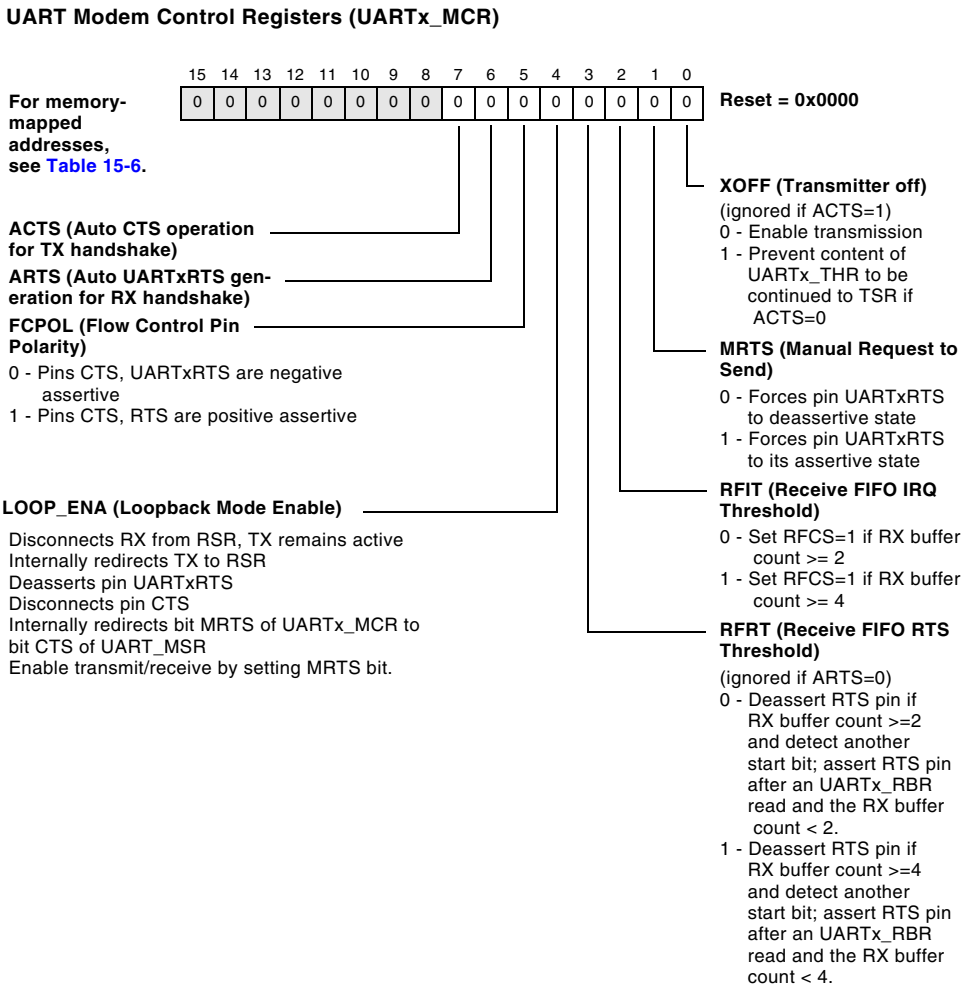


Figure 15-9. UART Modem Control Registers

Table 15-6. UART Modem Control Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_MCR	0xFFC0 0410
UART1_MCR	0xFFC0 2010

The receive FIFO interrupt threshold (RFIT) bit controls the timing of the RFCS status bit. If RFIT=0, the receive threshold is two. If RFIT=1, the threshold is four words in the receive buffer.

The manual request to send (MRTS) bit controls the state of the UARTxRTS output pin only if ARTS=0. A value of MRTS=0 forces the UARTxRTS pin to its deassertive state, signaling to the external device that the UART is not ready to receive. A value of MRTS=1 forces the UARTxRTS pin to its assertive state, signaling to the external device that the UART is ready to receive.

The automatic RTS (ARTS) bit enables the receive buffer to control the RTS output depending on the threshold programmed by the RFTR bit. If RFRT=0, the RTS signal is deasserted when already two words are held by the receive buffer and a third start bit is detected. It is re-asserted if the buffer contains less than two words. If RFRT=1, the RTS signal is deasserted when already four words are held by the receive buffer and a fifth start bit is detected. The RTS signal is re-asserted if the buffer contains less than four words.

Similarly, the automatic CTS (ACTS) bit must be set to enable the CTS input pin for UARTxTX handshaking. If enabled, the CTS status bit in the UARTx_MSR register holds the value (if FCPOL=1) or complement value (if FCPOL=0) of the CTS input pin. The CTS status bit can be used to determine if the external device is ready to receive data (CTS=1) or if it is busy (CTS=0). If ACTS=0, the UARTxTX handshaking protocol is disabled, and the UARTxTX line transmits data whenever there is data to send, regardless of the value of CTS. The transmitter off (XOFF) bit can be used to pause an on-going transmission by software when ACTS=0. Similarly to automatic CTS mode, the XOFF bit prevents the data in the UARTx_THR register

from being continued to the TSR shift register. When ACTS=1, the XOFF bit is ignored. When ACTS=0, the state of the CTS input signal is ignored.

The polarities of the UARTxCTS and UARTxRTS pins can be programmed using the FCPOL bit. If FCPOL=0, the pins are negative asserted. If FCPOL=1, the pins are positive asserted.

Loopback mode (LOOP_ENA=1) disconnects the receiver's input from the UARTxRX pin, and internally redirects the transmit output to the receiver. The UARTxTX pin remains active and continues to transmit data externally as well. Loopback mode also forces the UARTxRTS pin to its deassertive state, disconnects the UARTxCTS bit from the UARTxCTS input pin, and directly connects bit MRTS to bit UARTxCTS of the modem status register (UARTx_MSR). In loopback mode, writing a 1 to the MRTS bit sets bit UARTxCTS, DCTS and enable the UART's transmitter. Writing a 0 to the MRTS bit clears bit UARTxCTS and disable the UART's transmitter.

UARTx_LSR Registers

The line status (UARTx_LSR) registers contain UART status information as shown in Figure 15-10. Unlike the industrial standard, the ADSP-BF50x processor's UARTx_LSR register is not read only. Writes to this register can perform write-one-to-clear (W1C) operations on most status bits. Reading this register has no side effects.

The DR (data ready) bit indicates that data is available in the receiver and can be read from the UARTx_RBR register. The bit is set by hardware when the receiver detects the first valid stop bit. It is cleared by hardware when the UARTx_RBR register is read.

The OE (overrun error) bit indicates that further data is received while the internal receive buffer was full. It is set when sampling the stop bit of the 6th data word. To avoid overruns, read the UARTx_RBR register in time. In DMA receive mode overruns are very unlikely to happen ever. Once an overrun occurs, the UARTx_RBR and receive FIFO are protected from being overwritten by new data until the OE bit is cleared by software. The

content of receive shift register RSR, however, is lost as soon as the overrun occurs. The OE bit is sticky and can be cleared by W1C operations.

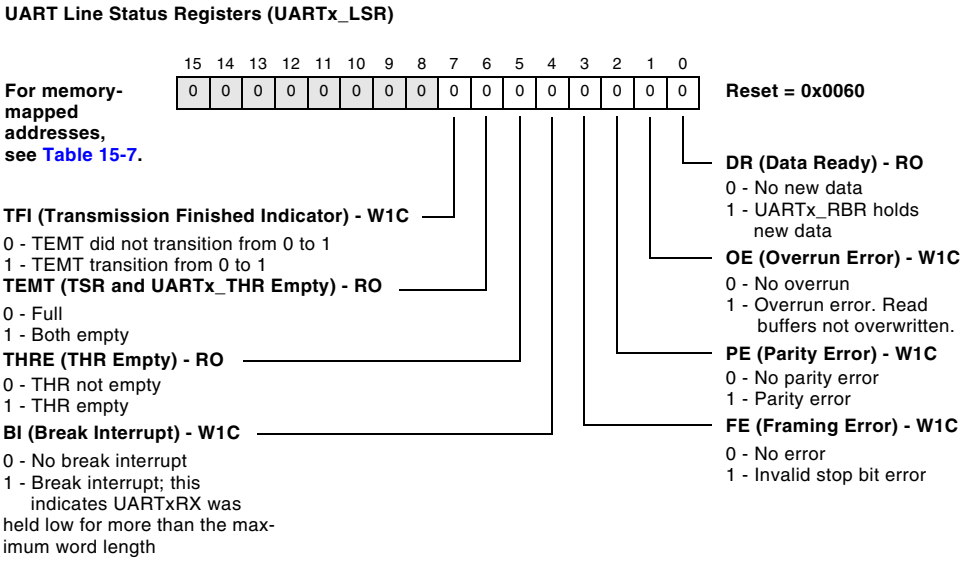


Figure 15-10. UART Line Status Registers

Table 15-7. UART Line Status Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_LSR	0xFFC0 0414
UART1_LSR	0xFFC0 2014

The PE (parity error) bit indicates that the received parity bit does not match the expected value. The PE bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations. Invalid parity bits can be simulated by setting the FPE bit in the UARTx_GCTL register.

The FE (framing error) bit indicates that the first stop bit is sampled. The FE bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations. Invalid stop bits can be simulated by setting the FFE bit in the UARTx_GCTL register.

The BI (break indicator) bit indicates that the first stop bit is sampled low and the entire data word, including parity bit, consists of low bits only. The BI bit is updated simultaneously with the DR bit, that is, by the time the first stop bit is received or when data is loaded from the receive FIFO to the UARTx_RBR register. The bit is sticky and can be cleared by W1C operations.

The THRE (transmit hold register empty) bit indicates that the UART transmit channel is ready for new data and software can write to UARTx_THR. Writes to UARTx_THR clear the THRE bit. It is set again when data is passed from UARTx_THR to the internal TSR register.

The TEMT (transmitter empty) bit indicates that both the UARTx_THR register and the internal TSR register are empty. In this case the program is permitted to write to the UARTx_THR register twice without losing data. The TEMT bit can also be used as indicator that pending UART transmission is completed. At that time it is safe to disable the UCEN bit or to three-state the off-chip line driver.

The TFI (transmission finished indicator) bit is a sticky version of the TEMT bit. While TEMT is automatically cleared by hardware when new data is written to the UARTx_THR register, the sticky TFI bit remains set until it is cleared by software (W1C). The TFI bit enables more flexible transmit interrupt timing.

UARTx_MSR Registers

The modem status (UARTx_MSR) registers, shown in Figure 15-11, contains current states of the UART's external UARTxCTS pin and current status of the UART's internal receive buffers.

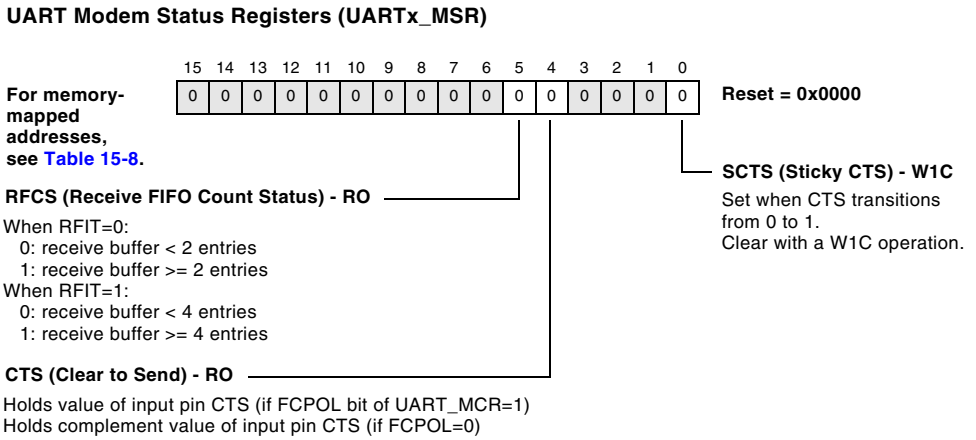


Figure 15-11. UART Modem Status Registers

Table 15-8. UART Modem Status Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_MSR	0xFFC0 0418
UART1_MSR	0xFFC0 2018

The UARTxCTS bit holds the value (if FCPOL = 1) or the complement value (if FCPOL = 0) of the UARTxCTS input pin. The ACTS bit in the UARTx_MCR register must be set to enable this feature. The core can read the value of UARTxCTS to determine if the external device is ready to receive (UARTxCTS = 1) or if it is busy (UARTxCTS = 0). If ACTS = 0, the UARTxTX handshaking protocol is disabled, and the UART transmits data as long as there is data to transmit, regardless of the value of UARTxCTS. When ACTS=0, the software can pause transmission temporarily by setting the XOFF bit.

The SCTS bit is a sticky bit that is set high when UARTxCTS transitions from 0 to 1, and is cleared by software with a W1C operation. The SCTS bit can trigger a line status interrupt if enabled by the EDSSI bit in the UARTx_IER_SET register.

The receiver FIFO count status (RFCS) bit is set when the receive buffer holds more or equal entries than a certain threshold. The threshold is controlled by the RFIT bit in the UARTx_MCR register. If RFIT=0, the threshold is two entries. If RFIT=1, the threshold is four entries. The RFCS bit cleared when the UARTx_RBR register is read sufficient times until the buffer is drained below the threshold. The RFCS bit can trigger a status interrupt if enabled by the ERFCI bit in the UARTx_IER_SET register.

In loopback mode (LOOP_ENA=1), the UARTxCTS bit is disconnected from the UARTxCTS input pin. Instead, it is directly connected to the MRTS bit of the UARTx_MCR register.

Previous implementations of the UART did not have this register. It is implemented to allow for hardware flow control between the UART and an external device.

UARTx_THR Registers

The write-only transmit hold (UARTx_THR) registers, shown in Figure 15-12, is the UART's transmit buffer. The THRE bit in the UARTx_LSR registers indicate whether UARTx_THR is ready for new data. Writes to UARTx_THR automatically propagate to the internal TSR register as soon as TSR is ready. Then transmit operation is initiated immediately.

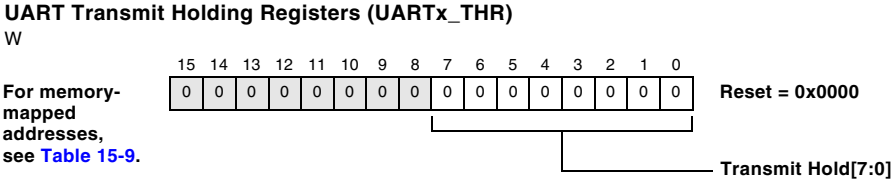


Figure 15-12. UART Transmit Holding Registers

Table 15-9. UART Transmit Holding Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_THR	0xFFC0 0428
UART1_THR	0xFFC0 2028

UARTx_RBR Registers

The read-only UARTx_RBR registers, shown in Figure 15-13, is the UART's receive buffer. It is updated by the internal RSR register when a complete data word is received or when there is pending data in the receive FIFO. Newly available data is signalled by the DR bit in the UARTx_LSR register.

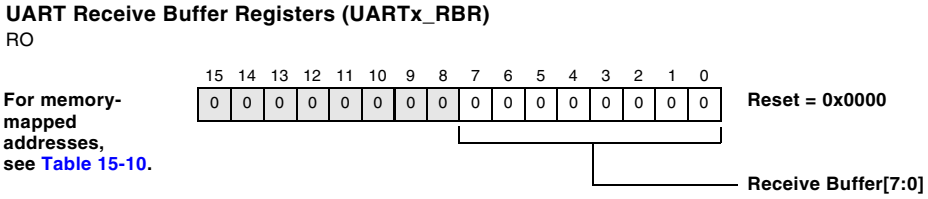


Figure 15-13. UART Receive Buffer Registers

Table 15-10. UART Receive Buffer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_RBR	0xFFC0 042C
UART1_RBR	0xFFC0 202C

UARTx_IER_SET and UARTx_IER_CLEAR Registers

The interrupt enable register is not implemented as a data register. Instead it is controlled by the UARTx_IER_SET and UARTx_IER_CLEAR register pair. Writing ones to UARTx_IER_SET enables interrupts, writing UARTx_IER_CLEAR disables them. Reads from either register return the enabled bits. This way, different interrupt service routines can control transmit, receive, and status interrupts independently and gracefully.

The UARTx_IER registers, shown in Figure 15-14 and Figure 15-15, are used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present.

Each UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless whether DMA is enabled or not. If no DMA channels are assigned to the UART, set the EGLSI bit in the UARTx_GCTL register to reroute transmit and receive interrupts to the status interrupt output.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

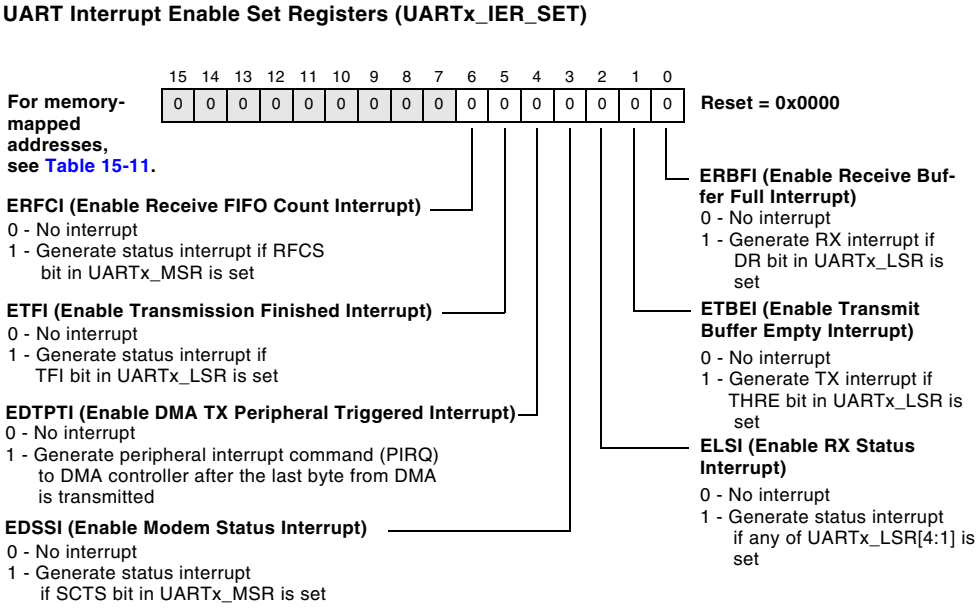


Figure 15-14. UART Interrupt Enable Set Registers

Table 15-11. UART Interrupt Enable Set Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_IER_SET	0xFFC0 0420
UART1_IER_SET	0xFFC0 2020

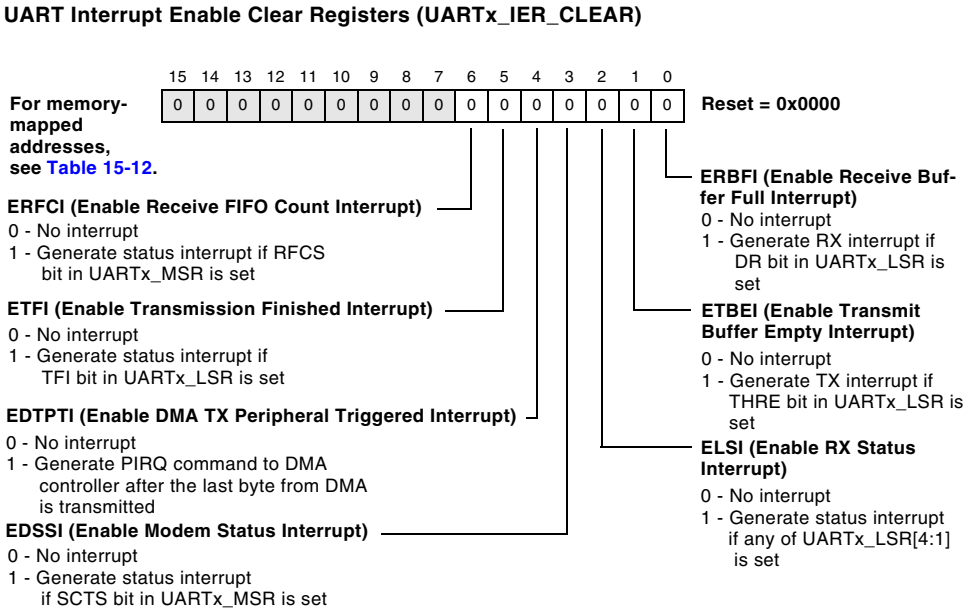


Figure 15-15. UART Interrupt Enable Clear Registers

Table 15-12. UART Interrupt Enable Clear Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_IER_CLEAR	0xFFC0 0424
UART1_IER_CLEAR	0xFFC0 2024

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART `ERBFI` and/or `ETBEI` interrupts in the `UARTx_IER` register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The ELSI bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UARTx_LSR register:

- Receive overrun error (OE)
- Receive parity error (PE)
- Receive framing error (FE)
- Break interrupt (BI)

The EDSSI bit enables a modem status interrupt on the same status interrupt channel when the SCTS bit in the UARTx_MSR register is set. This indicates CTS re-assertion. Write-1-to-clear (W1C) the SCTS bit to clear the interrupt request.

The ERFCI bit enables the receive buffer threshold interrupt if signalled by the RFCS bit. Read the UARTx_RBR register sufficient times to clear the interrupt request.

The ETFI bit enables interrupt generation on the status interrupt channel when both the transmit buffer register and transmit shift register are empty as indicated by the TFI bit in the UARTx_LSR register. The ETFI interrupt can be used to avoid expensive polling of the TEMT bit, when the UART clock or line drivers should be disabled after transmission has completed. W1C the TFI bit to clear the interrupt request. In DMA operation, the ETDPTI bit's functionality might be preferred.

The ETDPTI bit is required for DMA transmit operation only. It enables the DMA completion interrupt to be delayed until the data has left the UART completely. If set, it can generate a DMA interrupt by the time the TEMT bit goes high after the last DMA data word is transmitted.

If the ETDPTI bit is cleared, the DMA completion interrupt is generated when either the last data word is transferred from memory to the DMA FIFO (DMA's SYNC bit cleared) or when the last word has left the DMA FIFO (SYNC bit set). If ETDPTI is set, usually the DMA's DI_EN is not set in

a STOP mode DMA. Thus, the normal completion interrupt is suppressed. Rather, the TEMT event is signalled through the DMA controller and triggers the DMA interrupt. If both, DI_EN and ETDPTI are set, two interrupts are requested at the end of a STOP mode DMA.

The UARTx_IIR registers are not present on this implementation. Signalling interrupt sources can be identified by interrogating UARTx_LSR and UARTx_MSR status registers.

UARTx_DLL and UARTx_DLH Registers

The two 8-bit clock divisor latch registers (UARTx_DLH and UARTx_DLL) build a 16-bit clock divisor value. They divide the system clock SCLK down to the bit clock. These registers are shown in Figure 15-16.

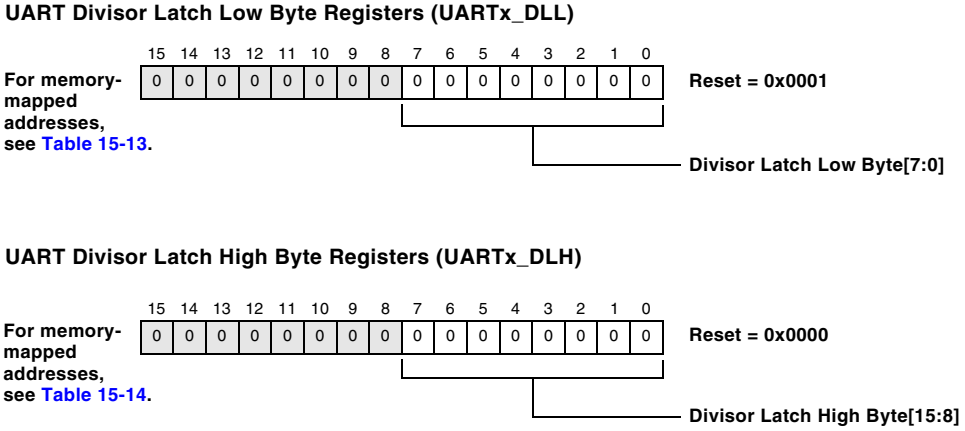


Figure 15-16. UART Divisor Latch Registers

Table 15-13. UART Divisor Latch Low Byte Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_DLL	0xFFC0 0400
UART1_DLL	0xFFC0 2000

Table 15-14. UART Divisor Latch High Byte Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_DLH	0xFFC0 0404
UART1_DLH	0xFFC0 2004

Note the 16-bit divisor formed by UARTx_DLH and UARTx_DLL resets to 0x0001, resulting in high clock frequency by default. If the UART is not used, disabling the UART clock saves power.

Note that the bit rate depends also on the EDBO bit in the UARTx_GCTL register. Refer to "Bit Rate Generation" on page 15-18.

UARTx_SCR Registers

The contents of the 8-bit scratch (UARTx_SCR) registers, shown in Figure 15-17, are reset to 0x00. They are used for general-purpose data storage and do not control the UART hardware in any way.

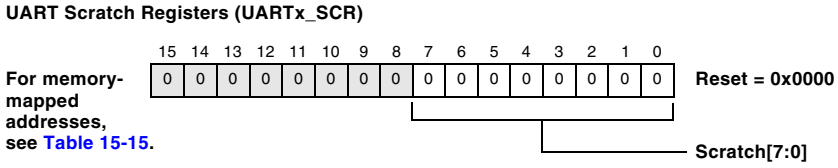


Figure 15-17. UART Scratch Registers

Table 15-15. UART Scratch Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_SCR	0xFFC0 041C
UART1_SCR	0xFFC0 201C

UARTx_GCTL Registers

The global control (UARTx_GCTL) registers, shown in Figure 15-18, contain the enable bit for internal UART clocks and for the IrDA mode of operation of the UARTs.

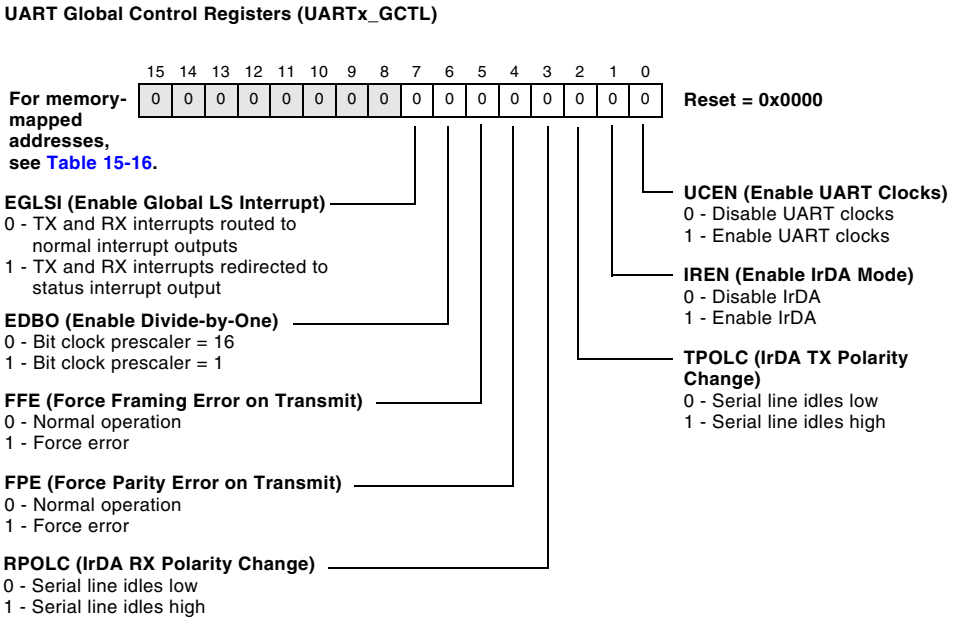


Figure 15-18. UART Global Control Registers

Table 15-16. UART Global Control Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
UART0_GCTL	0xFFC0 0408
UART1_GCTL	0xFFC0 2008

The UCEN bit enables the UART clocks. It also resets the state machine and control registers when cleared. Note that the UCEN bit was not present in previous UART implementations. It is introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

The EDBO bit enables bypassing of the divide-by-16 prescaler in bit clock generation. This improves bit rate granularity, especially at high bit rates. See "Bit Rate Generation" on page 15-18. Do not set this bit in IrDA mode.

The EGLSI bit redirects TX and RX interrupt requests to the status interrupt output of the UART by ORing them with all other kinds of UART status interrupt requests. Set this bit when no DMA channel is associated with the UART. Enabling EGLSI disables the RX/TX interrupt channels and negates the EDTPTI bit.

Programming Examples

The following programming examples show how to use the UART.

The subroutine in Listing 15-1 shows a typical UART initialization sequence.

Listing 15-1. UART Initialization

```
/*************************************************************
 * Configures UART in 8 data bits, no parity, 1 stop bit mode.
 * Input parameters: r0 holds divisor latch value to be
 *                    written into
 *                    DLH:DLL registers.
 *                    p0 contains the UARTx_GCTL register address
 * Return values:    none
 *************************************************************/
uart_init:
    [--sp] = r7;
    r7 = UCEN (z); /* First of all, enable UART clock */
    w[p0+UART0_GCTL-UART0_GCTL] = r7;

     w[p0+UART0_DLL-UART0_GCTL] = r0; /* write lower byte to DLL
*/
    r7 = r0 >> 8;
     w[p0+UART0_DLH-UART0_GCTL] = r7; /* write upper byte to DLH
*/

    r7 = STB | WLS(8) (z);              /* config to */
    w[p0+UART0_LCR-UART0_GCTL] = r7; /* 8 bits, no parity, 2
stop bits */

    r7 = [sp++];
    rts;
uart_init.end:
```

The subroutine in Listing 15-2 performs autobaud detection similarly to UART boot.

Listing 15-2. UART Autobaud Detection Subroutine

```
/*************************************************************
 * Assuming 8 data bits, this functions expects a '@'
 *  (ASCII 0x40) character
 *  on the UARTx RX pin. A Timer performs the autobaud detection.
 *  Input parameters: p0 contains the UARTx_GCTL register address
 *                    p1 contains the TIMERx_CONFIG register
 *  address
 *  Return values:    r0 holds timer period value (equals 8 bits)
*************************************************************/
uart_autobaud:
    [--sp] = (r7:5,p5:5);
    r5.h = hi(TIMER0_CONFIG);  /* for generic timer use calculate
*/
    r5.l = lo(TIMER0_CONFIG);  /* specific bits first */
    r7 = p1;
    r7 = r7 - r5;
    r7 >>= 4;  /* r7 holds the 'x' of TIMERx_CONFIG now */
    r5 = TIMEN0 (z);
    r5 <<= r7;  /* r5 holds TIMENx/TIMDISx now */
    r6 = TRUN0 | TOVL_ERR0 | TIMIL0 (z);
    r6 <<= r7;
    CC = r7 <= 3;
    r7 = r6 << 12;
    if !CC r6 = r7; /* r6 holds TRUNx | TOVL_ERRx | TIMILx */

    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
```

```
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6; /* clear pending
latches */
    /* period capture, falling edge to falling edge */
    r7 = TIN_SEL | IRQ_ENA | PERIOD_CNT | WDTH_CAP (z);
    w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
    w[p5+TIMER_ENABLE-TIMER_STATUS] = r5;

uart_autobaud.wait:  /* wait for timer event */
        r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
        r7 = r7 & r5;
        CC = r7 == 0;
        if CC jump uart_autobaud.wait;
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6; /* clear pending
latches */
    /* Save period value to R0 */
    r0 = [p1 + TIMER0_PERIOD - TIMER0_CONFIG];

    /* delay processing as autobaud character is still ongoing */
    r7 = OUT_DIS | IRQ_ENA | PERIOD_CNT | PWM_OUT (z);
    w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
    w[p5 + TIMER_ENABLE - TIMER_STATUS] = r5;

uart_autobaud.delay:
        r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
        r7 = r7 & r5;
        CC = r7 == 0;
        if CC jump uart_autobaud.delay;
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5;
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
    (r7:5,p5:5) = [sp++];
    rts;
uart_autobaud.end:
```

The parent routine in Listing 15-3 performs autobaud detection using UART0 and TIMER2.

Listing 15-3. UART Autobaud Detection Parent Routine

```
    p0.l = lo(PORTG_FER); /* function enable on UART0 pins PG12
and PG13 */
    p0.h = hi(PORTG_FER); /* by default PORTG_MUX register is all
set */
    r0 = PG12 | PG13 (z)
    w[p0] = r0;
    p0.l = lo(UART0_GCTL); /* select UART 0 */
    p0.h = hi(UART0_GCTL);
    p1.l = lo(TIMER2_CONFIG); /* select TIMER 2 */
    p1.h = hi(TIMER2_CONFIG);
    call uart_autobaud;
    r0 >>= 7;       /* divide PERIOD value by (16 x 8) */
    call uart_init;
    ...
```

The subroutine in Listing 15-4 transmits a character by polling operation.

Listing 15-4. UART Character Transmission

```
/*******************************************************
 * Transmit a single byte by polling the THRE bit.
 * Input parameters: r0 holds the character to be transmitted
 *                   p0 contains UARTx_GCTL register address
 * Return values: none
*******************************************************/
uart_putc:
    [--sp] = r7;
uart_putc.wait:
        r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
```

```
        CC = bittst(r7, bitpos(THRE));
        if !CC jump uart_putc.wait;
    w[p0+UART0_THR-UART0_GCTL] = r0; /* write initiates transfer
*/
    r7 = [sp++];
    rts;
uart_putc.end:
```

Use the routine shown in Listing 15-5 to transmit a C-style string that is terminated by a null character.

Listing 15-5. UART String Transmission

```
/*************************************************************
 * Transmit a null-terminated string.
 * Input parameters: p1 points to the string
 *                   p0 contains UARTx_GCTL register address
 * Return values: none
*************************************************************/
uart_puts:
    [--sp] = rets;
    [--sp] = r0;
uart_puts.loop:
        r0 = b[p1++] (z);
        CC = r0 == 0;
        if CC jump uart_puts.exit;
        call uart_putc;
        jump uart_puts.loop;
uart_puts.exit:
    r0 = [sp++];
    rets = [sp++];
    rts;
uart_puts.end:
```

Note that polling the UART0_LSR register for transmit purposes does not cause side effects on receive status bits as on former implementations.

In non-DMA interrupt operation, the three UART interrupt request lines may or may not be ORed together in the SIC controller or by the EGLSI control bit. If they had three different service routines, they may look as shown in Listing 15-6.

Listing 15-6. UART Non-DMA Interrupt Operation

```
isr_uart_rx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = w[p0+UART0_RBR-UART0_GCTL] (z);
    b[p4++] = r7;
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_rx.end:

isr_uart_tx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = b[p3++] (z);
    CC = r7 == 0;
    if CC jump isr_uart_tx.final;
    w[p0+UART0_THR-UART0_GCTL] = r7;
    r7 = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_tx.final:
    r7 = ETBEI (z) ;
    w[p0+UART0_IER_CLR] = r7; /* clear TX interrupt enable */
```

```
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:

isr_uart_error:
    [--sp] = astat;
    [--sp] = (r7:6);
    r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
    r6 = OE | BI | FE | PE (z);
    w[p0+UART0_LSR-UART0_GCTL] = r6;
    /* do something with the error */
    (r7:6) = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_error.end:
```

Listing 15-7 transmits a string by DMA operation, waits until DMA completes and sends an additional string by polling. Note the importance of the SYNC bit.

Listing 15-7. UART Transmission SYNC Bit Use

```
.section data;
.byte sHello[] = 'Hello Blackfin User',13,10,0;
.byte sWorld[] = 'How is life?',13,10,0;

.section program;
    ...
    p1.l = lo(IMASK);
    p1.h = hi(IMASK);
    r0.l = lo(isr_uart_tx);     /* register service routine */
    r0.h = hi(isr_uart_tx);     /* UART0 TX defaults to IVG10 */
```

```
    r0 = [p1 + IMASK - IMASK]; /* unmask interrupt in CEC */
    bitset(r0, bitpos(EVT_IVG10));
    [p1] = r0;
    p1.l = lo(SIC_IMASK0);
    p1.h = hi(SIC_IMASK0);       /* unmask interrupt in SIC */
    r0.l = 0x8000;
    r0.h = 0x0000;
    [p1] = r0;
    [--sp] = reti;        /* enable nesting of interrupts */

    p5.l = lo(DMA7_CONFIG);     /* setup DMA in STOP mode */
    p5.h = hi(DMA7_CONFIG);
    r7.l = lo(sHello);
    r7.h = hi(sHello);
    [p5+DMA7_START_ADDR-DMA7_CONFIG] = r7;
    r7 = length(sHello) (z);
    r7+= -1;    /* don't send trailing null character */
    w[p5+DMA7_X_COUNT-DMA7_CONFIG] = r7;
    r7 = 1;
    w[p5+DMA7_X_MODIFY-DMA7_CONFIG] = r7;
    r7 = FLOW_STOP | WDSIZE_8 | DI_EN | SYNC | DMAEN (z);
    w[p5] = r7;

    p0.l = lo(UART0_GCTL); /* select UART 0 */
    p0.h = hi(UART0_GCTL);
    r0 = ETBEI (z);         /* enable and issue first request */
    w[p0+UART0_IER-UART0_GCTL] = r0;

wait4dma: /* just one way to synchronize with the service routine
*/
    r0 = w[p5+DMA7_IRQ_STATUS-DMA7_CONFIG] (z);
    CC = bittst(r0,bitpos(DMA_RUN));
    if CC jump wait4dma;
    p1.l=lo(sWorld);
```

```
    p1.h=hi(sWorld);
    call uart_puts;

forever: jump forever;

isr_uart_tx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = DMA_DONE (z);    /* W1C interrupt request */
    w[p5+DMA7_IRQ_STATUS-DMA7_CONFIG] = r7;
    r7 = ETBEI (z);
    w[p0+UART0_IER_CLEAR-UART0_GCTL] = r7;
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:
```

16 TWO WIRE INTERFACE CONTROLLER

This chapter describes the two wire interface (TWI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of TWIs for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For TWI interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the TWIs is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each TWI, refer to Chapter A, "System MMR Assignments".

TWI behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF50x Processor" on page 16-59.

Overview

The TWI controller allows a device to interface to an inter IC bus as specified by the *Philips I²C Bus Specification version 2.1* dated January 2000.

The TWI is fully compatible with the widely used I²C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth the TWI controller can be set up with transfer initiated interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I²C bus protocol. The TWI controller includes these features:

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

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

Interface Overview

Figure 16-1 provides a block diagram of the TWI controller. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCL rate, to and from other TWI devices. The SCL signal synchronizes the shifting and sampling of the data on the serial data pin.

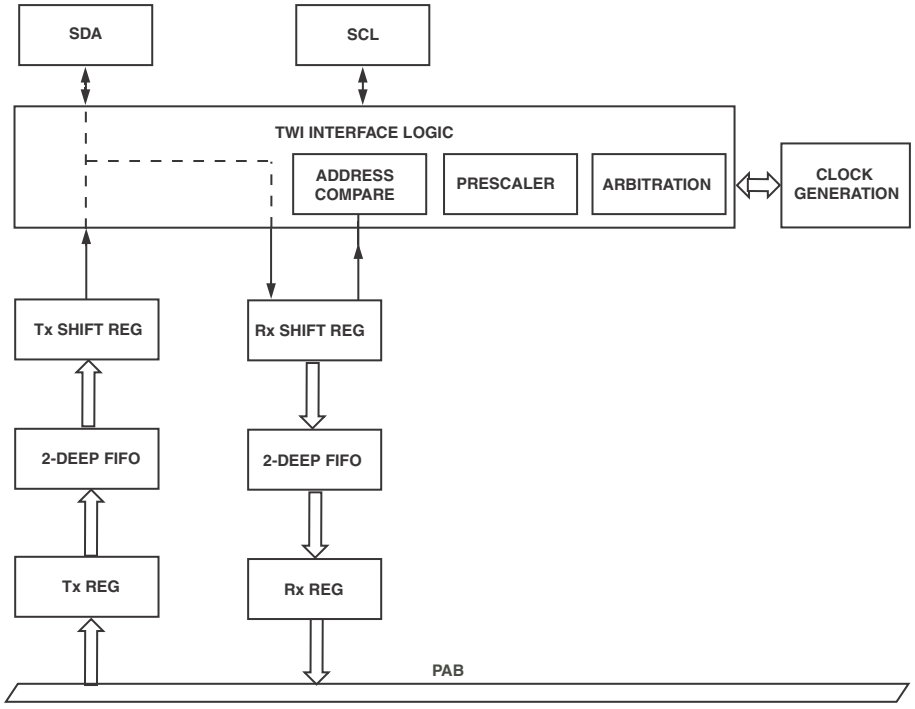


Figure 16-1. TWI Block Diagram

External Interface

The SDA (serial data) and SCL (serial clock) signals are open drain and as such require pull-up resistors.

Serial Clock Signal (SCL)

In slave mode this signal is an input and an external master is responsible for providing the clock.

In master mode the TWI controller must set this signal to the desired frequency. The TWI controller supports the standard mode of operation (up to 100 KHz) or fast mode (up to 400 KHz).

The TWI control register (TWI_CONTROL) is used to set the PRESCALE value which gives the relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

$$\text{PRESCALE} = f_{SCLK}/10\text{MHz}$$

The PRESCALE value is the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

Serial Data Signal (SDA)

This is a bidirectional signal on which serial data is transmitted or received depending on the direction of the transfer.

TWI Pins

Table 16-1 shows the pins for the TWI. Two bidirectional pins externally interface the TWI controller to the I^2C bus. The interface is simple and no other external connections or logic are required.

Table 16-1. TWI Pins

Pin	Description
SDA	In/Out TWI serial data, high impedance reset value.
SCL	In/Out TWI serial clock, high impedance reset value.

Internal Interfaces

The peripheral bus interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer's model. Status bits can be updated by their respective functional blocks.

The FIFO buffer is configured as a1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event the TWI controller module is accessed as a slave.

The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for SCL clock generation.

The clock generation module is used to generate an external SCL clock when in master mode. It includes the logic necessary for synchronization in a multi-master clock configuration and clock stretching when configured in slave mode.

Description of Operation

The following sections describe the operation of the TWI interface.

TWI Transfer Protocols

The TWI controller follows the transfer protocol of the *Philips I^2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 16-2.

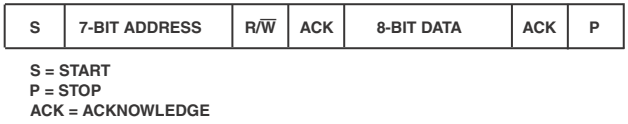


Figure 16-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, Figure 16-3 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI controller successfully transmits one byte of data. The slave has acknowledged both address and data.

Figure 16-3. Data Transfer With Bit Illustration

Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states. If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 16-4.

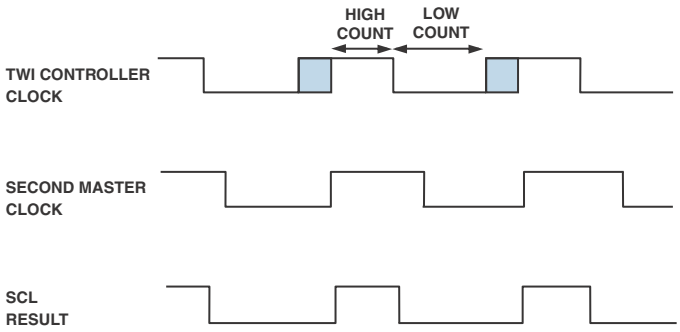


Figure 16-4. TWI Clock Synchronization

The TWI controller's serial clock (SCL) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controller initiates a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 16-5.

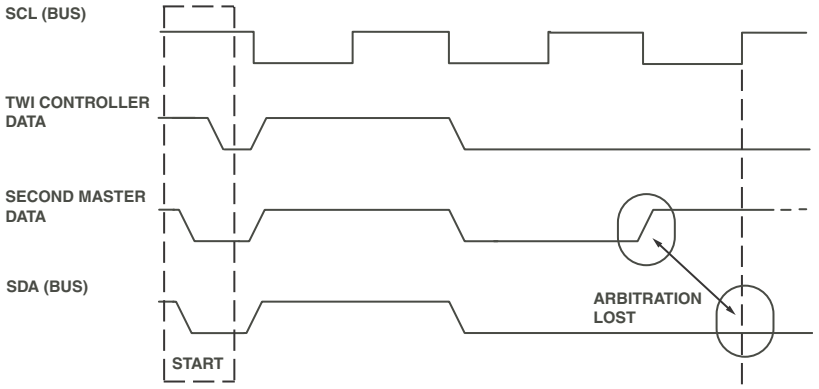


Figure 16-5. TWI Bus Arbitration

The TWI controller monitors the serial data bus (SDA) while SCL is high and if SDA is determined to be an active logic 0 level while the TWI controller's data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time SCL is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controller generates and recognizes these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start "combined" transfers, as shown in Figure 16-6.

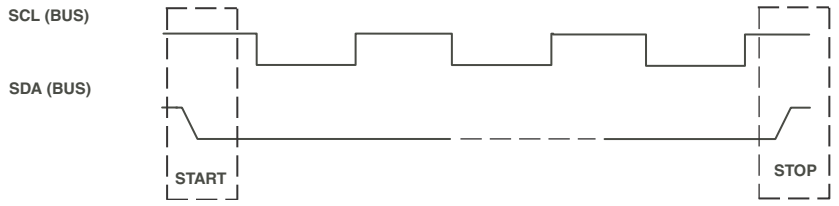


Figure 16-6. TWI Start and Stop Conditions

The TWI controller's special case start and stop conditions include:

- TWI controller addressed as a slave-receiver

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- TWI controller addressed as a slave-transmitter

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a slave transfer error (SERR).

- TWI controller as a master-transmitter or master-receiver

  If the stop bit is set during an active master transfer, the TWI controller issues a stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

General Call Support

The TWI controller always decodes and acknowledges a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). general call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most affected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition (t_{BUF})

Functional Description

The following sections describe the functional operation of the TWI.

General Setup

General setup refers to register writes that are required for both slave mode operation and master mode operation. General setup should be performed before either the master or slave enable bits are set.

- Program the TWI_CONTROL register to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of f_{SCLK}/10MHz

All values should be rounded up to the next whole number. The TWI_ENA bit enable must be set. Note once the TWI controller is enabled a bus

busy condition may be detected. This condition should clear after t_{BUF} has expired assuming no additional bus activity has been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program TWI_SLAVE_ADDR. The appropriate 7 bits are used in determining a match during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (SCL) is stretched and an interrupt is generated until data is written to the transmit FIFO.
3. Program TWI_INT_MASK. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.
4. Program TWI_SLAVE_CTL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 16-2 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 16-2. Slave Mode Setup Interaction

TWI Controller Master	Processor
Interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: SCOMP – Slave transfer complete.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

- Program `TWI_CLKDIV`. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program `TWI_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWI_XMT_DATA8` or `TWI_XMT_DATA16`. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.

3. Program TWI_FIFO_CTL. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8-bits) or with each two bytes transmitted (16-bits).

4. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

5. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 16-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 16-3. Master Mode Transmit Setup Interaction

TWI Controller Master	Processor
Interrupt: XMTEMPTY – Transmit buffer is empty.	Write transmit FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits.

Master Mode Receive

Follow these programming steps for a single master mode receive:

1. Program TWI_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.
2. Program TWI_FIFO_CTL. Indicate if receive FIFO buffer interrupts should occur with each byte received (8-bits) or with each two bytes received (16-bits).
3. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
4. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

Table 16-4 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 16-4. Master Mode Receive Setup Interaction

TWI Controller Master	Processor
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. The following sections guide the programmer in developing a service routine.

Transmit/Receive Repeated Start Sequence

Figure 16-7 shows a repeated start data transmit followed by a data receive sequence.

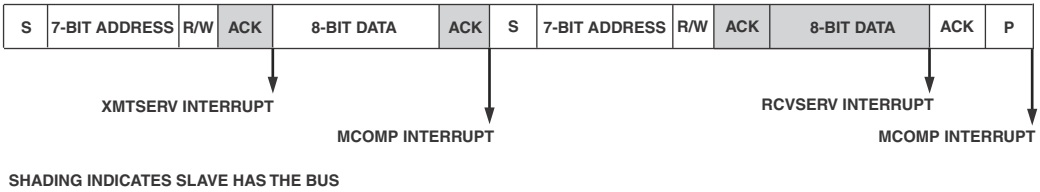


Figure 16-7. Transmit/Receive Data Repeated Start

The following tasks are performed at each interrupt.

- `XMTSERV` interrupt

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, `FIFO_STATUS` indicates the transmit FIFO is empty. When read, `DCNT` would be zero. Set the `RSTART` bit to indicate a repeated start and set the `MDIR` bit if the following transfer will be a data receive.

- MCOMP interrupt

  This interrupt was generated because all data has been transferred (DCNT = 0). If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to receive.

- RCVSERV interrupt

  This interrupt is generated due to the arrival of a byte in the receive FIFO. Simple data handling is all that is required.

- MCOMP interrupt

  The transfer is complete.

Receive/Transmit Repeated Start Sequence

Figure 16-8 illustrates a repeated start data receive followed by a data transmit sequence.

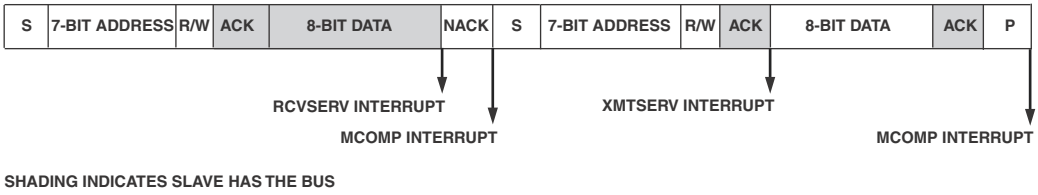


Figure 16-8. Receive/Transmit Data Repeated Start

The tasks performed at each interrupt are:

- RCVSERV interrupt

  This interrupt is generated due to the arrival of a data byte in the receive FIFO. Set the RSTART bit to indicate a repeated start and clear the MDIR bit if the following transfer will be a data transmit.

- MCOMP interrupt

  This interrupt has occurred due to the completion of the data receive transfer. If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to transmit.

- XMTSERV interrupt

  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- MCOMP interrupt

  The transfer is complete.

There is no timing constraint to meet the above conditions—the user can program the bits as required. Refer to "Clock Stretching During Repeated Start Condition" on page 16-20 for more on how the controller stretches the clock during Repeated Start transfers.

Clock Stretching

Clock stretching is an added functionality of the TWI controller in Master Mode operation. This new behavior utilizes self-induced stretching of the I^2C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this.

The TWI Controller as Master supports three modes of clock stretching: "Clock Stretching During FIFO Underflow", "Clock Stretching During FIFO Overflow" on page 16-19 and "Clock Stretching During Repeated Start Condition" on page 16-20.

Clock Stretching During FIFO Underflow

During a master mode transmit, an interrupt is generated at the instant the transmit FIFO becomes empty. At this time, the most recent byte

begins transmission. If the XMTSERV interrupt is not serviced, the concluding “acknowledge” phase of the transfer will be stretched. Stretching of the clock continues until new data bytes are written to the transmit FIFO (TWI_XMT_DATA8 or TWI_XMT_DATA16). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (DCNT = 0) at which time the transmission is concluded (MCOMP) as shown in Figure 16-9 and described in Table 16-5.

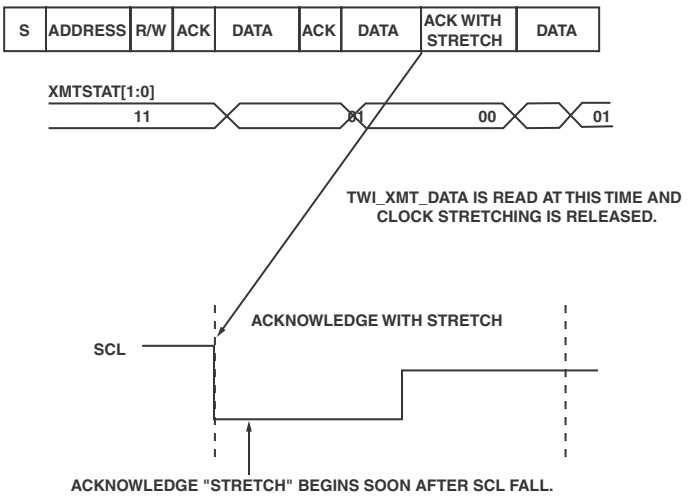


Figure 16-9. Clock Stretching During FIFO Underflow

Table 16-5. FIFO Underflow Case

TWI Controller	Processor
Interrupt: XMTSERV – Transmit FIFO buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).	Acknowledge: Clear interrupt source bits.

Clock Stretching During FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (TWI_RCV_DATA8, TWI_RCV_DATA16). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (DCNT = 0x00) at which time the reception is concluded (MCOMP) as shown in Figure 16-10 and described in Table 16-6.

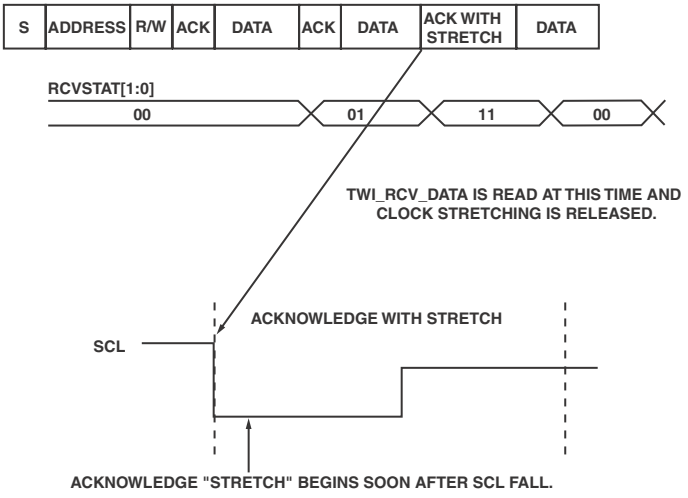


Figure 16-10. Clock Stretching During FIFO Overflow

Table 16-6. FIFO Overflow Case

TWI Controller	Processor
Interrupt: RCVSERV – Receive FIFO buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Clock Stretching During Repeated Start Condition

The repeated start feature in I^2C protocol requires transitioning between two subsequent transfers. With the use of clock stretching, the task of managing transitions becomes simpler and becomes common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock will initiate a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (`DCNT` = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 16-11 and described in Table 16-7.

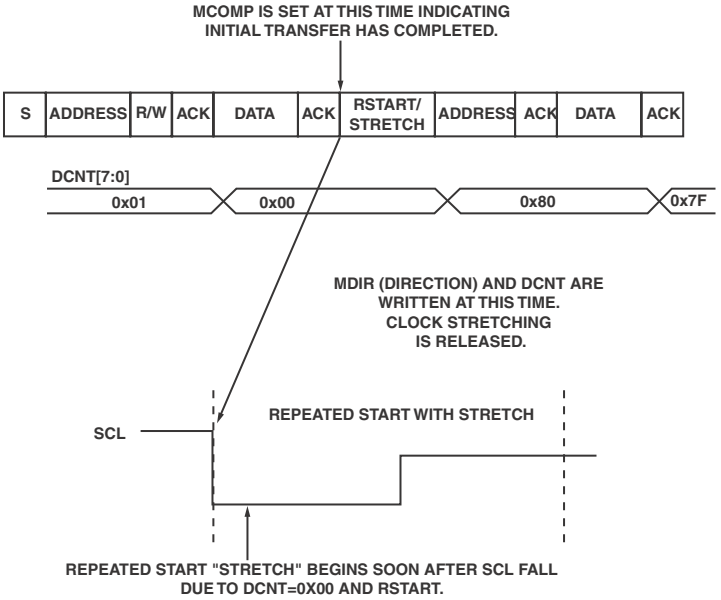


Figure 16-11. Clock Stretching During Repeated Start Condition

Table 16-7. Repeated Start Case

TWI Controller	Processor
Interrupt: MCOMP – Initial transmit has completed and DCNT = 0x00. Note: transfer in progress, RSTART previously set.	Acknowledge: Clear interrupt source bits. Write TWI_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).
Interrupt: RCVSERV – Receive FIFO is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Programming Model

Figure 16-12 and Figure 16-13 illustrate the programming model for the TWI.

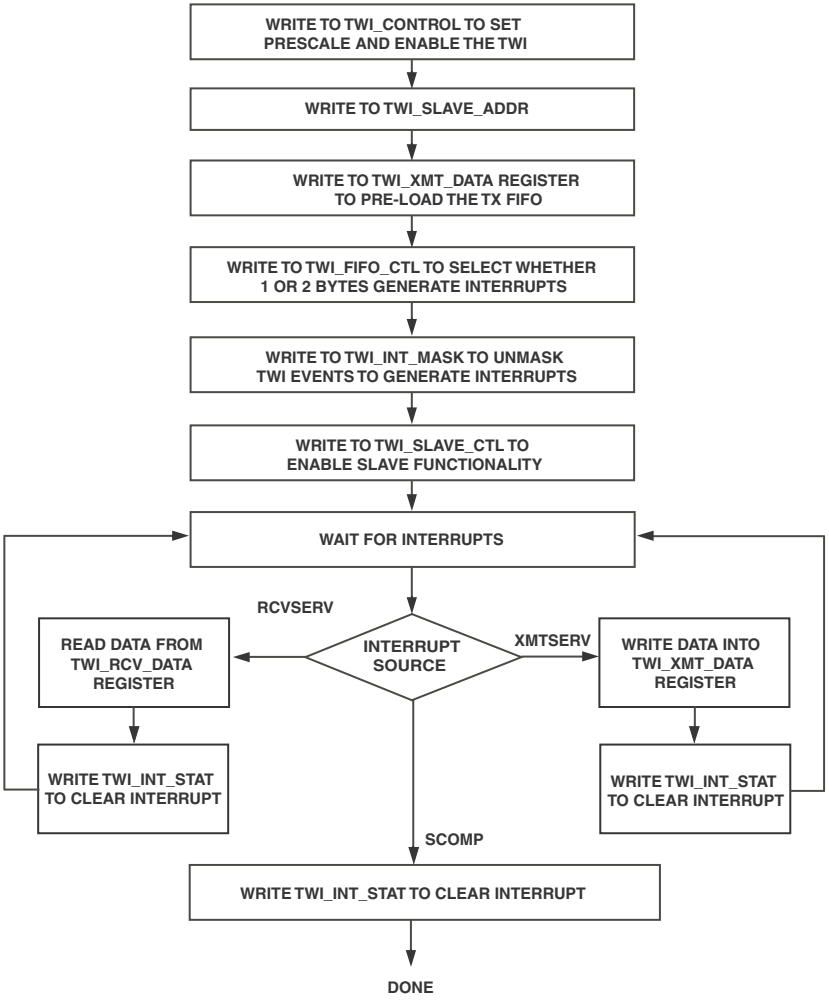


Figure 16-12. TWI Slave Mode

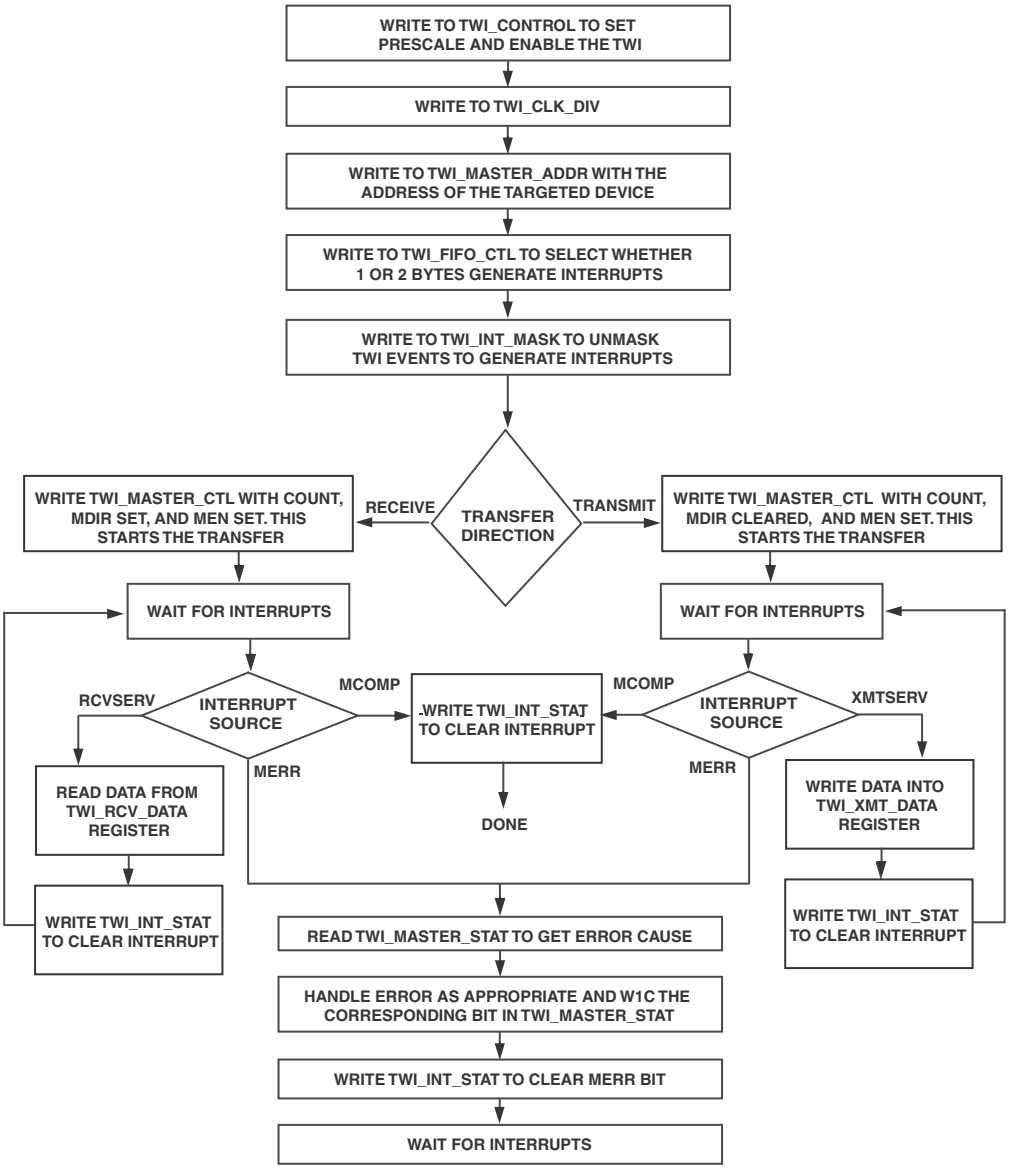


Figure 16-13. TWI Master Mode

Register Descriptions

The TWI controller has 16 registers described in the following sections. Figure 16-14 through Figure 16-31 on page 16-47 illustrate the registers.

TWI CONTROL Register (TWI_CONTROL)

The TWI_CONTROL register is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

PRESCALE = f_{SCLK}/10MHz

SCCB compatibility is an optional feature and should not be used in an I^2C bus system. This feature is turned on by setting the SCCB bit in the TWI_CONTROL register. When this feature is set all slave asserted acknowledgement bits are ignored by this master. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

For either master and/or slave mode of operation, the TWI controller is enabled by setting the TWI_ENA bit in the TWI_CONTROL register. It is recommended that this bit be set at the time PRESCALE is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

The PRESCALE field of the TWI_CONTROL register specifies the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

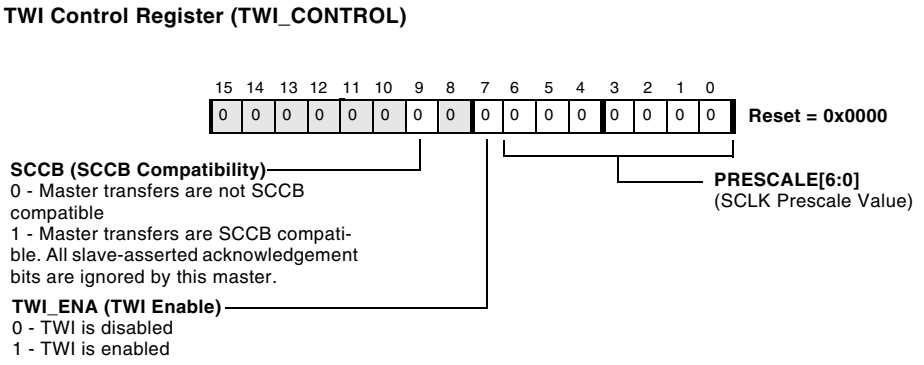


Figure 16-14. TWI Control Register

SCL Clock Divider Register (TWI_CLKDIV)

The clock signal SCL is an output in master mode and an input in slave mode.

During master mode operation, the TWI_CLKDIV register values are used to create the high and low durations of the serial clock (SCL). Serial clock frequencies can vary from 400 KHz to less than 20 KHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

CLKDIV = TWI SCL period / 10 MHz time reference

For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

CLKDIV = 2500 ns / 100 ns = 25

For an SCL with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8. Note that CLKLOW and CLKHI add up to CLKDIV.

The CLKHI field of the TWI_CLKDIV register specifies the number of 10 MHz time reference periods the serial clock (SCL) waits before a new clock

low period begins, assuming a single master. It is represented as an 8-bit binary value.

The CLKLOW field of the TWI_CLKDIV register specifies the number of internal time reference periods the serial clock (SCL) is held low. It is represented as an 8-bit binary value.

SCL Clock Divider Register (TWI_CLKDIV)

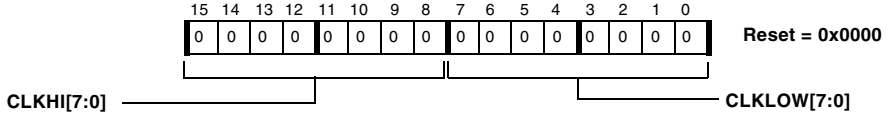


Figure 16-15. SCL Clock Divider Register

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

The TWI_SLAVE_CTL register controls the logic associated with slave mode operation. Settings in this register do not affect master mode operation and should not be modified to control master mode functionality.

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

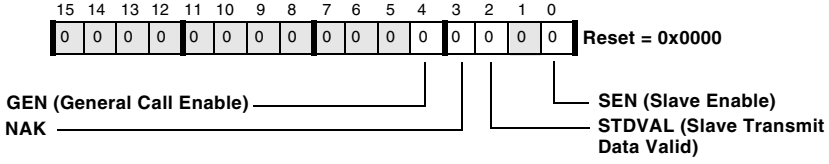


Figure 16-16. TWI Slave Mode Control Register

Additional information for the TWI_SLAVE_CTL register bits includes:

- **General call enable** (GEN)

  General call address detection is available only when slave mode is enabled.

  [0] General call address matching is not enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

- **NAK** (NAK)

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

  [1] Slave receive transfers generate a data NAK (not acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

- **Slave transmit data valid** (STDVAL)

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

  [1] Data in the transmit FIFO is available for a slave transmission.

- **Slave enable** (SEN)

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.

  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

TWI Slave Mode Address Register (TWI_SLAVE_ADDR)

The TWI_SLAVE_ADDR register holds the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

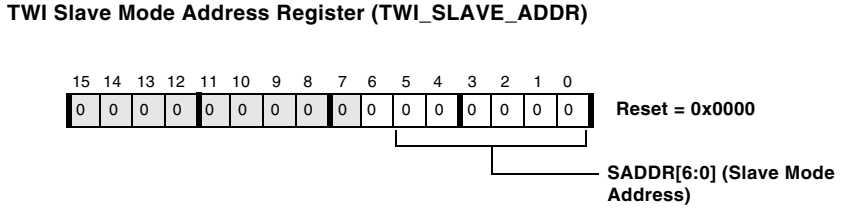


Figure 16-17. TWI Slave Mode Address Register

TWI Slave Mode Status Register (TWI_SLAVE_STAT)

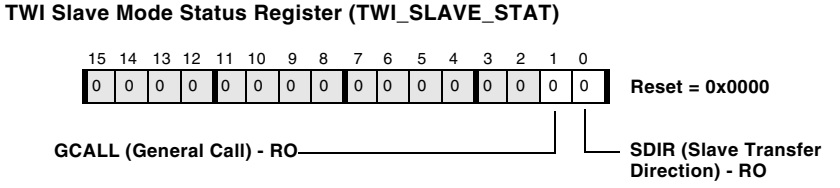


Figure 16-18. TWI Slave Mode Status Register

During and at the conclusion of register slave mode transfers, the TWI_SLAVE_STAT register holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

- **General call** (GCALL)

  This bit self clears if slave mode is disabled (SEN = 0).

  [0] At the time of addressing, the address was not determined to be a general call.

  [1] At the time of addressing, the address was determined to be a general call.

- **Slave transfer direction** (SDIR)

  This bit self clears if slave mode is disabled (SEN = 0).

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

TWI Master Mode Control Register (TWI_MASTER_CTL)

The TWI_MASTER_CTL register controls the logic associated with master mode operation. Bits in this register do not affect slave mode operation and should not be modified to control slave mode functionality.

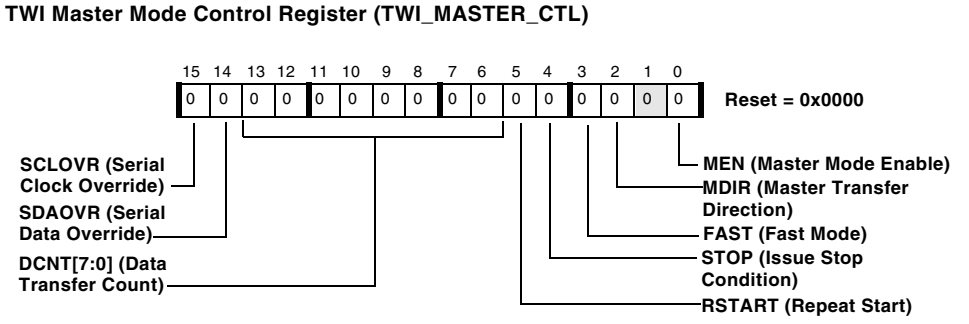


Figure 16-19. TWI Master Mode Control Register

Additional information for the TWI_MASTER_CTL register bits includes:

- **Serial clock override** (SCLOVR)

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

  [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.

  [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Serial data (SDA) override** (SDAOVR)

  This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

  [0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

  [1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Data transfer count** (DCNT[7:0])

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat start** (RSTART)

  [0] Transfer concludes with a stop condition.

  [1] Issue a repeat start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

- **Issue stop condition** (STOP)

  [0] Normal transfer operation.

  [1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt mask register (TWI_INT_MASK) is updated along with any associated status bits.

- **Fast mode** (FAST)

  [0] Standard mode (up to 100K bits/s) timing specifications in use.

  [1] Fast mode (up to 400K bits/s) timing specifications in use.

- **Master transfer direction** (MDIR)

  [0] The initiated transfer is master transmit.

  [1] The initiated transfer is master receive.

- **Master mode enable** (MEN)

  This bit self clears at the completion of a transfer. This includes transfers terminated due to errors.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDA, SCL) are no longer driven. Write-1-to-clear status bits are not affected.

  [1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

TWI Master Mode Address Register (TWI_MASTER_ADDR)

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI_MASTER_ADDR register. When programming this register, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to this register. For example, if the slave address is b#1010000X, where X is the read/write bit, then TWI_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the master mode control register.

TWI Master Mode Address Register (TWI_MASTER_ADDR)

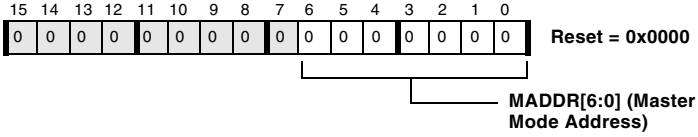


Figure 16-20. TWI Master Mode Address Register

TWI Master Mode Status Register (TWI_MASTER_STAT)

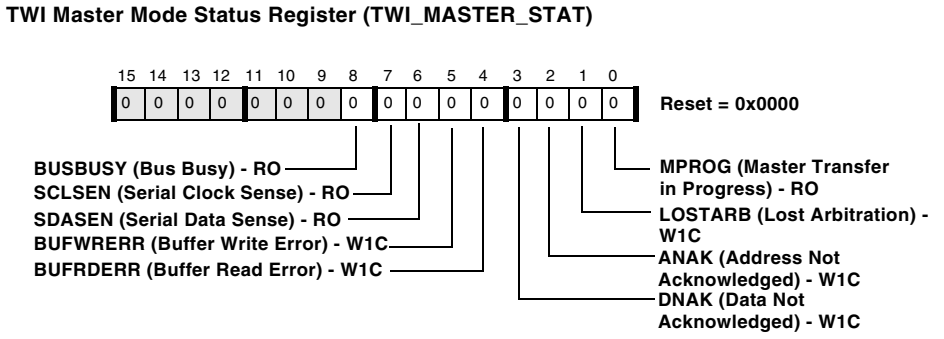


Figure 16-21. TWI Master Mode Status Register

The TWI_MASTER_STAT register holds information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

Note that—while the SCLSEN bit is set (this could be due to having no pull-up resistor on SCL or another agent is driving SCL low)—the acknowledge bits (ANAK and DNAK) do not update. This result occurs because the acknowledge conditions are sampled during the high phase of SCL.

- **Bus busy** (BUSBUSY)

  Indicates whether the bus is currently busy or free. This indication is not limited to only this device but is for all devices. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after t_{BUF}.

[0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.

[1] The bus is busy. Clock or data activity has been detected.

- **Serial clock sense** (SCLSEN)

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [0] An inactive “one” is currently being sensed on the serial clock.

  [1] An active “zero” is currently being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

- **Serial data sense** (SDASEN)

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [0] An inactive “one” is currently being sensed on the serial data line.

  [1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

- **Buffer write error** (BUFWRERR)

  [0] The current master receive has not detected a receive buffer write error.

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time. This bit is W1C.

- **Buffer read error** (BUFRDERR)

  [0] The current master transmit has not detected a buffer read error.

  [1] The current master transfer was aborted due to a transmit buffer read error. At the time data was required by the transmit shift register the buffer was empty. This bit is W1C.

- **Data not acknowledged** (DNAK)

  [0] The current master receive has not detected a NAK during data transmission.

  [1] The current master transfer was aborted due to the detection of a NAK during data transmission. This bit is W1C.

- **Address not acknowledged** (ANAK)

  [0] The current master transmit has not detected NAK during addressing.

  [1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer. This bit is W1C.

- **Lost arbitration** (LOSTARB)

  [0] The current transfer has not lost arbitration with another master.

  [1] The current transfer was aborted due to the loss of arbitration with another master. This bit is W1C.

- **Master transfer in progress** (MPROG)

  [0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

  [1] A master transfer is in progress.

TWI FIFO Control Register (TWI_FIFO_CTL)

The TWI_FIFO_CTL register control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

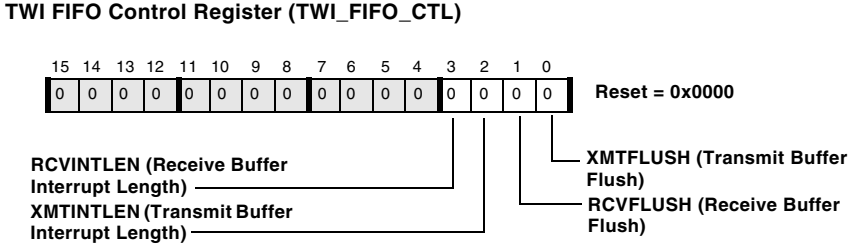


Figure 16-22. TWI FIFO Control Register

Additional information for the TWI_FIFO_CTL register bits includes:

- **Receive buffer interrupt length** (RCVINTLEN)

  This bit determines the rate at which receive buffer interrupts are to be generated. Interrupts may be generated with each byte received or after two bytes are received.

  [0] An interrupt (RCVSERV) is set when RCVSTAT indicates one or two bytes in the FIFO are full (01 or 11).

  [1] An interrupt (RCVSERV) is set when the RCVSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are full (11).

- **Transmit buffer interrupt length** (XMTINTLEN)

  This bit determines the rate at which transmit buffer interrupts are to be generated. Interrupts may be generated with each byte transmitted or after two bytes are transmitted.

  [0] An interrupt (XMTSERV) is set when XMTSTAT indicates one or two bytes in the FIFO are empty (01 or 00).

  [1] An interrupt (XMTSERV) is set when the XMTSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are empty (00).

- **Receive buffer flush** (RCVFLUSH)

  [0] Normal operation of the receive buffer and its status bits.

  [1] Flush the contents of the receive buffer and update the RCVSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active receive the receive buffer in this state responds to the receive logic as if it is full.

- **Transmit buffer flush** (XMTFLUSH)

  [0] Normal operation of the transmit buffer and its status bits.

  [1] Flush the contents of the transmit buffer and update the XMTSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active transmit the transmit buffer in this state responds as if the transmit buffer is empty.

TWI FIFO Status Register (TWI_FIFO_STAT)

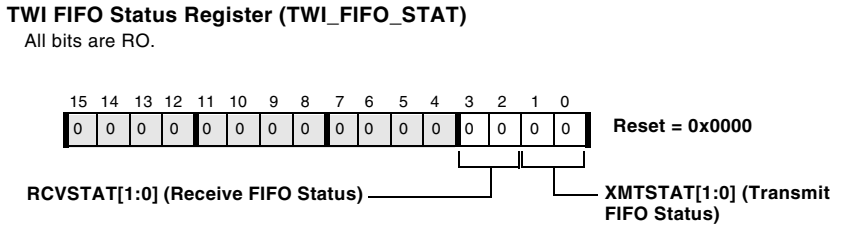


Figure 16-23. TWI FIFO Status Register

TWI FIFO Status

The fields in the TWI_FIFO_STAT register indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

- **Receive FIFO status** (RCVSTAT[1:0])

  The RCVSTAT field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty.

[01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

[10] Reserved

[11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.

- **Transmit FIFO status** (`XMTSTAT[1:0]`)

  The `XMTSTAT` field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

  [01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [10] Reserved

  [11] The FIFO is full and contains two bytes of data.

TWI Interrupt Mask Register (TWI_INT_MASK)

The `TWI_INT_MASK` register enables interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the `TWI_INT_STAT` register. Reading and writing the `TWI_INT_MASK` register does not affect the contents of the `TWI_INT_STAT` register.

TWI Interrupt Mask Register (TWI_INT_MASK)
For all bits, 0 = Interrupt generation disabled, 1 = Interrupt generation enabled.

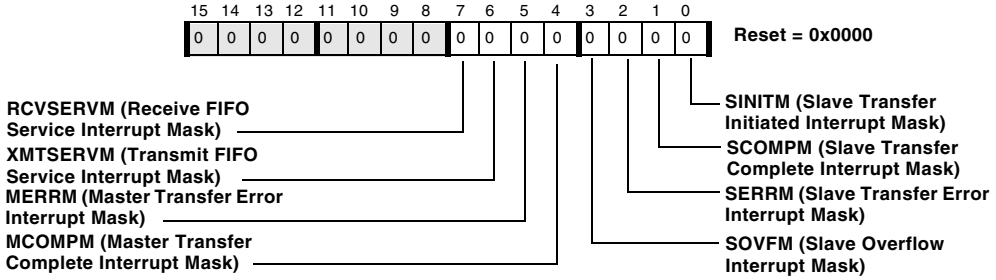


Figure 16-24. TWI Interrupt Mask Register

TWI Interrupt Status Register (TWI_INT_STAT)

TWI Interrupt Status Register (TWI_INT_STAT)
All bits are sticky and W1C.

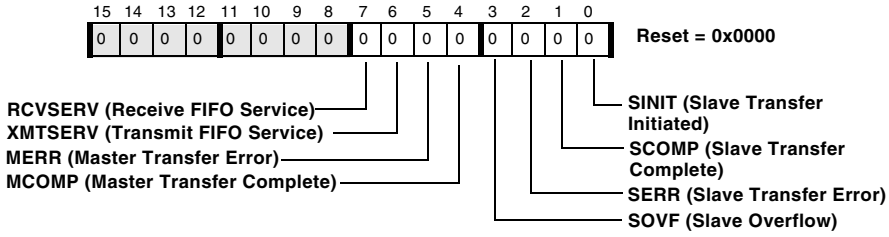


Figure 16-25. TWI Interrupt Status Register

The `TWI_INT_STAT` register contains information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit by writing a 1 to it.

- **Receive FIFO service** (RCVSERV)

  If RCVINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the RCVSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to or 11.

  [0] No errors have been detected.

  [1] The FIFO does not require servicing or the RCVSTAT field has not changed since this bit was last cleared.

- **Transmit FIFO service** (XMTSERV)

  If XMTINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the XMTSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to 00.

  [0] FIFO does not require servicing or XMTSTAT field has not changed since this bit was last cleared.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

- **Master transfer error** (MERR)

  [0] No errors have been detected.

  [1] A master error has occurred. The conditions surrounding the error are indicated by the master status register (TWI_MASTER_STAT).

- **Master transfer complete** (MCOMP)

  [0] The completion of a transfer has not been detected.

  [1] The initiated master transfer has completed. In the absence of a repeat start, the bus has been released.

- **Slave overflow** (SOVF)

  [0] No overflow has been detected.

  [1] The slave transfer complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.

- **Slave transfer error** (SERR)

  [0] No errors have been detected.

  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.

- **Slave transfer complete** (SCOMP)

  [0] The completion of a transfer has not been detected.

  [1] The transfer is complete and either a stop, or a restart was detected.

- **Slave transfer initiated** (SINIT)

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

  [1] The slave has detected an address match and a transfer has been initiated.

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)

The TWI_XMT_DATA8 register holds an 8-bit data value written into the FIFO buffer. Transmit data is entered into the corresponding transmit buffer in a first-in first-out order. For 16-bit PAB writes, a write access to TWI_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With

each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

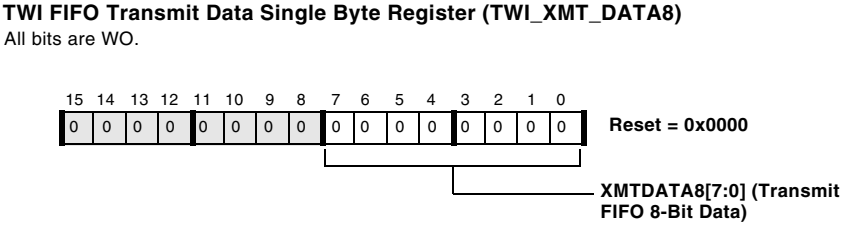


Figure 16-26. TWI FIFO Transmit Data Single Byte Register

TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)

The TWI_XMT_DATA16 register holds a 16-bit data value written into the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access.

The data is written in little endian byte order as shown in Figure 16-27 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

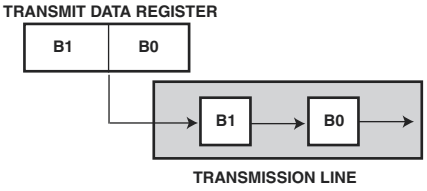


Figure 16-27. Transmit Little Endian Byte Order

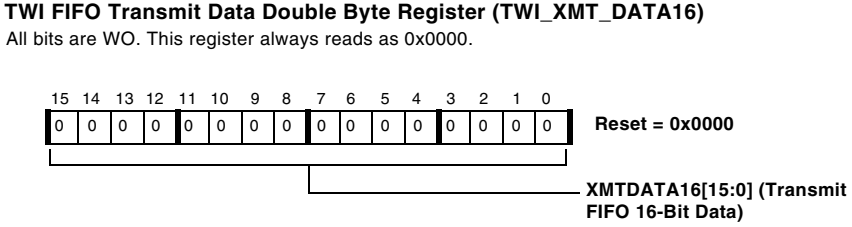


Figure 16-28. TWI FIFO Transmit Data Double Byte Register

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)

The TWI_RCV_DATA8 register holds an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWI_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.

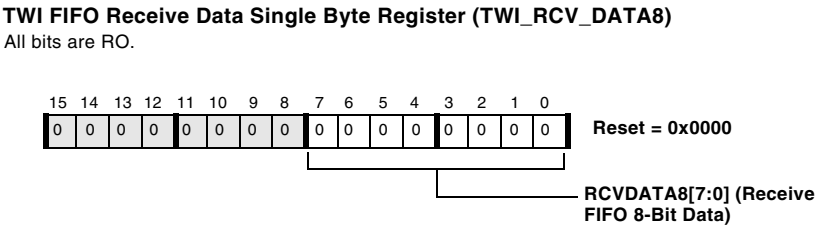


Figure 16-29. TWI FIFO Receive Data Single Byte Register

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)

The TWI_RCV_DATA16 register holds a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access.

The data is read in little endian byte order as shown in Figure 16-30 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated to indicate it is empty. If an access is performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.

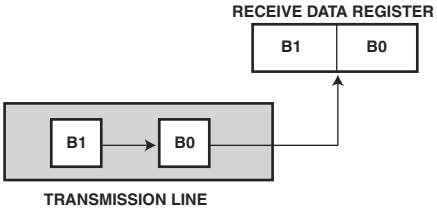


Figure 16-30. Receive Little Endian Byte Order

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)
All bits are WO.

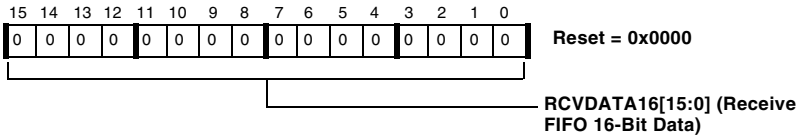


Figure 16-31. TWI FIFO Receive Data Double Byte Register

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

Master Mode Setup

Listing 16-1 shows how to initiate polled receive and transmit transfers in master mode.

Listing 16-1. Master Mode Receive/Transmit Transfer

```
/*************************************************************
 Macro for the count field of the TWI_MASTER_CTL register
 x can be any value between 0 and 0xFE (254). A value of
 0xFF disables the counter.
*************************************************************/
#define TWICount(x) (DCNT & ((x) << 6))

.section L1_data_b;
.byte TX_file[file_size] = "DATA.hex";
```

```
.BYTE RX_CHECK[file_size];
.byte rcvFirstWord[2];

.SECTION program;
_main:
/*********************************************************
TWI Master Initialization subroutine
*********************************************************/

TWI_INIT:
/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz

P1 points to the base of the system MMRs
*********************************************************/

R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;

/*********************************************************
Set CLKDIV:
For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns)
and an internal time reference of 10 MHz (period = 100 ns):
CLKDIV = 2500 ns / 100 ns = 25
For an SCL with a 30% duty cycle, then CLKLOW = 17 (0x11) and
CLKHI = 8.
*********************************************************/
R5 = CLKHI(0x8) | CLKLOW(0x11) (z);
W[P1 + LO(TWI_CLKDIV)] = R5;
```

```
/*********************************************************
enable these signals to generate a TWI interrupt: optional
*********************************************************/
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*********************************************************
The address needs to be shifted one place to the right
e.g., 1010 001x becomes 0101 0001 (0x51) the TWI controller
will actually send out 1010 001x where x is either a 0 for
writes or 1 for reads
*********************************************************/
R6 = 0xBF;
R6 = R6 >> 1;
TWI_INIT.END: W[P1 + LO(TWI_MASTER_ADDR)] = R6;

/******************** END OF TWI INIT **********************/

/*********************************************************
Starting the Read transfer
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or SLOW
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. This will kick off the master transfer
*********************************************************/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
ssync;
```

```
/*********************************************************
Poll the FIFO Status register to know when
2 bytes have been shifted into the RX FIFO
*********************************************************/
Rx_stat:
R1 = W[P1 + LO(TWI_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF ! cc jump Rx_stat;
R0 = W[P1 + LO(TWI_RCV_DATA16)](Z); /* Read data from the RX fifo
*/
ssync;

/*********************************************************
check that master transfer has completed
MCOMP will be set when Count reaches zero
*********************************************************/
M_COMP:
  R1 = W[P1 + LO(TWI_INT_STAT)](z);
  CC = BITTST (R1, bitpos(MCOMP));
  if ! CC jump M_COMP;
M_COMP.END:  W[P1 + LO(TWI_INT_STAT)] = R1;

/* load the pointer with the address of the transmit buffer */
P2.H = TX_file;
P2.L = TX_file;

/*********************************************************
Pre-load the tx FIFO with the first two bytes: this is
necessary to avoid the generation of the Buffer Read Error
(BUFRDERR) which occurs whenever a transmit transfer is
initiated while the transmit buffer is empty
*********************************************************/
```

```
R3 = W[P2++](Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
Initiating the Write operation
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or Standard
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. Setting this bit will kick off the transfer
*********************************************************/
R1 = TWICount(0xFE) | FAST | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
SSYNC;

/*********************************************************
  loop to write data to a TWI slave device P3 times
*********************************************************/
P3 = length(TX_file);

LSETUP (Loop_Start1, Loop_End1) LC0 = P3;
  Loop_Start1:
    /*******************************************************
    check that there's at least one byte location empty in
    the tx fifo
    *******************************************************/
    XMTSERV_Status:
    R1 = W[P1 + LO(TWI_INT_STAT)](z);
    CC = BITTST (R1, bitpos(XMTSERV)); /* test XMTSERV bit */
    if ! CC jump XMTSERV_Status;
```

```
    W[P1 + LO(TWI_INT_STAT)] = R1; /* clear status */
    SSYNC;

    /*******************************************************
    write byte into the transmit FIFO
    *******************************************************/
    R3 = B[P2++](Z);
    W[P1 + LO(TWI_XMT_DATA8)] = R3;
  Loop_End1:  SSYNC;

/* check that master transfer has completed */
M_COMP1:
R1 = W[P1 + LO(TWI_INT_STAT)](z);
CC = BITTST (R1, bitpos(MCOMP1));
if ! CC jump M_COMP;
M_COMP1.END:W[P1 + LO(TWI_INT_STAT)] = R1;

idle;
_main.end:
```

Slave Mode Setup

Listing 16-2 shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine `_TWI_ISR` shown in Listing 16-3.

Listing 16-2. Slave Mode Setup

```
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/
#include "startup.h"

#define file_size 254
#define SYSMMR_BASE 0xFFC00000
```

```
#define COREMMR_BASE 0xFFE00000

.GLOBAL _main;
.EXTERN _TWI_ISR;

.section L1_data_b;
.BYTE TWI_RX[file_size];
.BYTE TWI_TX[file_size] = "transmit.dat";

.section L1_code;
_main:

/*********************************************************
TWI Slave Initialization subroutine
*********************************************************/
TWI_SLAVE_INIT:

/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz
P1 points to the base of the system MMRs
P0 points to the base of the core MMRs
*********************************************************/
R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;

/*********************************************************
Slave address
program the address to which this slave will respond to.
this is an arbitrary 7-bit value
*********************************************************/
```

```
R1 = 0x5F;
W[P1 + LO(TWI_SLAVE_ADDR)] = R1;

/*********************************************************
Pre-load the TX FIFO with the first two bytes to be
transmitted in the event the slave is addressed and a transmit
is required
*********************************************************/
R3=0xB537(Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
FIFO Control determines whether an interrupt is generated
for every byte transferred or for every two bytes.
A value of zero which is the default, allows for single byte
events to generate interrupts
*********************************************************/
R1 = 0;
 W[P1 + LO(TWI_FIFO_CTL)] = R1;

/*********************************************************
enable these signals to generate a TWI interrupt
*********************************************************/
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*********************************************************
Enable the TWI Slave
Program the Slave Control register with:
1. Slave transmit data valid (STDVAL) set so that the contents of
the TX FIFO can be used by this slave when a master requests data
from it.
2. Slave Enable SEN to enable Slave functionality
*********************************************************/
```

```
R1 = STDVAL | SEN;
W[P1 + LO(TWI_SLAVE_CTL)] = R1;
TWI_SLAVE_INIT.END:

P2.H = HI(TWI_RX);
P2.L = LO(TWI_RX);

P4.H = HI(TWI_TX);
P4.L = LO(TWI_TX);
/*********************************************************
Remap the vector table pointer from the default __I10HANDLER
to the new _TWI_ISR interrupt service routine
*********************************************************/
R1.H = HI(_TWI_ISR);
R1.L = LO(_TWI_ISR);
[P0 + LO(EVT10)] = R1;   /* note that P0 points to the base of
the core MMR registers */

/*********************************************************
ENABLE TWI generate to interrupts at the system level
*********************************************************/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWI));
[P1 + LO(SIC_IMASK)] = R1;

/*********************************************************
ENABLE TWI to generate interrupts at the core level
*********************************************************/
R1 = [P0 + LO(IMASK)];
BITSET(R1,BITPOS(EVT_IVG10));
[P0 + LO(IMASK)] = R1;
```

```
/*************************************************************
 wait for interrupts
*************************************************************/
idle;

_main.END:
```

Listing 16-3. TWI Slave Interrupt Service Routine

```
/*************************************************************
Function:  _ TWI_ISR
Description:  This ISR is executed when the TWI controller
detects a slave initiated transfer. After an interrupt is ser-
viced, its corresponding bit is cleared in the TWI_INT_STAT
register. This done by writing a 1 to the particular bit posi-
tion. All bits are write 1 to clear.
*************************************************************/
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/

.GLOBAL _TWI_ISR;

.section L1_code;
_TWI_ISR:

/*************************************************************
read the source of the interrupt
*************************************************************/
R1 = W[P1 + LO(TWI_INT_STAT)](z);

/*************************************************************
Slave Transfer Initiated
*************************************************************/
```

```
CC = BITTST(R1, BITPOS(SINIT));
if ! CC JUMP RECEIVE;
R0 = SINIT (Z);
W[P1 + LO(TWI_INT_STAT)] = R0;  /* clear interrupt source bit */
ssync;

/*********************************************************
Receive service
*********************************************************/
RECEIVE:
CC = BITTST(R1, BITPOS(RCVSERV));
if ! CC JUMP TRANSMIT;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);   /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */
R0 = RCVSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /*clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */

/*********************************************************
Transmit service
*********************************************************/
TRANSMIT:
CC = BITTST(R1, BITPOS(XMTSERV));
if ! CC JUMP SlaveError;
R0 = B[P4++](Z);
W[P1 + LO(TWI_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */
```

```
/*************************************************************
slave transfer error
*************************************************************/
SlaveError:
CC = BITTST(R1, BITPOS(SERR));
if ! CC JUMP SlaveOverflow;
R0 = SERR(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */

/*************************************************************
slave overflow
*************************************************************/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */

/*************************************************************
 slave transfer complete
*************************************************************/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWI_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc.... */
R0 = W[P1 + LO(TWI_FIFO_STAT)](z);
```

```
CC = BITTST(R0,BITPOS(RCV_HALF));    /* BIT 2 indicates whether
there's a byte in the FIFO or not */
if !CC JUMP _TWI_ISR.END;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);  /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */

_TWI_ISR.END:RTI;
```

Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I^2C Bus Specification version 2.1* dated January 2000.

Unique Information for the ADSP-BF50x Processor

None.

17 CAN MODULE

This chapter describes the Controller Area Network (CAN) module. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples. Familiarity with the CAN standard is assumed. Refer to Version 2.0 of CAN Specification from Robert Bosch GmbH.

This chapter contains:

Overview

Key features of the CAN module are:

- Conforms to the CAN 2.0B (active) standard
- Supports both standard (11-bit) and extended (29-bit) identifiers
- Supports data rates of up to 1 Mbit/s

- 32 mailboxes (8 transmit, 8 receive, 16 configurable)
- Dedicated acceptance mask for each mailbox
- Data filtering (first 2 bytes) can be used for acceptance filtering (DeviceNet™ mode)
- Error status and warning registers
- Universal counter module
- Readable receive and transmit pin values

The CAN module is a low bit rate serial interface intended for use in applications where bit rates are typically up to 1 Mbit/s. The CAN protocol incorporates a data CRC check, message error tracking and fault node confinement as means to improve network reliability to the level required for control applications.

Interface Overview

The interface to the CAN bus is a simple two-wire line. See Figure 17-1 for a symbolic representation of the CAN transceiver interconnection, and Figure 17-2 for a block diagram. The Blackfin processor's CANTX output and CANRX input pins are connected to an external CAN transceiver's TX and RX pins (respectively). The CANTX and CANRX pins operate with TTL levels and are appropriate for operation with CAN bus transceivers according to ISO/DIS 11898.

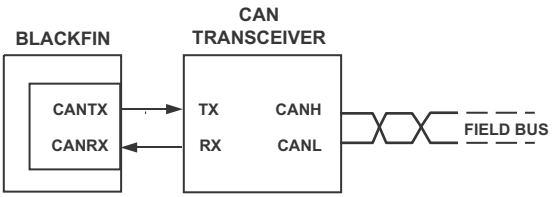


Figure 17-1. Representation of CAN Transceiver Interconnection

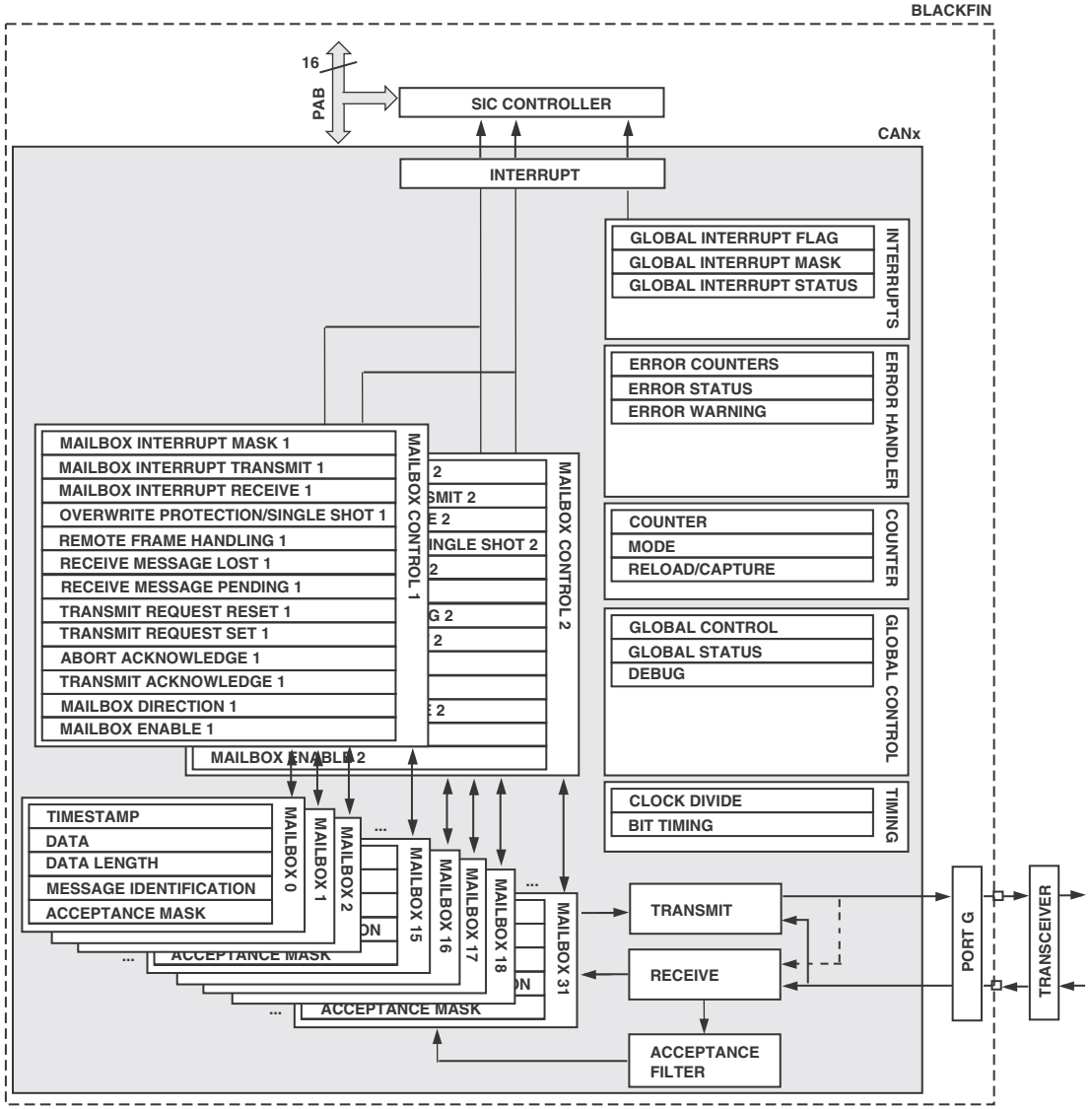


Figure 17-2. CAN Block Diagram

The CANRX and CANTX signals can be found on GPIO Port G, pins PG1 and PG2. CAN data is defined to be either *dominant* (logic 0) or *recessive* (logic 1). The default state of the CANTX output is recessive.

The PG1 pin (CANRX input pin) is also internally routed to the alternated capture input TACI5 of GP timer 5. This way, GP timer 5 can be used to auto-detect or adjust the bit rate on the CAN bus.

CAN Mailbox Area

The full-CAN controller features 32 message buffers, which are called mailboxes. Eight mailboxes are dedicated for message transmission, eight are for reception, and 16 are programmable in direction. Accordingly, the CAN module architecture is based around a 32-entry mailbox RAM. The mailbox is accessed sequentially by the CAN serial interface or the Blackfin core. Each mailbox consists of eight 16-bit control and data registers and two optional 16-bit acceptance mask registers, all of which must be configured before the mailbox itself is enabled. Since the mailbox area is implemented as RAM, the reset values of these registers are undefined. The data is divided into fields, which includes a message identifier, a time stamp, a byte count, up to 8 bytes of data, and several control bits. See Figure 17-3.

The CAN mailbox identification (CAN_MBxx_IDO/1) register pair includes:

- The 29 bit identifier (base part BASEID plus extended part EXTID_LO/HI)
- The acceptance mask enable bit (AME)
- The remote transmission request bit (RTR)
- The identifier extension bit (IDE)

Do not write to the identifier of a message object while the mailbox is enabled for the CAN module (the corresponding bit in CAN_MCx is set).

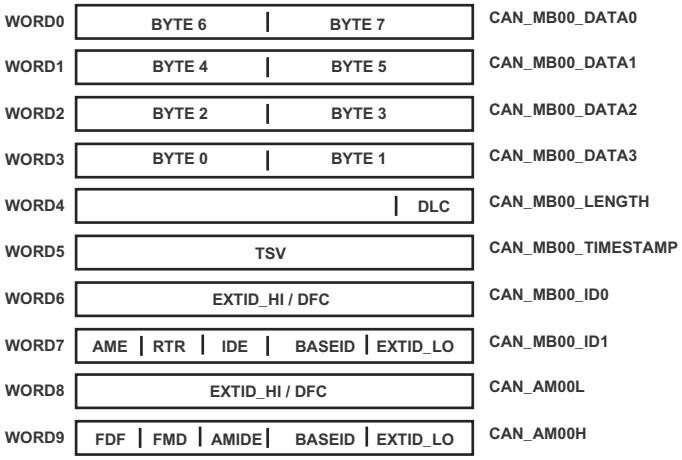


Figure 17-3. CAN Mailbox Area

The other mailbox area registers are:

- The data length code (DLC) in CAN_MBxx_LENGTH. The upper 12 bits of CAN_MBxx_LENGTH of each mailbox are marked as reserved. These 12 bits should always be set to 0. If DLC is programmed to a value greater than eight, the internal logic will set it to eight.
- Up to eight bytes for the data field, sent MSB first from the CAN_MBxx_DATA3/2/1/0 registers, respectively, based on the number of bytes defined in the DLC. For example, if only one byte is transmitted or received (DLC = 1), then it is stored in the most significant byte of the CAN_MBxx_DATA3 register.
- Two bytes for the time stamp value (TSV) in the CAN_MBxx_TIMESTAMP register

The final registers in the mailbox area are the acceptance mask registers (CAN_AMxxH and CAN_AMxxL). The acceptance mask is enabled when the AME bit is set in the CAN_MBxx_ID1 register. If the "filtering on data field" option is enabled (DNM = 1 in the CAN_CONTROL register and FDF = 1 in the corresponding acceptance mask), the EXTID_HI[15:0] bits of CAN_MBxx_IDO are reused as acceptance code (DFC) for the data field filtering. For more details, see "Receive Operation" on page 17-15 of this chapter.

CAN Mailbox Control

Mailbox control MMRs function as control and status registers for the 32 mailboxes. Each bit in these registers represents one specific mailbox. Since CAN MMRs are all 16 bits wide, pairs of registers are required to manage certain functionality for all 32 individual mailboxes. Mailboxes 0-15 are configured/monitored in registers with a suffix of 1. Similarly, mailboxes 16-31 use the same named register with a suffix of 2. For example, the CAN mailbox direction registers (CAN_MDx) would control mailboxes as shown in Figure 17-4.

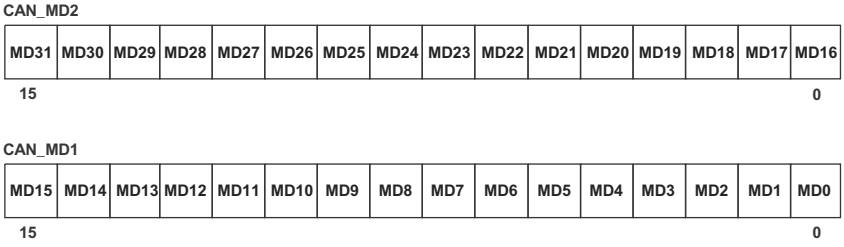


Figure 17-4. CAN Register Pairs

The mailbox control register area consists of these register pairs:

- CAN_MC1 and CAN_MC2 (mailbox enable registers)
- CAN_MD1 and CAN_MD2 (mailbox direction registers)

- CAN_TA1 and CAN_TA2 (transmit acknowledge registers)
- CAN_AA1 and CAN_AA2 (abort acknowledge registers)
- CAN_TRS1 and CAN_TRS2 (transmit request set registers)
- CAN_TRR1 and CAN_TRR2 (transmit request reset registers)
- CAN_RMP1 and CAN_RMP2 (receive message pending registers)
- CAN_RML1 and CAN_RML2 (receive message lost registers)
- CAN_RFH1 and CAN_RFH2 (remote frame handling registers)
- CAN_OPSS1 and CAN_OPSS2 (overwrite protection/single shot transmission registers)
- CAN_MBIM1 and CAN_MBIM2 (mailbox interrupt mask registers)
- CAN_MBTIF1 and CAN_MBTIF2 (mailbox transmit interrupt flag registers)
- CAN_MBRIF1 and CAN_MBRIF2 (mailbox receive interrupt flag registers)

Since mailboxes 24–31 support transmit operation only and mailboxes 0–7 are receive-only mailboxes, the lower eight bits in the "1" registers and the upper eight bits in the "2" registers are sometimes reserved or are restricted in their usage.

CAN Protocol Basics

Although the CANRX and CANTX pins are TTL-compliant signals, the CAN signals beyond the transceiver (see Figure 17-1) have asymmetric drivers. A low state on the CANTX pin activates strong drivers while a high state is driven weakly. Consequently, active low is called the "dominant" state and active high is called "recessive." If the CAN module is passive, the CANTX

pin is always high. If two CAN nodes transmit at the same time, dominant bits overwrite recessive bits.

The CAN protocol defines that all nodes trying to send a message on the CAN bus attempt to send a frame once the CAN bus becomes available. The start of frame indicator (SOF) signals the beginning of a new frame. Each CAN node then begins transmitting its message starting with the message ID. While transmitting, the CAN controller samples the CANRX pin to verify that the logic level being driven is the value it just placed on the CANTX pin. This is where the names for the logic levels apply. If a transmitting node places a recessive '1' on CANTX and detects a dominant '0' on the CANRX pin, it knows that another node has placed a dominant bit on the bus, which means another node has higher priority. So, if the value sensed on CANRX is the value driven on CANTX, transmission continues, otherwise the CAN controller senses that it has lost arbitration and configuration determines what the next course of action is once arbitration is lost. See Figure 17-5 for more details regarding CAN frame structure.

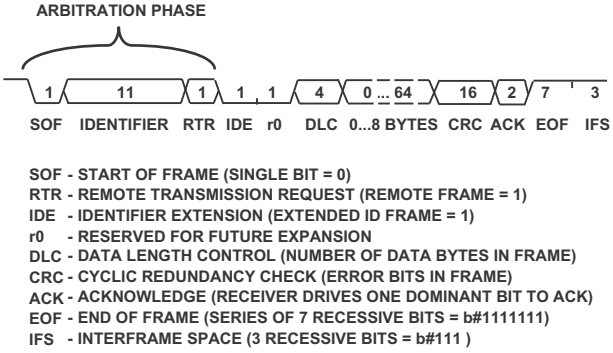


Figure 17-5. Standard CAN Frame

Figure 17-5 is a basic 11-bit identifier frame. After the SOF and identifier is the RTR bit, which indicates whether the frame contains data (data frame) or is a request for data associated with the message identifier in the frame being sent (remote frame).

Due to the inherent nature of the CAN protocol, a dominant bit in the RTR field wins arbitration against a remote frame request (RTR=1) for the same message ID, thereby defining a remote request to be lower priority than a data frame.

The next field of interest is the IDE. When set, it indicates that the message is an extended frame with a 29-bit identifier instead of an 11-bit identifier. In an extended frame, the first part of the message resembles Figure 17-6.

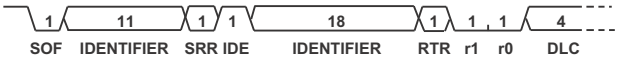


Figure 17-6. Extended CAN Frame

As could be concluded with regards to the RTR field, a dominant bit in the IDE field wins arbitration against an extended frame with the same lower 11-bits, therefore, standard frames are higher priority than extended frames. The substitute remote request bit (SRR, always sent as recessive), the reserved bits r0 and r1 (always sent as dominant), and the checksum (CRC) are generated automatically by the internal logic.

CAN Operation

The CAN controller is in configuration mode when coming out of processor reset or hibernate. It is only when the CAN is in configuration mode that hardware behavior can be altered. Before initializing the mailboxes themselves, the CAN bit timing must be set up to work on the CAN bus that the controller is expected to connect to.

Bit Timing

The CAN controller does not have a dedicated clock. Instead, the CAN clock is derived from the system clock (SCLK) based on a configurable number of time quanta. The Time Quantum (TQ) is derived from the formula TQ = (BRP+1)/SCLK, where BRP is the 10-bit BRP field in the CAN_CLOCK register. Although the BRP field can be set to any value, it is recommended that the value be greater than or equal to 4, as restrictions apply to the bit timing configuration when BRP is less than 4.

The CAN_CLOCK register defines the TQ value, and multiple time quanta make up the duration of a CAN bit on the bus. The CAN_TIMING register controls the nominal bit time and the sample point of the individual bits in the CAN protocol. Figure 17-7 shows the three phases of a CAN bit—the synchronization segment, the segment before the sample point, and the segment after the sample point.

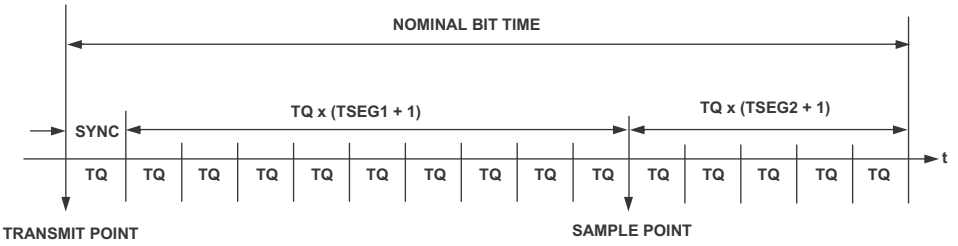


Figure 17-7. Three Phases of a CAN Bit

The synchronization segment is fixed to one TQ. It is required to synchronize the nodes on the bus. All signal edges are expected to occur within this segment.

The TSEG1 and TSEG2 fields of CAN_TIMING control how many TQs the CAN bits consist of, resulting in the CAN bit rate. The nominal bit time is given by the formula tBIT = TQ x (1 + (1 + TSEG1) + (1 + TSEG2)). For safe receive operation on given physical networks, the sample point is programmable by the TSEG1 field. The TSEG2 field holds the number of TQs

needed to complete the bit time. Often, best sample reliability is achieved with sample points in the high 80% range of the bit time. Never use sample points lower than 50%. Thus, TSEG1 should always be greater than or equal to TSEG2.

The Blackfin CAN module does not distinguish between the propagation segment and the phase segment 1 as defined by the standard. The TSEG1 value is intended to cover both of them. The TSEG2 value represents the phase segment 2.

If the CAN module detects a recessive-to-dominant edge outside the synchronization segment, it can automatically move the sampling point such that the CAN bit is still handled properly. The synchronization jump width (SJW) field specifies the maximum number of TQs, ranging from 1 to 4 (SJW + 1), allowed for such a re-synchronization attempt. The SJW value should not exceed TSEG2 or TSEG1. Therefore, the fundamental rule for writing CAN_TIMING is:

```
SJW <= TSEG2 <= TSEG1
```

In addition to this fundamental rule, phase segment 2 must also be greater than or equal to the Information Processing Time (IPT). This is the time required by the logic to sample CANRX input. On the Blackfin CAN module, this is 3 SCLK cycles. Because of this, restrictions apply to the minimal value of TSEG2 if the clock prescaler BRP is lower than 2. If BRP is set to 0, the TSEG2 field must be greater than or equal to 2. If the prescaler is set to 1, the minimum TSEG2 is 1.

All nodes on a CAN bus should use the same nominal bit rate.

With all the timing parameters set, the final consideration is how sampling is performed. The default behavior of the CAN controller is to sample the CAN bit once at the sampling point described by the CAN_TIMING register, controlled by the SAM bit. If the SAM bit is set, however, the input signal is oversampled three times at the SCLK rate. The

resulting value is generated by a majority decision of the three sample values. Always keep the SAM bit cleared if the BRP value is less than 4.

Do not modify the CAN_CLOCK or CAN_TIMING registers during normal operation. Always enter configuration mode first. Writes to these registers have no effect if not in configuration or debug mode. If not coming out or processor reset or hibernate, enter configuration mode by setting the CCR bit in the master control (CAN_CONTROL) register and poll the global CAN status (CAN_STATUS) register until the CCA bit is set.

If the TSEG1 field of the CAN_TIMING register is programmed to '0,' the module doesn't leave the configuration mode.

During configuration mode, the module is not active on the CAN bus line. The CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. After leaving the configuration mode, all CAN core internal registers and the CAN error counters are set to their initial values.

A software reset does not change the values of CAN_CLOCK and CAN_TIMING. Thus, an ongoing transfer via the CAN bus cannot be corrupted by changing the bit timing parameter or initiating the software reset (SRS = 1 in CAN_CONTROL).

Transmit Operation

Figure 17-8 shows the CAN transmit operation. Mailboxes 24-31 are dedicated transmitters. Mailboxes 8-23 can be configured as transmitters by writing 0 to the corresponding bit in the CAN_MDx register. After writing the data and the identifier into the mailbox area, the message is sent after mailbox n is enabled (MCn = 1 in CAN_MCx) and, subsequently, the corresponding transmit request bit is set (TRSn = 1 in CAN_TRSx).

When a transmission completes, the corresponding bits in the transmit request set register and in the transmit request reset register (TRRn in CAN_TRRx) are cleared. If transmission was successful, the corresponding

bit in the transmit acknowledge register (TAn in CAN_TAx) is set. If the transmission was aborted due to lost arbitration or a CAN error, the corresponding bit in the abort acknowledge register (AAn in CAN_AAx) is set. A requested transmission can also be manually aborted by setting the corresponding TRRn bit in CAN_TRRx.

Multiple CAN_TRSx bits can be set simultaneously by software, and these bits are reset after either a successful or an aborted transmission. The TRSn bits can also be set by the CAN hardware when using the auto-transmit mode of the universal counter, when a message loses arbitration and the single-shot bit is not set (OPSSn = 0 in CAN_OPSSx), or in the event of a remote frame request. The latter is only possible for receive/transmit mailboxes if the automatic remote frame handling feature is enabled (RFHn = 1 in CAN_RFHx).

Special care should be given to mailbox area management when a TRSn bit is set. Write access to the mailbox is permissible with TRSn set, but changing data in such a mailbox may lead to unexpected data during transmission.

Enabling and disabling mailboxes has an impact on transmit requests. Setting the TRSn bit associated with a disabled mailbox may result in erroneous behavior. Similarly, disabling a mailbox before the associated TRSn bit is reset by the internal logic can cause unpredictable results.

Retransmission

Normally, the current message object is sent again after arbitration is lost or an error frame is detected on the CAN bus line. If there is more than one transmit message object pending, the message object with the highest mailbox is sent first (see Figure 17-8). The currently aborted transmission is restarted after any messages with higher priority are sent.

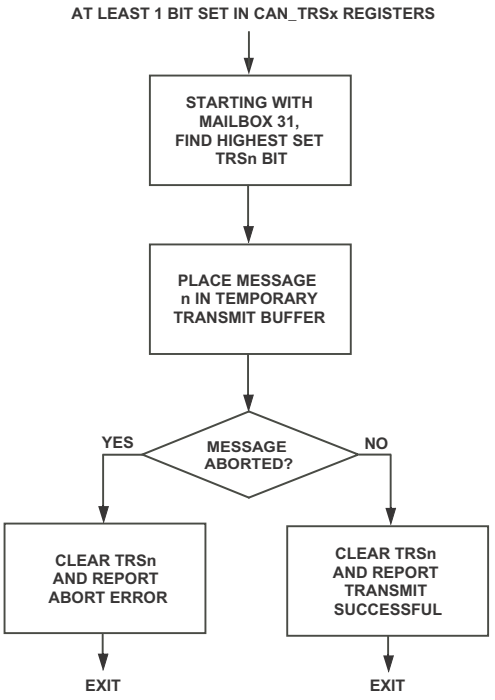


Figure 17-8. CAN Transmit Operation Flow Chart

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message in the buffer is not replaced until it is sent successfully, the arbitration on the CAN bus line is lost, or there is an error frame on the CAN bus line.

Single Shot Transmission

If the single shot transmission feature is used (`OPSSn = 1` in `CAN_OPSSx`), the corresponding `TRSn` bit is cleared after the message is successfully sent or if the transmission is aborted due to a lost arbitration or an error frame on

the CAN bus line. Thus, there is no further attempt to transmit the message again if the initial try failed, and the abort error is reported (AAn = 1 in CAN_AAx)

Auto-Transmission

In auto-transmit mode, the message in mailbox 11 can be sent periodically using the universal counter. This mode is often used to broadcast heartbeats to all CAN nodes. Accordingly, messages sent this way usually have high priority.

The period value is written to the CAN_UCRC register. When enabled in this mode (set UCCNF[3:0] = 0x3 in CAN_UCCNF), the counter (CAN_UCCNT) is loaded with the value in the CAN_UCRC register. The counter decrements at the CAN bit clock rate down to 0 and is then reloaded from CAN_UCRC. Each time the counter reaches a value of 0, the TRS11 bit is automatically set by internal logic, and the corresponding message from mailbox 11 is sent.

For proper auto-transmit operation, mailbox 11 must be configured as a transmit mailbox and must contain valid data (identifier, control bits, and data) before the counter first expires after this mode is enabled.

Receive Operation

The CAN hardware autonomously receives messages and discards invalid messages. Once a valid message has been successfully received, the receive logic interrogates all enabled receive mailboxes sequentially, from mailbox 23 down to mailbox 0, whether the message is of interest to the local node or not.

Each incoming data frame is compared to all identifiers stored in active receive mailboxes (MDn = 1 and MCn = 1) and to all active transmit mailboxes with the remote frame handling feature enabled (RFHn = 1 in CAN_RFHx).

The message identifier of the received message, along with the identifier extension (IDE) and remote transmission request (RTR) bits, are compared against each mailbox's register settings. If the AME bit is not set, a match is signalled only if IDE, RTR, and all identifier bits are exact. If, however, AME is set, the acceptance mask registers determine which of the identifier, IDE, and RTR bits need to match. The logic applies Received Message XNOR CAN_IDx or AME AND CAN_AMx. A one at the respective bit position in the CAN_AMxx mask registers means that the bit does not need to match when AME = 1. This way, a mailbox can accept a group of messages.

Table 17-1. Mailbox Used for Acceptance Mask Filtering

Mailbox Used for Acceptance Filtering				
MCn	**MDn**	**RFHn**	**Mailbox n**	**Comment**
0	x	x	Ignored	Mailbox n disabled
1	0	0	Ignored	Mailbox n enabled Mailbox n configured for transmit Remote frame handling disabled
1	0	1	Used	Mailbox n enabled Mailbox n configured for transmit Remote frame handling enabled
1	1	x	Used	Mailbox n enabled Mailbox n configured for receive

If the acceptance filter finds a matching identifier, the content of the received data frame is stored in that mailbox. A received message is stored only once, even if multiple receive mailboxes match its identifier. If the current identifier does not match any mailbox, the message is not stored.

Figure 17-9 illustrates the decision tree of the receive logic when processing the individual mailboxes.

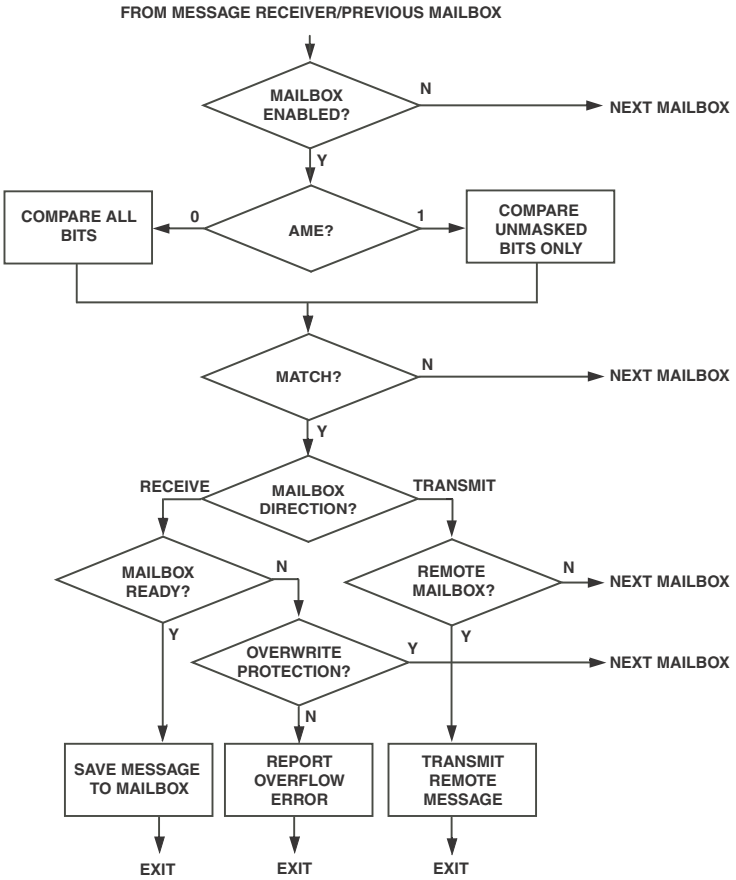


Figure 17-9. CAN Receive Operation Flow Chart

If a message is received for a mailbox and that mailbox still contains unread data (`RMPn = 1`), the user has to decide whether the old message should be overwritten or not. If `OPSSn = 0`, the receive message lost bit (`RMLn` in `CAN_RMLx`) is set and the stored message is overwritten. This

results in the receive message lost interrupt being raised in the global CAN interrupt status register (RMLIS = 1 in CAN_GIS). If OPSSn = 1, the next mailboxes are checked for another matching identifier. If no match is found, the message is discarded and the next message is checked.

If a receive mailbox is disabled, an ongoing receive message for that mailbox is lost even if a second mailbox is configured to receive the same identifier.

Data Acceptance Filter

If DeviceNet mode is enabled (DNM = 1 in CAN_CONTROL) and the mailbox is set up for filtering on data field, the filtering is done on the standard ID of the message and data fields. The data field filtering can be programmed for either the first byte only or the first two bytes, as shown in Table 17-2.

Table 17-2. Data Field Filtering

FDF Filter On Data Field	FMD Full Mask Data Field	Description
0	0	Do not allow filtering on the data field
0	1	Not allowed. FMD must be 0 if FDF is 0.
1	0	Filter on first data byte only
1	1	Filter on first two data bytes

If the FDF bit is set in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the data field mask (DFM[15:0]). If the FDF bit is cleared in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the extended identifier mask (EXTID_HI[15:0]).

Remote Frame Handling

Automatic handling of remote frames can be enabled/disabled by setting/clearing the corresponding bit in the remote frame handling registers (CAN_RFHx) of a transmit mailbox.

Remote frames are data frames with no data field and the RTR bit set. The data length code of the responding data frame is overruled by the DLC of the requesting remote frame. A data length code can be programmed with values in the range of 0 to 15, but data length code values greater than 8 are considered as 8. A remote frame contains:

- the identifier bits
- the control field DLC
- the remote transmission request (RTR) bit

Only configurable mailboxes 8–23 can process remote frames, but all mailboxes can receive and transmit remote frame requests. When setup for automatic remote frame handling, the CAN_OPSSx register has no effect. All content of a mailbox is always overwritten by an incoming message.

If a remote frame is received, the DLC of the corresponding mailbox is overwritten with the received value.

Erroneous behavior may result when the remote frame handling bit (RFHn) is changed and the corresponding mailbox is currently processed. See "Temporarily Disabling Mailboxes" on page 17-21 for safe mailbox handling.

Watchdog Mode

Watchdog mode is used to make sure messages are received periodically. It is often used to observe whether or not a certain node on the network is alive and functioning properly, and, if not, to detect and manage its failure case accordingly.

Upon programming the universal counter to watchdog mode (set UCCNF[3:0] = 0x2 in CAN_UCCNF), the counter in the CAN_UCCNT register is loaded with the predefined value contained in the CAN universal counter reload/capture register (CAN_UCRC). This counter then decrements at the CAN bit rate. If the UCCT and UCRC bits in the CAN_UCCNF register are set and a message is received in mailbox 4 before the counter counts down to 0, the counter is reloaded with the CAN_UCRC contents. If the counter has counted down to 0 without receiving a message in mailbox 4, the UCEIS bit in the global CAN interrupt status (CAN_GIS) register is set, and the counter is automatically reloaded with the contents of the CAN_UCRC register. If an interrupt is desired, the UCEIM bit in the CAN_GIM register must also be set. With the mask bit set, when a watchdog interrupt occurs, the UCEIF bit in the CAN_GIF register is also set.

The counter can be reloaded with the contents of CAN_UCRC or disabled by writing to the CAN_UCCNF register.

The time period it takes for the watchdog interrupt to occur is controlled by the value written into the CAN_UCRC register by the user.

Time Stamps

To get an indication of the time of reception or the time of transmission for each message, program the CAN universal counter to time stamp mode (set UCCNF[3:0] = 0x1 in CAN_UCCNF). The value of the 16-bit free-running counter (CAN_UCCNT) is then written into the CAN_MBxx_TIMESTAMP register of the corresponding mailbox when a received message has been stored or a message has been transmitted.

The time stamp value is captured at the sample point of the start of frame (SOF) bit of each incoming or outgoing message. Afterwards, this time stamp value is copied to the CAN_MBxx_TIMESTAMP register of the corresponding mailbox.

If the mailbox is configured for automatic remote frame handling, the time stamp value is written for transmission of a data frame (mailbox

configured as transmit) or the reception of the requested data frame (mailbox configured as receive).

The counter can be cleared (set UCRC bit to 1) or disabled (set UCE bit to 0) by writing to the CAN_UCCNF register. The counter can also be loaded with a value by writing to the counter register itself (CAN_UCCNT).

It is also possible to clear the counter (CAN_UCCNT) by reception of a message in mailbox number 4 (synchronization of all time stamp counters in the system). This is accomplished by setting the UCCT bit in the CAN_UCCNF register.

An overflow of the counter sets a bit in the global CAN interrupt status register (UCEIS in the CAN_GIS register). A global CAN interrupt can optionally occur by unmasking the bit in the global CAN interrupt mask register (UCEIM in the CAN_GIM register). If the interrupt source is unmasked, a bit in the global CAN interrupt flag register is also set (UCEIF in the CAN_GIF register).

Temporarily Disabling Mailboxes

If this mailbox is used for automatic remote frame handling, the data field must be updated without losing an incoming remote request frame and without sending inconsistent data. Therefore, the CAN controller allows for temporary mailbox disabling, which can be enabled by programming the mailbox temporary disable register (CAN_MBTD).

The pointer to the requested mailbox must be written to the TDPTR[4:0] bits of the CAN_MBTD register and the mailbox temporary disable request bit (TDR) must be set. The corresponding mailbox temporary disable flag (TDA) is subsequently set by the internal logic.

If a mailbox is configured as “transmit” (MDn = 0) and TDA is set, the content of the data field of that mailbox can be updated. If there is an incoming remote request frame while the mailbox is temporarily disabled, the corresponding transmit request set bit (TRSn) is set by the internal

logic and the data length code of the incoming message is written to the corresponding mailbox. However, the message being requested is not sent until the temporary disable request is cleared (TDR = 0). Similarly, all transmit requests for temporarily disabled mailboxes are ignored until TDR is cleared. Additionally, transmission of a message is immediately aborted if the mailbox is temporarily disabled and the corresponding TRRn bit for this mailbox is set.

If a mailbox is configured as "receive" (MDn = 1), the temporary disable flag is set and the mailbox is not processed. If there is an incoming message for the mailbox n being temporarily disabled, the internal logic waits until the reception is complete or there is an error on the CAN bus to set TDA. Once TDA is set, the mailbox can then be completely disabled (MCn = 0) without the risk of losing an incoming frame. The temporary disable request (TDR) bit must then be reset as soon as possible.

When TDA is set for a given mailbox, only the data field of that mailbox can be updated. Accesses to the control bits and the identifier are denied.

Functional Operation

The following sections describe the functional operation of the CAN module, including interrupts, the event counter, warnings and errors, debug features, and low power features.

CAN Interrupts

The CAN module provides three independent interrupts: two mailbox interrupts (mailbox receive interrupt MBRIRQ and mailbox transmit interrupt MBTIRQ) and the global CAN interrupt GIRQ. The values of these three interrupts can also be read back in the interrupt status registers.

Mailbox Interrupts

Each of the 32 mailboxes in the CAN module may generate a receive or transmit interrupt, depending on the mailbox configuration. To enable a mailbox to generate an interrupt, set the corresponding MBIMn bit in CAN_MBIMx.

If a mailbox is configured as a receive mailbox, the corresponding receive interrupt flag is set (MBRIFn = 1 in CAN_MBRIFx) after a received message is stored in mailbox n (RMPn = 1 in CAN_RMPx). If the automatic remote frame handling feature is used, the receive interrupt flag is set after the requested data frame is stored in the mailbox. If any MBRIFn bits are set in CAN_MBRIFx, the MBRIRQ interrupt output is raised in CAN_INTR. In order to clear the MBRIRQ interrupt request, all of the set MBRIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBRIFx.

If a mailbox is configured as a transmit mailbox, the corresponding transmit interrupt flag is set (MBTIFn = 1 in CAN_MBTIFx) after the message in mailbox n is sent correctly (TAn = 1 in CAN_TAx). The TAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0). If the automatic remote frame handling feature is used, the transmit interrupt flag is set after the requested data frame is sent from the mailbox. If any MBTIFn bits are set in CAN_MBTIFx, the MBTIRQ interrupt output is raised in CAN_INTR. In order to clear the MBTIRQ interrupt request, all of the set MBTIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBTIFx.

Global CAN Status Interrupt

The global CAN status interrupt logic is implemented with three registers—the global CAN interrupt mask register (CAN_GIM), where each interrupt source can be enabled or disabled separately; the global CAN interrupt status register (CAN_GIS); and the global CAN interrupt flag register (CAN_GIF). The interrupt mask bits only affect the content of the global CAN interrupt flag register (CAN_GIF). If the mask bit is not set, the corresponding flag bit is not set when the event occurs. The interrupt

status bits in the global CAN interrupt status register, however, are always set if the corresponding interrupt event occurs, independent of the mask bits. Thus, the interrupt status bits can be used for polling of interrupt events.

The global CAN status interrupt output (GIRQ) bit in the global CAN interrupt status register is only asserted if a bit in the CAN_GIF register is set. The GIRQ bit remains set as long as at least one bit in the interrupt flag register CAN_GIF is set. All bits in the interrupt status and in the interrupt flag registers remain set until cleared by software or a software reset has occurred.

There are several interrupt events that can activate this GIRQ interrupt:

- **Access denied interrupt** (ADIM, ADIS, ADIF)

  At least one access to the mailbox RAM occurred during a data update by internal logic.

- **External trigger output interrupt** (EXTIM, EXTIS, EXTIF)

  The external trigger event occurred.

- **Universal counter exceeded interrupt** (UCEIM, UCEIS, UCEIF)

  There was an overflow of the universal counter (in time stamp mode or event counter mode) or the counter has reached the value 0x0000 (in watchdog mode).

- **Receive message lost interrupt** (RMLIM, RMLIS, RMLIF)

  A message has been received for a mailbox that currently contains unread data. At least one bit in the receive message lost register (CAN_RMLx) is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_RMLx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_RMLx is set.

- **Abort acknowledge interrupt** (AAIM, AAIS, AAIF)

  At least one AAn bit in the abort acknowledge registers CAN_AAx is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_AAx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_AAx is set. The AAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0).

- **Access to unimplemented address interrupt** (UIAIM, UIAIS, UIAIF)

  There was a CPU access to an address which is not implemented in the controller module.

- **Wakeup interrupt** (WUIM, WUIS, WUIF)

  The CAN module has left the sleep mode because of detected activity on the CAN bus line.

- **Bus-Off interrupt** (BOIM, BOIS, BOIF)

  The CAN module has entered the bus-off state. This interrupt source is active if the status of the CAN core changes from normal operation mode to the bus-off mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the bus-off mode is still active, this bit is not set again. If the module leaves the bus-off mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error-Passive interrupt** (EPIM, EPIS, EPIF)

  The CAN module has entered the error-passive state. This interrupt source is active if the status of the CAN module changes from the error-active mode to the error-passive mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the error-passive mode is still active, this bit is not set again. If the module leaves the error-passive mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning receive interrupt** (EWRIM, EWRIS, EWRIF)

The CAN receive error counter (RXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning transmit interrupt** (EWTIM, EWTIS, EWTIF)

  The CAN transmit error counter (TXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

Event Counter

For diagnostic functions, it is possible to use the universal counter as an event counter. The counter can be programmed in the 4-bit UCCNF[3:0] field of CAN_UCCNF to increment on one of these conditions:

- UCCNF[3:0] = 0x6 – CAN error frame. Counter is incremented if there is an error frame on the CAN bus line.
- UCCNF[3:0] = 0x7 – CAN overload frame. Counter is incremented if there is an overload frame on the CAN bus line.
- UCCNF[3:0] = 0x8 – Lost arbitration. Counter is incremented every time arbitration on the CAN line is lost during transmission.
- UCCNF[3:0] = 0x9 – Transmission aborted. Counter is incremented every time arbitration is lost or a transmit request is cancelled (AAn is set).
- UCCNF[3:0] = 0xA – Transmission succeeded. Counter is incremented every time a message sends without detected errors (TAn is set).

- UCCNF[3:0] = 0xB – Receive message rejected. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because there is no matching identifier found.
- UCCNF[3:0] = 0xC – Receive message lost. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because the mailbox contains unread data (RMLn is set).
- UCCNF[3:0] = 0xD – Message received. Counter is incremented every time a message is received without detected errors, whether the received message is rejected or stored in a mailbox.
- UCCNF[3:0] = 0xE – Message stored. Counter is incremented every time a message is received without detected errors, has an identifier that matches an enabled receive mailbox, and is stored in the receive mailbox (RMPn is set).
- UCCNF[3:0] = 0xF – Valid message. Counter is incremented every time a valid transmit or receive message is detected on the CAN bus line.

CAN Warnings and Errors

CAN warnings and errors are controlled using the CAN_CEC register, the CAN_ESR register, and the CAN_EWR register.

Programmable Warning Limits

It is possible to program the warning level for EWTIS (error warning transmit interrupt status) and EWRIS (error warning receive interrupt status) separately by writing to the error warning level error count fields for receive (EWLREC) and transmit (EWLTEC) in the CAN error counter warning level (CAN_EWR) register. After powerup reset, the CAN_EWR register is set to the default warning level of 96 for both error counters. After software reset, the content of this register remains unchanged.

CAN Error Handling

Error management is an integral part of the CAN standard. Five different kinds of bus errors may occur during transmissions:

- **Bit error**

  A bit error can be detected by the transmitting node only. Whenever a node is transmitting, it continuously monitors its receive pin (CANRX) and compares the received data with the transmitted data. During the arbitration phase, the node simply postpones the transmission if the received and transmitted data do not match. However, after the arbitration phase (that is, once the RTR bit has been sent successfully), a bit error is signaled any time the value on CANRX does not equal what is being transmitted on CANTX.

- **Form error**

  A form error occurs any time a fixed-form bit position in the CAN frame contains one or more illegal bits, that is, when a dominant bit is detected at a delimiter or end-of-frame bit position.

- **Acknowledge error**

  An acknowledge error occurs whenever a message has been sent and no receivers drive an acknowledge bit.

- **CRC error**

  A CRC error occurs whenever a receiver calculates the CRC on the data it received and finds it different than the CRC that was transmitted on the bus itself.

- **Stuff error**

  The CAN specification requires the transmitter to insert an extra stuff bit of opposite value after 5 bits have been transmitted with the same value. The receiver disregards the value of these stuff bits. However, it takes advantage of the signal edge to re-synchronize itself. A stuff error occurs on receiving nodes whenever the 6th consecutive bit value is the same as the previous five bits.

Once the CAN module detects any of the above errors, it updates the error status register `CAN_ESR` as well as the error counter register `CAN_CEC`. In addition to the standard errors, the `CAN_ESR` register features a flag that signals when the `CANRX` pin sticks at dominant level, indicating that shorted wires are likely.

Error Frames

It is of central importance that all nodes on the CAN bus ignore data frames that one single node failed to receive. To accomplish this, every node sends an error frame as soon as it has detected an error. See Figure 17-10.

Once a device has detected an error, it still completes the ongoing bit and initiates an error frame by sending six dominant and eight recessive bits to the bus. This is a violation to the bit stuffing rule and informs all nodes that the ongoing frame needs to be discarded.

All receivers that did not detect the transmission error in the first instance now detect a stuff bit error. The transmitter may detect a normal bit error sooner. It aborts the transmission of the ongoing frame and tries sending it again later.

Finally, all nodes on the bus have detected an error. Consequently, all of them send 6 dominant and 8 recessive bits to the bus as well. The resulting error frame consists of two different fields. The first field is given by the superposition of error flags contributed from the different stations, which is a sequence of 6 to 12 dominant bits. The second field is the error delimiter and consists of 8 recessive bits indicating the end of frame.

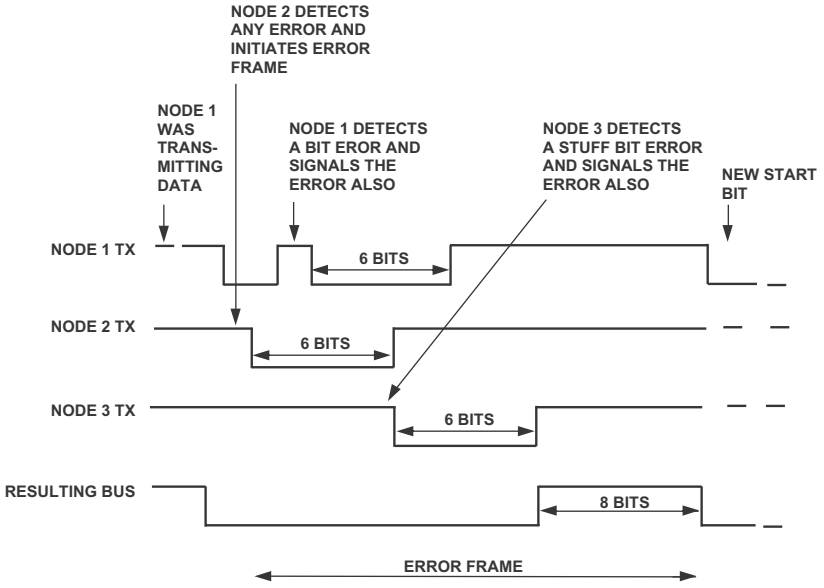


Figure 17-10. CAN Error Scenario Example

For CRC errors, the error frame is initiated at the end of the frame, rather than immediately after the failing bit.

After having received 8 recessive bits, every node knows that the error condition has been resolved and starts transmission if messages are pending. The former transmitter that had to abort its operation must win the new arbitration again, otherwise its message is delayed as determined by priority.

Because the transmission of an error frame destroys the frame under transmission, a faulty node erroneously detecting an error can block the bus. Because of this, there are two node states which determine a node's right to signal an error—error active and error passive. Error active nodes are those which have an error detection rate below a certain limit. These nodes drive an 'active error flag' of 6 dominant bits.

Nodes with a higher error detection rate are suspected of having a local problem and, therefore, have a limited right to signal errors. These error passive nodes drive a 'passive error flag' consisting of 6 recessive bits. Thus, an error passive transmitting node is still able to inform the other nodes about the abortion of a self-transmitted frame, but it is no longer able to destroy correctly received frames of other nodes.

Error Levels

The CAN specification requires each node in the system to operate in one of three levels. See Table 17-3. This prevents nodes with high error rates from blocking the entire network, as the errors might be caused by local hardware. The Blackfin CAN module provides an error counter for transmit (TEC) and an error counter for receive (REC). The CAN error count register CAN_CEC houses each of these 8-bit counters.

After initialization, both the TEC and the REC counters are 0. Each time a bus error occurs, one of the counters is incremented by either 1 or 8, depending on the error situation (documented in Version 2.0 of *CAN Specification*). Successful transmit and receive operations decrement the respective counter by 1.

If either of the error counters exceeds 127, the CAN module goes into a passive state and the CAN error passive mode (EP) bit in CAN_STATUS is set. Then, it is not allowed to send any more active error frames. However, it is still allowed to transmit messages and to signal passive error frames in case the transmission fails because of a bit error.

If one of the counters exceeds 255 (that is, when the 8-bit counters overflow), the CAN module is disconnected from the bus. It goes into bus off mode and the CAN error bus off mode (EBO) bit is set in CAN_STATUS. Software intervention is required to recover from this state.

Table 17-3. CAN Error Level Description

Level	Condition	Description
Error Active	Transmit and receive error counters < 128	This is the initial condition level. As long as errors stay below 128, the node will drive active error flags during error frames.
Error Passive	Transmit or receive error counters ≥ 128, but < 256	Errors have accumulated to a level which requires the node to drive passive error flags during error frames.
Bus Off	Transmit or receive error counters ≥ 256	CAN module goes into bus off mode

In addition to these levels, the CAN module also provides a warning mechanism, which is an enhancement to the CAN specification. There are separate warnings for transmit and receive. By default, when one of the error counters exceeds 96, a warning is signaled and is represented in the CAN_STATUS register by either the CAN receive warning flag (WR) or CAN transmit warning flag (WT) bits. The error warning level can be programmed using the error warning register, CAN_EWR. More information is available on page 17-84.

Additionally, interrupts can occur for all of these levels by unmasking them in the global CAN interrupt mask register (CAN_GIM) shown on page 17-47. The interrupts include the bus off interrupt (BOIM), the error-passive interrupt (EPIM), the error warning receive interrupt (EWRIM), and the error warning transmit interrupt (EWTIM).

During the bus off recovery sequence, the configuration mode request bit in the CAN_CONTROL register is set by the internal logic (CCR = 1), thus the CAN core module does not automatically come out of the bus off mode. The CCR bit cannot be reset until the bus off recovery sequence is finished.

This behavior can be over-ridden by setting the auto-bus on (ABO) bit in the CAN_CONTROL register. After exiting the bus off or configuration modes, the CAN error counters are reset.

Debug and Test Modes

The CAN module contains test mode features that aid in the debugging of the CAN software and system. Listing 17-1 provides an example of enabling CAN debug features.

When these features are used, the CAN module may not be compliant to the CAN specification. All test modes should be enabled or disabled only when the module is in configuration mode (CCA = 1 in the CAN_STATUS register) or in suspend mode (CSA = 1 in CAN_STATUS).

The CDE bit is used to gain access to all of the debug features. This bit must be set to enable the test mode, and must be written first before subsequent writes to the CAN_DEBUG register. When the CDE bit is cleared, all debug features are disabled.

Listing 17-1. Enabling CAN Debug Features in C

```
#include <cdefBF537.h>
/* Enable debug mode, CDE must be set before other flags can be
changed in register */
*pCAN_DEBUG |= CDE ;
```

```
/* Set debug flags */
*pCAN_DEBUG &= ~DTO ;
*pCAN_DEBUG |= MRB | MAA | DIL ;

/* Run test code */

/* Disable debug mode */
*pCAN_DEBUG &= ~CDE ;
```

When the CDE bit is set, it enables writes to the other bits of the CAN_DEBUG register. It also enables these features, which are not compliant with the CAN standard:

- Bit timing registers can be changed anytime, not only during configuration mode. This includes the CAN_CLOCK and CAN_TIMING registers.
- Allows write access to the read-only transmit/receive error counter register CAN_CEC.

The mode read back bit (MRB) is used to enable the read back mode. In this mode, a message transmitted on the CAN bus (or via an internal loop back mode) is received back directly to the internal receive buffer. After a correct transmission, the internal logic treats this as a normal receive message. This feature allows the user to test most of the CAN features without an external device.

The mode auto acknowledge bit (MAA) allows the CAN module to generate its own acknowledge during the ACK slot of the CAN frame. No external devices or connections are necessary to read back a transmit message. In this mode, the message that is sent is automatically stored in the internal receive buffer. In auto acknowledge mode, the module itself transmits the acknowledge. This acknowledge can be programmed to appear on the CANTX pin if DIL=1 and DTO=0. If the acknowledge is only going to be used internally, then these test mode bits should be set to DIL=0 and DTO=1.

The disable internal loop bit (DIL) is used to internally enable the transmit output to be routed back to the receive input.

The disable transmit output bit (DTO) is used to disable the CANTX output pin. When this bit is set, the CANTX pin continuously drives recessive bits.

The disable receive input bit (DRI) is used to disable the CANRX input. When set, the internal logic receives recessive bits or receives the internally generated transmit value in the case of the internal loop enabled (DIL=0). In either case, the value on the CANRX input pin is ignored.

The disable error counters bit (DEC) is used to disable the transmit and receive error counters in the CAN_CEC register. When this bit is set, the CAN_CEC holds its current contents and is not allowed to increment or decrement the error counters. This mode does not conform to the CAN specification.

Writes to the error counters should be in debug mode only. Write access during reception may lead to undefined values. The maximum value which can be written into the error counters is 255. Thus, the error counter value of 256 which forces the module into the bus off state can not be written into the error counters.

Table 17-4 shows several common combinations of test mode bits.

Table 17-4. CAN Test Modes

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
X	X	X	X	X	0	Normal mode, not debug mode.
0	X	X	X	X	X	No read back of transmit message.
1	0	1	0	0	1	Normal transmission on CAN bus line. Read back. External acknowledge from external device required.

Table 17-4. CAN Test Modes (Cont'd)

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
1	1	1	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is enabled.
1	1	0	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input and internal loop are enabled (internal OR of TX and RX).
1	1	0	0	1	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is ignored. Internal loop is enabled
1	1	0	1	1	1	No transmission on CAN bus line. Read back. No external acknowledge required. Neither transmit message nor acknowledge are transmitted on CANTX. CANRX input is ignored. Internal loop is enabled.

Low Power Features

The Blackfin processor provides a low power hibernate state, and the CAN module includes built-in sleep and suspend modes to save power. The behavior of the CAN module in these three modes is described in the following sections.

CAN Built-In Suspend Mode

The most modest of power savings modes is the suspend mode. This mode is entered by setting the suspend mode request (CSR) bit in the CAN_CONTROL register. The module enters the suspend mode after the current operation of the CAN bus is finished, at which point the internal logic sets the suspend mode acknowledge (CSA) bit in CAN_STATUS.

If the suspend mode is requested during the bus off recovery sequence, the module stops after the bus-off recovery sequence has completed. The module does not enter the suspend mode and the CSA bit is not set. Software must manually clear the CSR bit to restart the module.

Once this mode is entered, the module is no longer active on the CAN bus line, slightly reducing power consumption. When the CAN module is in suspend mode, the CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. The content of the CAN error counters remains unchanged.

The suspend mode can subsequently be exited by clearing the CSR bit in CAN_CONTROL. The only differences between suspend mode and configuration mode are that writes to the CAN_CLOCK and CAN_TIMING registers are still locked in suspend mode and the CAN control and status registers are not reset when exiting suspend mode.

CAN Built-In Sleep Mode

The next level of power savings can be realized by using the CAN module's built-in sleep mode. This mode is entered by setting the sleep mode request (SMR) bit in the CAN_CONTROL register. The module enters the sleep mode after the current operation of the CAN bus is finished. Once this mode is entered, many of the internal CAN module clocks are shut off, reducing power consumption, and the sleep mode acknowledge (SMACK) bit is set in CAN_INTR. When the CAN module is in sleep mode, all register reads return the contents of CAN_INTR instead of the usual contents. All register writes, except to CAN_INTR, are ignored in sleep mode.

A small part of the module is clocked continuously to allow for wakeup out of sleep mode. A write to the CAN_INTR register ends sleep mode. If the WBA bit in the CAN_CONTROL register is set before entering sleep mode, a dominant bit on the CANRX pin also ends sleep mode.

CAN Wakeup From Hibernate State

For greatest power savings, the Blackfin processor provides a hibernate state, where the internal voltage regulator shuts off the internal power supply to the chip, turning off the core and system clocks in the process. In this mode, the only power drawn (roughly 50 µA) is that used by the regulator circuitry awaiting any of the possible hibernate wakeup events. One such event is a wakeup due to CAN bus activity. After hibernation, the CAN module must be re-initialized.

For low power designs, the external CAN bus transceiver is typically put into standby mode via one of the Blackfin processor's general purpose I/O pins. While in standby mode, the CAN transceiver continually drives the recessive logic '1' level onto the CANRX pin. If the transceiver then senses CAN bus activity, it will, in turn, drive the CANRX pin to the dominant logic '0' level. This signals to the Blackfin processor that CAN bus activity has been detected. If the internal voltage regulator is programmed to recognize CAN bus activity as an event to exit hibernate state, the part

responds appropriately. Otherwise, the activity on the CANRX pin has no effect on the processor state.

To enable this functionality, the voltage control register (VR_CTL) must be programmed with the CAN wakeup enable bit set. The typical sequence of events to use the CAN wakeup feature is:

1. Use a general-purpose I/O pin to put the external transceiver into standby mode.
2. Program VR_CTL with the CAN wakeup enable bit (CANWE) set and the HIBERNATEB bit set to 0.

CAN Register Definitions

The following sections describe the CAN register definitions.

- "Global CAN Registers" on page 17-43
- "Mailbox/Mask Registers" on page 17-48
- "Mailbox Control Registers" on page 17-68
- "Universal Counter Registers" on page 17-82
- "Error Registers" on page 17-84

Table 17-5 through Table 17-9 show the functions of the CAN registers.

Table 17-5. Global CAN Register Mapping

Register Name	Function	Notes
CAN_CONTROL	Master control register	Reserved bits 15:8 and 3 must always be written as '0'
CAN_STATUS	Global CAN status register	Write accesses have no effect
CAN_DEBUG	CAN debug register	Use of these modes is not CAN-compliant

Table 17-5. Global CAN Register Mapping (Cont'd)

Register Name	Function	Notes
CAN_CLOCK	CAN clock register	Accessible only in configuration mode
CAN_TIMING	CAN timing register	Accessible only in configuration mode
CAN_INTR	CAN interrupt register	Reserved bits 15:8 and 5:4 must always be written as '0'
CAN_GIM	Global CAN interrupt mask register	Bits 15:11 are reserved
CAN_GIS	Global CAN interrupt status register	Bits 15:11 are reserved
CAN_GIF	Global CAN interrupt flag register	Bits 15:11 are reserved

Table 17-6. CAN Mailbox/Mask Register Mapping

Register Name	Function	Notes
CAN_AMxxH/L	Acceptance mask registers	Change only when mailbox MBxx is disabled
CAN_MBxx_ID1/0	Mailbox word 7/6 register	Do not write when MBxx is enabled
CAN_MBxx_TIMESTAMP	Mailbox word 5 register	Holds timestamp information when timestamp mode is active
CAN_MBxx_LENGTH	Mailbox word 4 register	Values greater than 8 are treated as 8
CAN_MBxx_DATA3/2/1/0	Mailbox word 3/2/1/0 register	Software controls reading correct data based on DLC

Table 17-7. CAN Mailbox Control Register Mapping

Register Name	Function	Notes
CAN_MCx	Mailbox configuration registers	Always disable before modifying mailbox area or direction
CAN_MDx	Mailbox direction registers	Never change MDn direction when mailbox n is enabled. MD[31:24] and MD[7:0] are read only
CAN_RMPx	Receive message pending registers	Clearing RMPn bits also clears corresponding RMLn bits
CAN_RMLx	Receive message lost registers	Write accesses have no effect
CAN_OPSSx	Overwrite protection or single-shot transmission register	Function depends on mailbox direction. Has no effect when RFHn = 1. Do not modify OPSSn bit if mailbox n is enabled
CAN_TRSx	Transmission request set registers	May by set by internal logic under certain circumstances. TRS[7:0] are read-only
CAN_TRRx	Transmission request reset registers	TRRn bits must not be set if mailbox n is disabled or TRSn = 0
CAN_AAx	Abort acknowledge registers	AAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_TAx	Transmission acknowledge registers	TAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_MBTD	Temporary mailbox disable feature register	Allows safe access to data field of an enabled mailbox
CAN_RFHx	Remote frame handling registers	Available only to configurable mailboxes 23:8. RFH[31:24] and RFH[7:0] are read-only
CAN_MBIMx	Mailbox interrupt mask registers	Mailbox interrupts are raised only if these bits are set

Table 17-7. CAN Mailbox Control Register Mapping (Cont'd)

Register Name	Function	Notes
CAN_MBTIFx	Mailbox transmit interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBTIFn = 1 results in MBRIFn = 1 and MBTIFn = 0
CAN_MBRIFx	Mailbox receive interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBRIFn = 1 results in MBTIFn = 1 and MBRIFn = 0

Table 17-8. CAN Universal Counter Register Mapping

Register Name	Function	Notes
CAN_UCCNF	Universal counter mode register	Bits 15:8 and bit 4 are reserved
CAN_UCCNT	Universal counter register	Counts up or down based on universal counter mode
CAN_UCRC	Universal counter reload/capture register	In timestamp mode, holds time of last successful transmit or receive

Table 17-9. CAN Error Register Mapping

Register Name	Function	Notes
CAN_CEC	CAN error counter register	Undefined while in bus off mode, not affected by software reset
CAN_ESR	Error status register	Only the first error is stored. SA0 flag is cleared by recessive bit on CAN bus
CAN_EWR	CAN error counter warning level register	Default is 96 for each counter

Global CAN Registers

Figure 17-11 through Figure 17-19 show the global CAN registers.

CAN_CONTROL Register

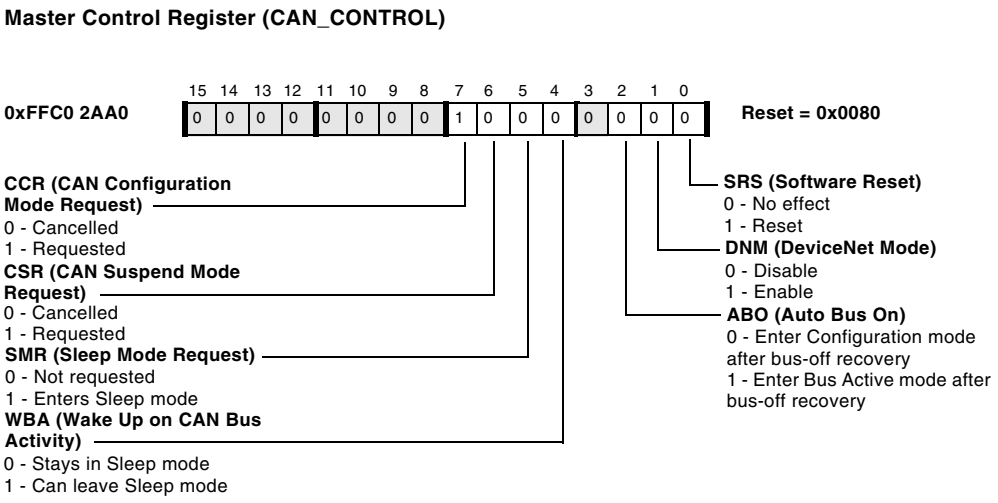


Figure 17-11. Master Control Register

CAN_STATUS Register

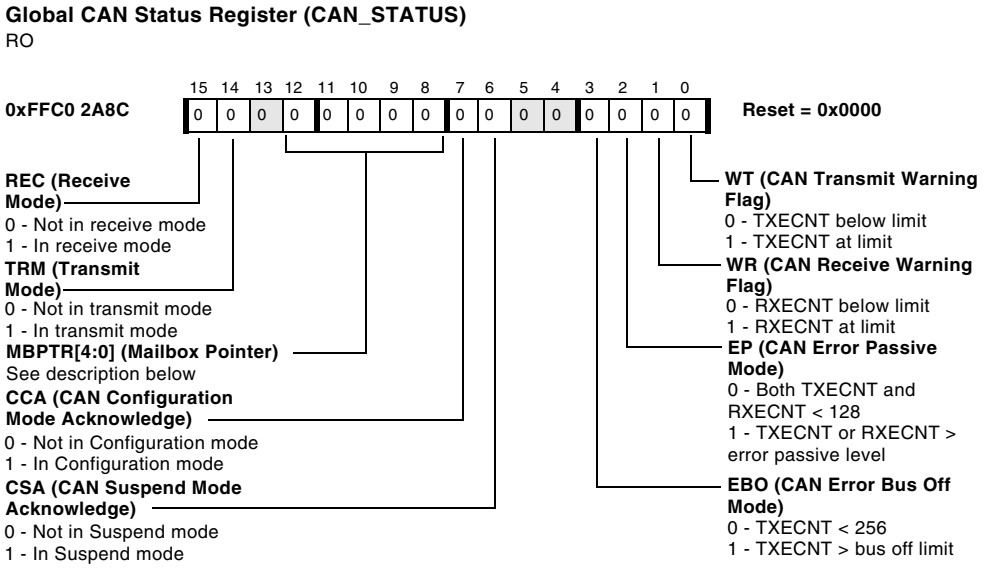


Figure 17-12. Global CAN Status Register

- **Mail box pointer** (`MBPTR[4:0]`)

  Represents the mailbox number of the current transmit message. After a successful transmission, these bits remain unchanged.

  [11111] The message of mailbox 31 is currently being processed.

  …

  …

  …

  [00000] The message of mailbox 0 is currently being processed.

CAN_DEBUG Register

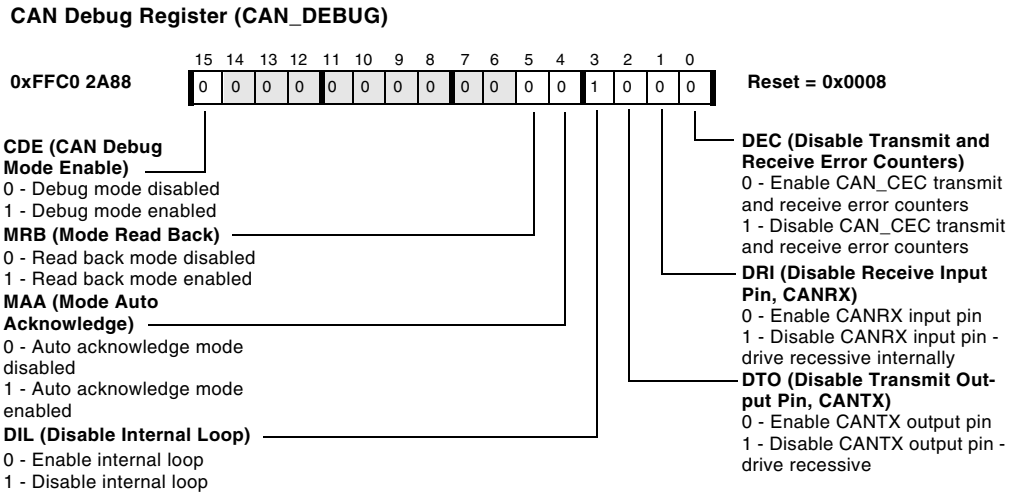


Figure 17-13. CAN Debug Register

CAN_CLOCK Register

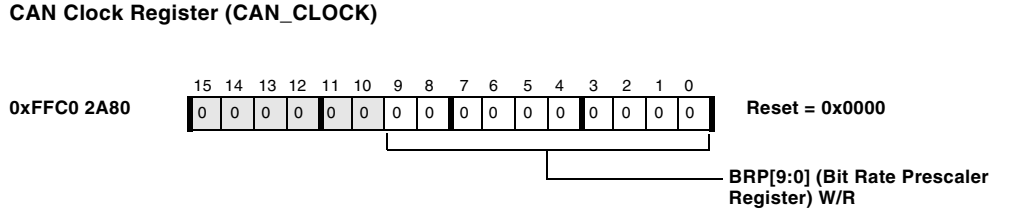


Figure 17-14. CAN Clock Register

CAN_TIMING Register

CAN Timing Register (CAN_TIMING)

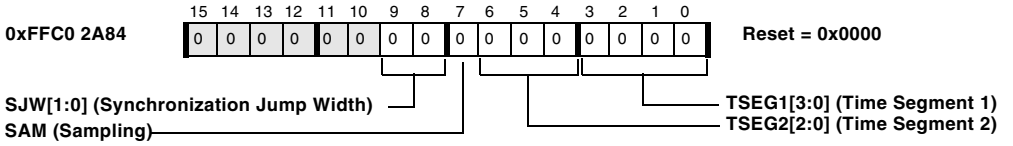


Figure 17-15. CAN Timing Register

CAN_INTR Register

CAN Interrupt Register (CAN_INTR)
RO

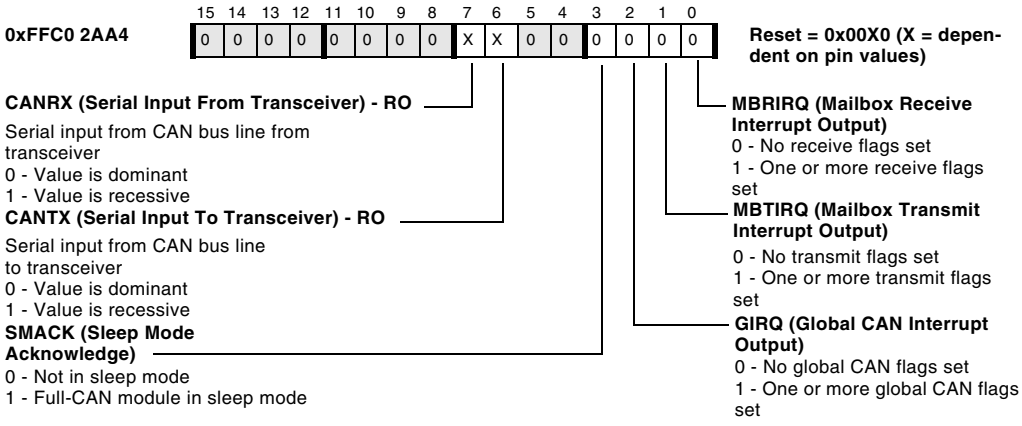


Figure 17-16. CAN Interrupt Register

CAN_GIM Register

Global CAN Interrupt Mask Register (CAN_GIM)

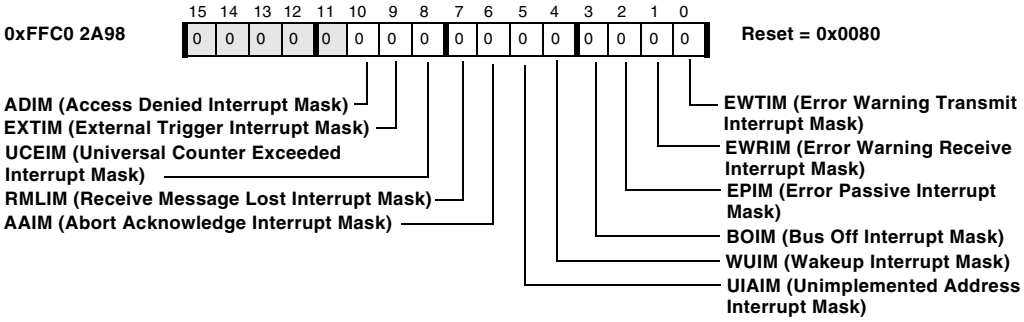


Figure 17-17. Global CAN Interrupt Mask Register

CAN_GIS Register

Global CAN Interrupt Status Register (CAN_GIS)
All bits are W1C

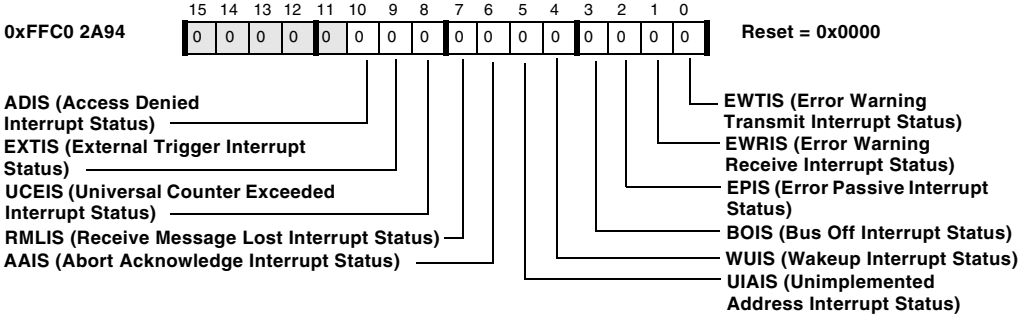


Figure 17-18. Global CAN Interrupt Status Register

CAN_GIF Register

Global CAN Interrupt Flag Register (CAN_GIF)

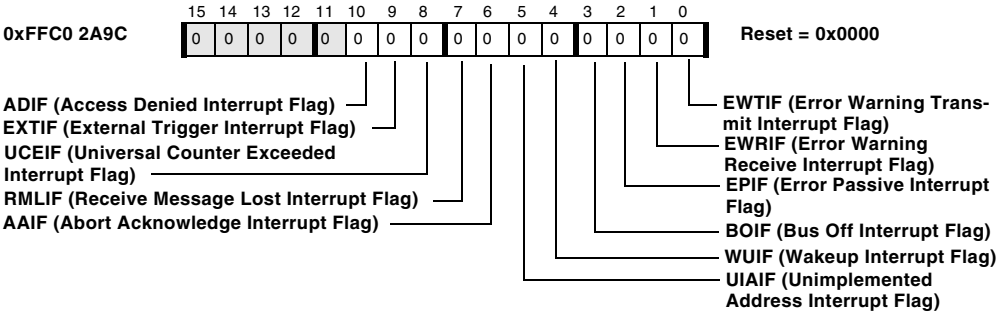


Figure 17-19. Global CAN Interrupt Flag Register

Mailbox/Mask Registers

Figure 17-20 through Figure 17-29 show the CAN mailbox and mask registers.

CAN_AMxx Registers

Acceptance Mask Register (CAN_AMxxH)

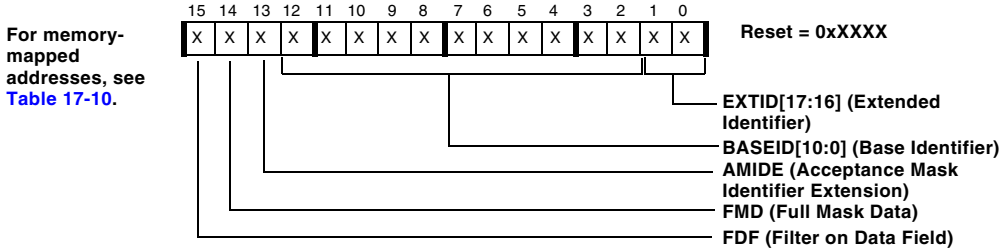


Figure 17-20. Acceptance Mask Register (H)

The value of the acceptance mask register does not care when the AME bit is zero. If AME is set, only those bits are compared that have the corresponding mask bit cleared. A bit position that is one in the mask register does not need to match.

Table 17-10. Acceptance Mask Register (H) Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_AM00H	0xFFC0 2B04
CAN_AM01H	0xFFC0 2B0C
CAN_AM02H	0xFFC0 2B14
CAN_AM03H	0xFFC0 2B1C
CAN_AM04H	0xFFC0 2B24
CAN_AM05H	0xFFC0 2B2C
CAN_AM06H	0xFFC0 2B34
CAN_AM07H	0xFFC0 2B3C
CAN_AM08H	0xFFC0 2B44
CAN_AM09H	0xFFC0 2B4C
CAN_AM10H	0xFFC0 2B54
CAN_AM11H	0xFFC0 2B5C
CAN_AM12H	0xFFC0 2B64
CAN_AM13H	0xFFC0 2B6C
CAN_AM14H	0xFFC0 2B74
CAN_AM15H	0xFFC0 2B7C
CAN_AM16H	0xFFC0 2B84
CAN_AM17H	0xFFC0 2B8C
CAN_AM18H	0xFFC0 2B94
CAN_AM19H	0xFFC0 2B9C
CAN_AM20H	0xFFC0 2BA4

Table 17-10. Acceptance Mask Register (H) Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_AM21H	0xFFC0 2BAC
CAN_AM22H	0xFFC0 2BB4
CAN_AM23H	0xFFC0 2BBC
CAN_AM24H	0xFFC0 2BC4
CAN_AM25H	0xFFC0 2BCC
CAN_AM26H	0xFFC0 2BD4
CAN_AM27H	0xFFC0 2BDC
CAN_AM28H	0xFFC0 2BE4
CAN_AM29H	0xFFC0 2BEC
CAN_AM30H	0xFFC0 2BF4
CAN_AM31H	0xFFC0 2BFC

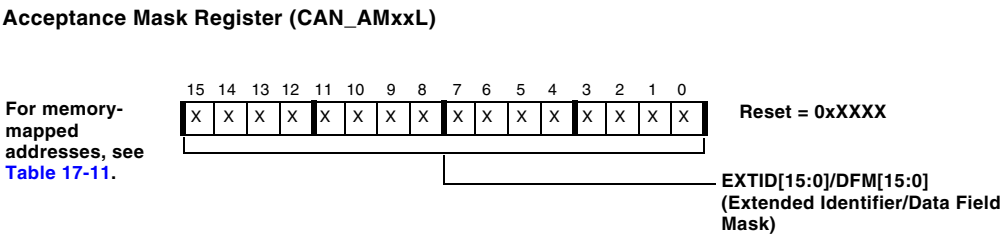


Figure 17-21. Acceptance Mask Register (L)

Table 17-11. Acceptance Mask Register (L) Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_AM00L	0xFFC0 2B00
CAN_AM01L	0xFFC0 2B08
CAN_AM02L	0xFFC0 2B10

Table 17-11. Acceptance Mask Register (L) Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_AM03L	0xFFC0 2B18
CAN_AM04L	0xFFC0 2B20
CAN_AM05L	0xFFC0 2B28
CAN_AM06L	0xFFC0 2B30
CAN_AM07L	0xFFC0 2B38
CAN_AM08L	0xFFC0 2B40
CAN_AM09L	0xFFC0 2B48
CAN_AM10L	0xFFC0 2B50
CAN_AM11L	0xFFC0 2B58
CAN_AM12L	0xFFC0 2B60
CAN_AM13L	0xFFC0 2B68
CAN_AM14L	0xFFC0 2B70
CAN_AM15L	0xFFC0 2B78
CAN_AM16L	0xFFC0 2B80
CAN_AM17L	0xFFC0 2B88
CAN_AM18L	0xFFC0 2B90
CAN_AM19L	0xFFC0 2B98
CAN_AM20L	0xFFC0 2BA0
CAN_AM21L	0xFFC0 2BA8
CAN_AM22L	0xFFC0 2BB0
CAN_AM23L	0xFFC0 2BB8
CAN_AM24L	0xFFC0 2BC0
CAN_AM25L	0xFFC0 2BC8
CAN_AM26L	0xFFC0 2BD0
CAN_AM27L	0xFFC0 2BD8

Table 17-11. Acceptance Mask Register (L) Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_AM28L	0xFFC0 2BE0
CAN_AM29L	0xFFC0 2BE8
CAN_AM30L	0xFFC0 2BF0
CAN_AM31L	0xFFC0 2BF8

CAN_MBxx_ID1 Registers

Mailbox Word 7 Register (CAN_MBxx_ID1)

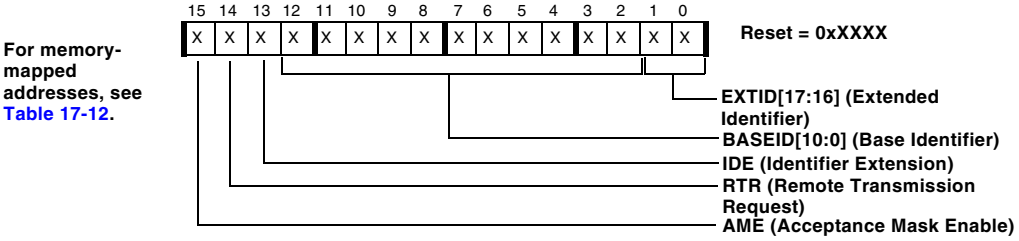


Figure 17-22. Mailbox Word 7 Register

Table 17-12. Mailbox Word 7 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_ID1	0xFFC0 2C1C
CAN_MB01_ID1	0xFFC0 2C3C
CAN_MB02_ID1	0xFFC0 2C5C
CAN_MB03_ID1	0xFFC0 2C7C
CAN_MB04_ID1	0xFFC0 2C9C
CAN_MB05_ID1	0xFFC0 2CBC
CAN_MB06_ID1	0xFFC0 2CDC

Table 17-12. Mailbox Word 7 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB07_ID1	0xFFC0 2CFC
CAN_MB08_ID1	0xFFC0 2D1C
CAN_MB09_ID1	0xFFC0 2D3C
CAN_MB10_ID1	0xFFC0 2D5C
CAN_MB11_ID1	0xFFC0 2D7C
CAN_MB12_ID1	0xFFC0 2D9C
CAN_MB13_ID1	0xFFC0 2DBC
CAN_MB14_ID1	0xFFC0 2DDC
CAN_MB15_ID1	0xFFC0 2DFC
CAN_MB16_ID1	0xFFC0 2E1C
CAN_MB17_ID1	0xFFC0 2E3C
CAN_MB18_ID1	0xFFC0 2E5C
CAN_MB19_ID1	0xFFC0 2E7C
CAN_MB20_ID1	0xFFC0 2E9C
CAN_MB21_ID1	0xFFC0 2EBC
CAN_MB22_ID1	0xFFC0 2EDC
CAN_MB23_ID1	0xFFC0 2EFC
CAN_MB24_ID1	0xFFC0 2F1C
CAN_MB25_ID1	0xFFC0 2F3C
CAN_MB26_ID1	0xFFC0 2F5C
CAN_MB27_ID1	0xFFC0 2F7C
CAN_MB28_ID1	0xFFC0 2F9C
CAN_MB29_ID1	0xFFC0 2FBC
CAN_MB30_ID1	0xFFC0 2FDC
CAN_MB31_ID1	0xFFC0 2FFC

CAN_MBxx_ID0 Registers

Mailbox Word 6 Register (CAN_MBxx_ID0)

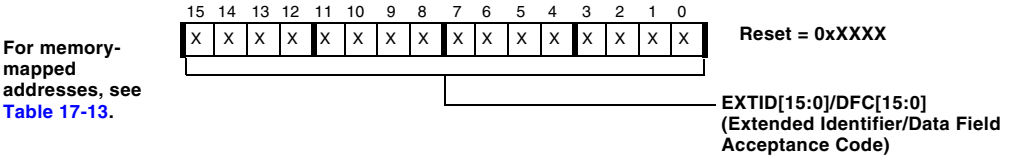


Figure 17-23. Mailbox Word 6 Register

Table 17-13. Mailbox Word 6 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_ID0	0xFFC0 2C18
CAN_MB01_ID0	0xFFC0 2C38
CAN_MB02_ID0	0xFFC0 2C58
CAN_MB03_ID0	0xFFC0 2C78
CAN_MB04_ID0	0xFFC0 2C98
CAN_MB05_ID0	0xFFC0 2CB8
CAN_MB06_ID0	0xFFC0 2CD8
CAN_MB07_ID0	0xFFC0 2CF8
CAN_MB08_ID0	0xFFC0 2D18
CAN_MB09_ID0	0xFFC0 2D38
CAN_MB10_ID0	0xFFC0 2D58
CAN_MB11_ID0	0xFFC0 2D78
CAN_MB12_ID0	0xFFC0 2D98
CAN_MB13_ID0	0xFFC0 2DB8
CAN_MB14_ID0	0xFFC0 2DD8
CAN_MB15_ID0	0xFFC0 2DF8

Table 17-13. Mailbox Word 6 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB16_ID0	0xFFC0 2E18
CAN_MB17_ID0	0xFFC0 2E38
CAN_MB18_ID0	0xFFC0 2E58
CAN_MB19_ID0	0xFFC0 2E78
CAN_MB20_ID0	0xFFC0 2E98
CAN_MB21_ID0	0xFFC0 2EB8
CAN_MB22_ID0	0xFFC0 2ED8
CAN_MB23_ID0	0xFFC0 2EF8
CAN_MB24_ID0	0xFFC0 2F18
CAN_MB25_ID0	0xFFC0 2F38
CAN_MB26_ID0	0xFFC0 2F58
CAN_MB27_ID0	0xFFC0 2F78
CAN_MB28_ID0	0xFFC0 2F98
CAN_MB29_ID0	0xFFC0 2FB8
CAN_MB30_ID0	0xFFC0 2FD8
CAN_MB31_ID0	0xFFC0 2FF8

CAN_MBxx_TIMESTAMP Registers

Mailbox Word 5 Register (CAN_MBxx_ID0)

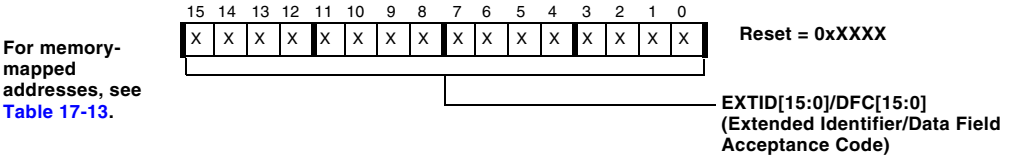


Figure 17-24. Mailbox Word 5 Register

Table 17-14. Mailbox Word 5 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_TIMESTAMP	0xFFC0 2C14
CAN_MB01_TIMESTAMP	0xFFC0 2C34
CAN_MB02_TIMESTAMP	0xFFC0 2C54
CAN_MB03_TIMESTAMP	0xFFC0 2C74
CAN_MB04_TIMESTAMP	0xFFC0 2C94
CAN_MB05_TIMESTAMP	0xFFC0 2CB4
CAN_MB06_TIMESTAMP	0xFFC0 2CD4
CAN_MB07_TIMESTAMP	0xFFC0 2CF4
CAN_MB08_TIMESTAMP	0xFFC0 2D14
CAN_MB09_TIMESTAMP	0xFFC0 2D34
CAN_MB10_TIMESTAMP	0xFFC0 2D54
CAN_MB11_TIMESTAMP	0xFFC0 2D74
CAN_MB12_TIMESTAMP	0xFFC0 2D94
CAN_MB13_TIMESTAMP	0xFFC0 2DB4
CAN_MB14_TIMESTAMP	0xFFC0 2DD4
CAN_MB15_TIMESTAMP	0xFFC0 2DF4

Table 17-14. Mailbox Word 5 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB16_TIMESTAMP	0xFFC0 2E14
CAN_MB17_TIMESTAMP	0xFFC0 2E34
CAN_MB18_TIMESTAMP	0xFFC0 2E54
CAN_MB19_TIMESTAMP	0xFFC0 2E74
CAN_MB20_TIMESTAMP	0xFFC0 2E94
CAN_MB21_TIMESTAMP	0xFFC0 2EB4
CAN_MB22_TIMESTAMP	0xFFC0 2ED4
CAN_MB23_TIMESTAMP	0xFFC0 2EF4
CAN_MB24_TIMESTAMP	0xFFC0 2F14
CAN_MB25_TIMESTAMP	0xFFC0 2F34
CAN_MB26_TIMESTAMP	0xFFC0 2F54
CAN_MB27_TIMESTAMP	0xFFC0 2F74
CAN_MB28_TIMESTAMP	0xFFC0 2F94
CAN_MB29_TIMESTAMP	0xFFC0 2FB4
CAN_MB30_TIMESTAMP	0xFFC0 2FD4
CAN_MB31_TIMESTAMP	0xFFC0 2FF4

CAN_MBxx_LENGTH Registers

Mailbox Word 4 Register (CAN_MBxx_LENGTH)

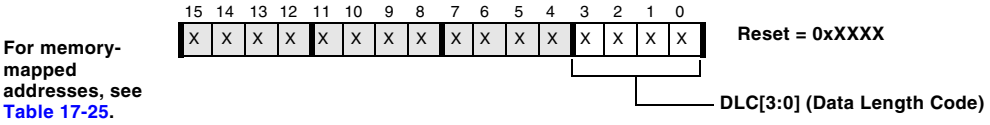


Figure 17-25. Mailbox Word 4 Register

Table 17-15. Mailbox Word 4 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_LENGTH	0xFFC0 2C10
CAN_MB01_LENGTH	0xFFC0 2C30
CAN_MB02_LENGTH	0xFFC0 2C50
CAN_MB03_LENGTH	0xFFC0 2C70
CAN_MB04_LENGTH	0xFFC0 2C90
CAN_MB05_LENGTH	0xFFC0 2CB0
CAN_MB06_LENGTH	0xFFC0 2CD0
CAN_MB07_LENGTH	0xFFC0 2CF0
CAN_MB08_LENGTH	0xFFC0 2D10
CAN_MB09_LENGTH	0xFFC0 2D30
CAN_MB10_LENGTH	0xFFC0 2D50
CAN_MB11_LENGTH	0xFFC0 2D70
CAN_MB12_LENGTH	0xFFC0 2D90
CAN_MB13_LENGTH	0xFFC0 2DB0
CAN_MB14_LENGTH	0xFFC0 2DD0
CAN_MB15_LENGTH	0xFFC0 2DF0
CAN_MB16_LENGTH	0xFFC0 2E10

Table 17-15. Mailbox Word 4 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB17_LENGTH	0xFFC0 2E30
CAN_MB18_LENGTH	0xFFC0 2E50
CAN_MB19_LENGTH	0xFFC0 2E70
CAN_MB20_LENGTH	0xFFC0 2E90
CAN_MB21_LENGTH	0xFFC0 2EB0
CAN_MB22_LENGTH	0xFFC0 2ED0
CAN_MB23_LENGTH	0xFFC0 2EF0
CAN_MB24_LENGTH	0xFFC0 2F10
CAN_MB25_LENGTH	0xFFC0 2F30
CAN_MB26_LENGTH	0xFFC0 2F50
CAN_MB27_LENGTH	0xFFC0 2F70
CAN_MB28_LENGTH	0xFFC0 2F90
CAN_MB29_LENGTH	0xFFC0 2FB0
CAN_MB30_LENGTH	0xFFC0 2FD0
CAN_MB31_LENGTH	0xFFC0 2FF0

CAN_MBxx_DATAx Registers

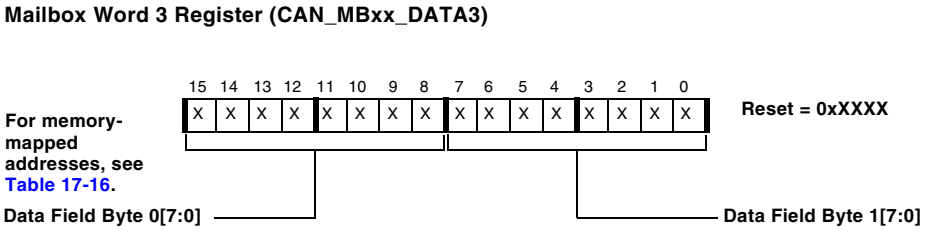


Figure 17-26. Mailbox Word 3 Register

Table 17-16. Mailbox Word 3 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_DATA3	0xFFC0 2C0C
CAN_MB01_DATA3	0xFFC0 2C2C
CAN_MB02_DATA3	0xFFC0 2C4C
CAN_MB03_DATA3	0xFFC0 2C6C
CAN_MB04_DATA3	0xFFC0 2C8C
CAN_MB05_DATA3	0xFFC0 2CAC
CAN_MB06_DATA3	0xFFC0 2CCC
CAN_MB07_DATA3	0xFFC0 2CEC
CAN_MB08_DATA3	0xFFC0 2D0C
CAN_MB09_DATA3	0xFFC0 2D2C
CAN_MB10_DATA3	0xFFC0 2D4C
CAN_MB11_DATA3	0xFFC0 2D6C
CAN_MB12_DATA3	0xFFC0 2D8C
CAN_MB13_DATA3	0xFFC0 2DAC
CAN_MB14_DATA3	0xFFC0 2DCC
CAN_MB15_DATA3	0xFFC0 2DEC
CAN_MB16_DATA3	0xFFC0 2E0C
CAN_MB17_DATA3	0xFFC0 2E2C
CAN_MB18_DATA3	0xFFC0 2E4C
CAN_MB19_DATA3	0xFFC0 2E6C
CAN_MB20_DATA3	0xFFC0 2E8C
CAN_MB21_DATA3	0xFFC0 2EAC
CAN_MB22_DATA3	0xFFC0 2ECC
CAN_MB23_DATA3	0xFFC0 2EEC
CAN_MB24_DATA3	0xFFC0 2F0C

Table 17-16. Mailbox Word 3 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB25_DATA3	0xFFC0 2F2C
CAN_MB26_DATA3	0xFFC0 2F4C
CAN_MB27_DATA3	0xFFC0 2F6C
CAN_MB28_DATA3	0xFFC0 2F8C
CAN_MB29_DATA3	0xFFC0 2FAC
CAN_MB30_DATA3	0xFFC0 2FCC
CAN_MB31_DATA3	0xFFC0 2FEC

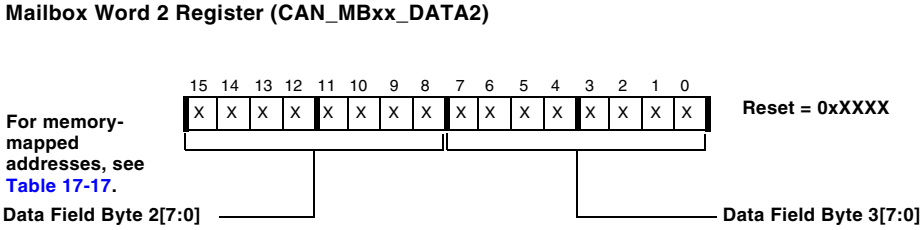


Figure 17-27. Mailbox Word 2 Register

Table 17-17. Mailbox Word 2 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_DATA2	0xFFC0 2C08
CAN_MB01_DATA2	0xFFC0 2C28
CAN_MB02_DATA2	0xFFC0 2C48
CAN_MB03_DATA2	0xFFC0 2C68
CAN_MB04_DATA2	0xFFC0 2C88
CAN_MB05_DATA2	0xFFC0 2CA8
CAN_MB06_DATA2	0xFFC0 2CC8
CAN_MB07_DATA2	0xFFC0 2CE8
CAN_MB08_DATA2	0xFFC0 2D08
CAN_MB09_DATA2	0xFFC0 2D28
CAN_MB10_DATA2	0xFFC0 2D48
CAN_MB11_DATA2	0xFFC0 2D68
CAN_MB12_DATA2	0xFFC0 2D88
CAN_MB13_DATA2	0xFFC0 2DA8
CAN_MB14_DATA2	0xFFC0 2DC8
CAN_MB15_DATA2	0xFFC0 2DE8
CAN_MB16_DATA2	0xFFC0 2E08
CAN_MB17_DATA2	0xFFC0 2E28

Table 17-17. Mailbox Word 2 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB18_DATA2	0xFFC0 2E48
CAN_MB19_DATA2	0xFFC0 2E68
CAN_MB20_DATA2	0xFFC0 2E88
CAN_MB21_DATA2	0xFFC0 2EA8
CAN_MB22_DATA2	0xFFC0 2EC8
CAN_MB23_DATA2	0xFFC0 2EE8
CAN_MB24_DATA2	0xFFC0 2F08
CAN_MB25_DATA2	0xFFC0 2F28
CAN_MB26_DATA2	0xFFC0 2F48
CAN_MB27_DATA2	0xFFC0 2F68
CAN_MB28_DATA2	0xFFC0 2F88
CAN_MB29_DATA2	0xFFC0 2FA8
CAN_MB30_DATA2	0xFFC0 2FC8
CAN_MB31_DATA2	0xFFC0 2FE8

Mailbox Word 1 Register (CAN_MBxx_DATA1)

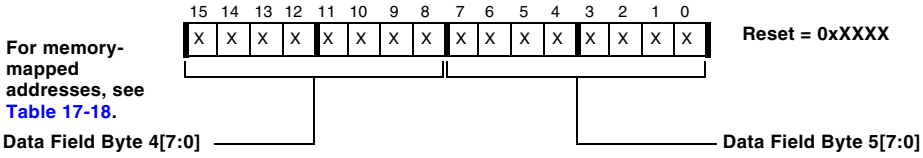


Figure 17-28. Mailbox Word 1 Register

Table 17-18. Mailbox Word 1 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_DATA1	0xFFC0 2C04
CAN_MB01_DATA1	0xFFC0 2C24
CAN_MB02_DATA1	0xFFC0 2C44
CAN_MB03_DATA1	0xFFC0 2C64
CAN_MB04_DATA1	0xFFC0 2C84
CAN_MB05_DATA1	0xFFC0 2CA4
CAN_MB06_DATA1	0xFFC0 2CC4
CAN_MB07_DATA1	0xFFC0 2CE4
CAN_MB08_DATA1	0xFFC0 2D04
CAN_MB09_DATA1	0xFFC0 2D24
CAN_MB10_DATA1	0xFFC0 2D44
CAN_MB11_DATA1	0xFFC0 2D64
CAN_MB12_DATA1	0xFFC0 2D84
CAN_MB13_DATA1	0xFFC0 2DA4
CAN_MB14_DATA1	0xFFC0 2DC4
CAN_MB15_DATA1	0xFFC0 2DE4
CAN_MB16_DATA1	0xFFC0 2E04
CAN_MB17_DATA1	0xFFC0 2E24

Table 17-18. Mailbox Word 1 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB18_DATA1	0xFFC0 2E44
CAN_MB19_DATA1	0xFFC0 2E64
CAN_MB20_DATA1	0xFFC0 2E84
CAN_MB21_DATA1	0xFFC0 2EA4
CAN_MB22_DATA1	0xFFC0 2EC4
CAN_MB23_DATA1	0xFFC0 2EE4
CAN_MB24_DATA1	0xFFC0 2F04
CAN_MB25_DATA1	0xFFC0 2F24
CAN_MB26_DATA1	0xFFC0 2F44
CAN_MB27_DATA1	0xFFC0 2F64
CAN_MB28_DATA1	0xFFC0 2F84
CAN_MB29_DATA1	0xFFC0 2FA4
CAN_MB30_DATA1	0xFFC0 2FC4
CAN_MB31_DATA1	0xFFC0 2FE4

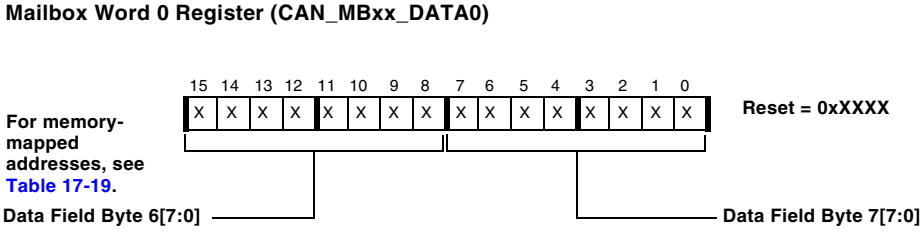


Figure 17-29. Mailbox Word 0 Register

Table 17-19. Mailbox Word 0 Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
CAN_MB00_DATA0	0xFFC0 2C00
CAN_MB01_DATA0	0xFFC0 2C20
CAN_MB02_DATA0	0xFFC0 2C40
CAN_MB03_DATA0	0xFFC0 2C60
CAN_MB04_DATA0	0xFFC0 2C80
CAN_MB05_DATA0	0xFFC0 2CA0
CAN_MB06_DATA0	0xFFC0 2CC0
CAN_MB07_DATA0	0xFFC0 2CE0
CAN_MB08_DATA0	0xFFC0 2D00
CAN_MB09_DATA0	0xFFC0 2D20
CAN_MB10_DATA0	0xFFC0 2D40
CAN_MB11_DATA0	0xFFC0 2D60
CAN_MB12_DATA0	0xFFC0 2D80
CAN_MB13_DATA0	0xFFC0 2DA0
CAN_MB14_DATA0	0xFFC0 2DC0
CAN_MB15_DATA0	0xFFC0 2DE0
CAN_MB16_DATA0	0xFFC0 2E00
CAN_MB17_DATA0	0xFFC0 2E20

Table 17-19. Mailbox Word 0 Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
CAN_MB18_DATA0	0xFFC0 2E40
CAN_MB19_DATA0	0xFFC0 2E60
CAN_MB20_DATA0	0xFFC0 2E80
CAN_MB21_DATA0	0xFFC0 2EA0
CAN_MB22_DATA0	0xFFC0 2EC0
CAN_MB23_DATA0	0xFFC0 2EE0
CAN_MB24_DATA0	0xFFC0 2F00
CAN_MB25_DATA0	0xFFC0 2F20
CAN_MB26_DATA0	0xFFC0 2F40
CAN_MB27_DATA0	0xFFC0 2F60
CAN_MB28_DATA0	0xFFC0 2F80
CAN_MB29_DATA0	0xFFC0 2FA0
CAN_MB30_DATA0	0xFFC0 2FC0
CAN_MB31_DATA0	0xFFC0 2FE0

Mailbox Control Registers

Figure 17-30 through Figure 17-56 show the mailbox control registers.

CAN_MCx Registers

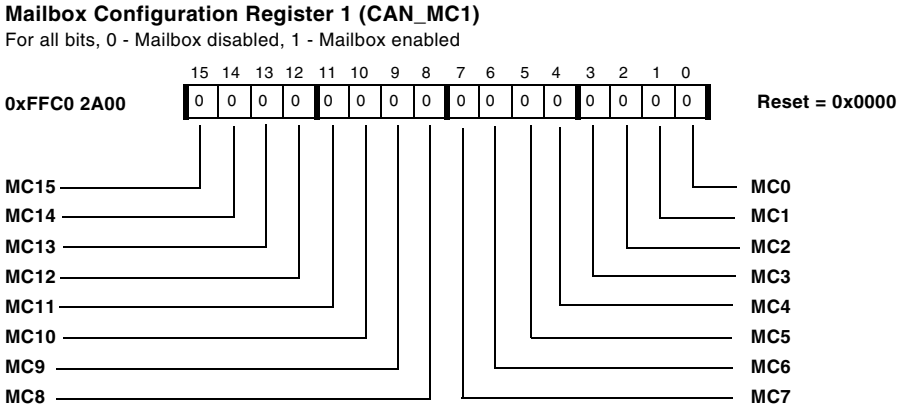


Figure 17-30. Mailbox Configuration Register 1

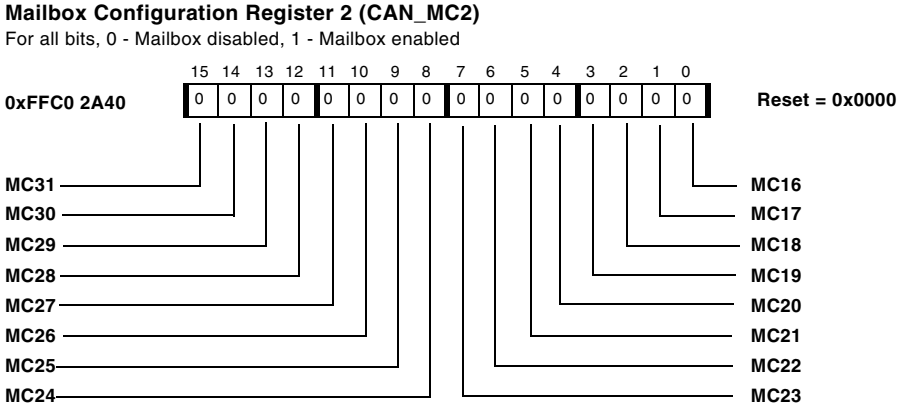


Figure 17-31. Mailbox Configuration Register 2

CAN_MDx Registers

Mailbox Direction Register 1 (CAN_MD1)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

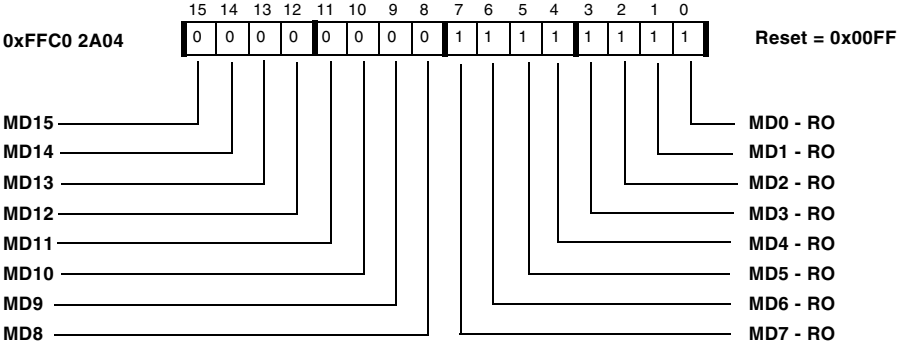


Figure 17-32. Mailbox Direction Register 1

Mailbox Direction Register 2 (CAN_MD2)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

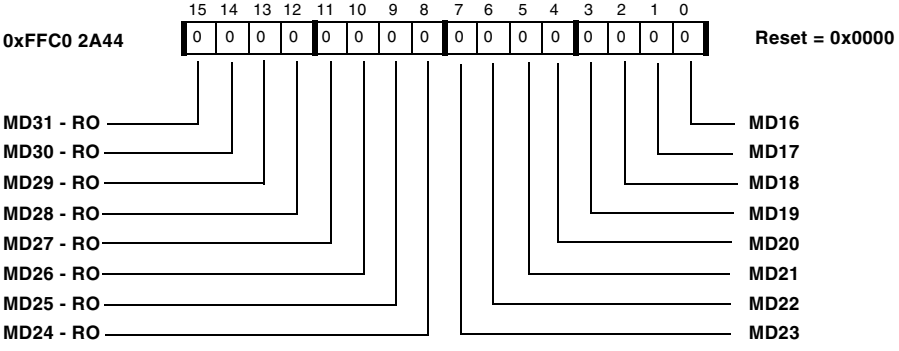


Figure 17-33. Mailbox Direction Register 2

CAN_RMPx Register

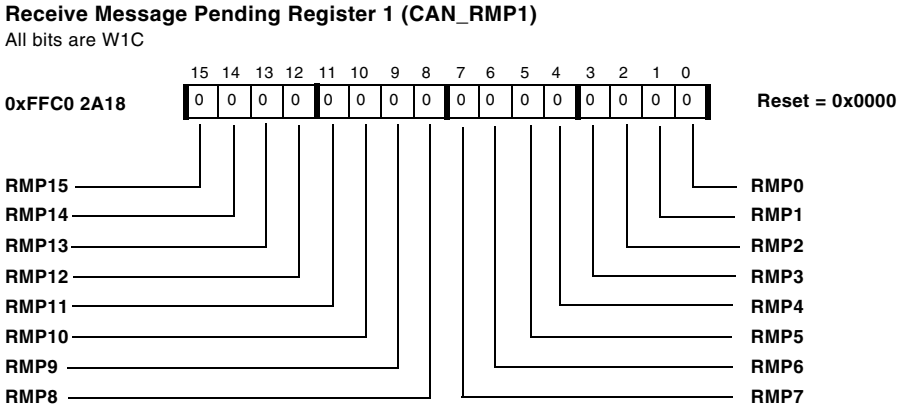


Figure 17-34. Receive Message Pending Register 1

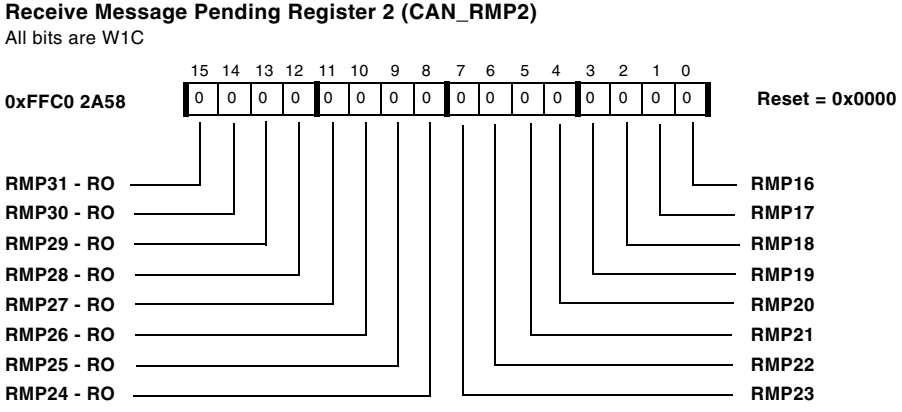


Figure 17-35. Receive Message Pending Register 2

CAN_RMLx Register

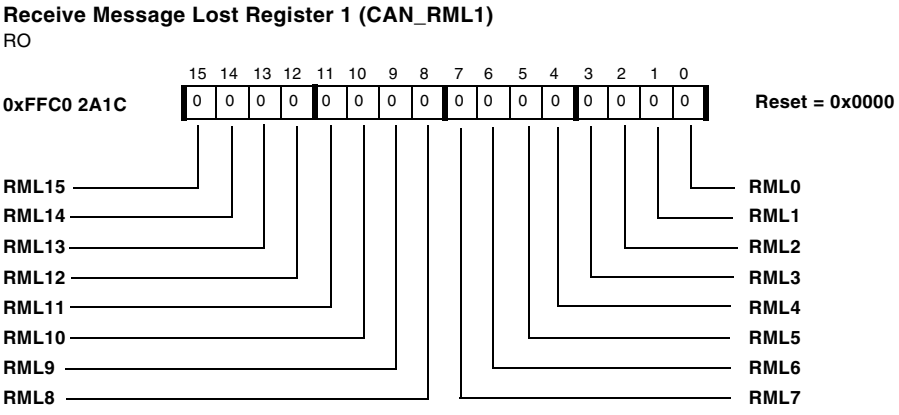


Figure 17-36. Receive Message Lost Register 1

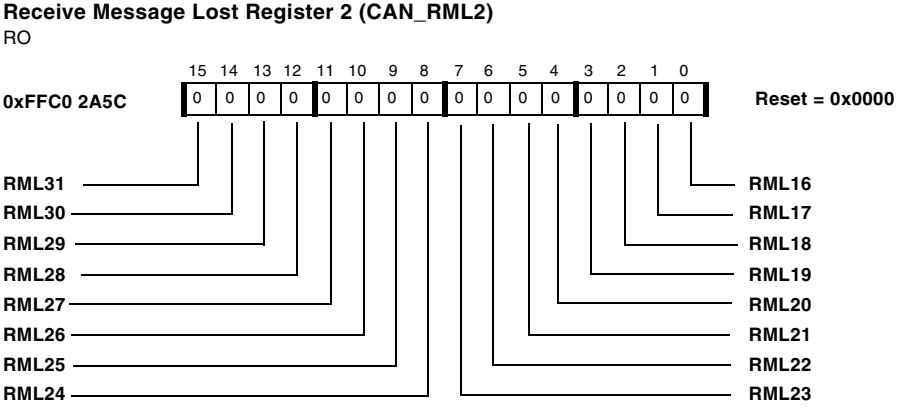


Figure 17-37. Receive Message Lost Register 2

CAN_OPSSx Register

Overwrite Protection/Single Shot Transmission Register 1 (CAN_OPSS1)

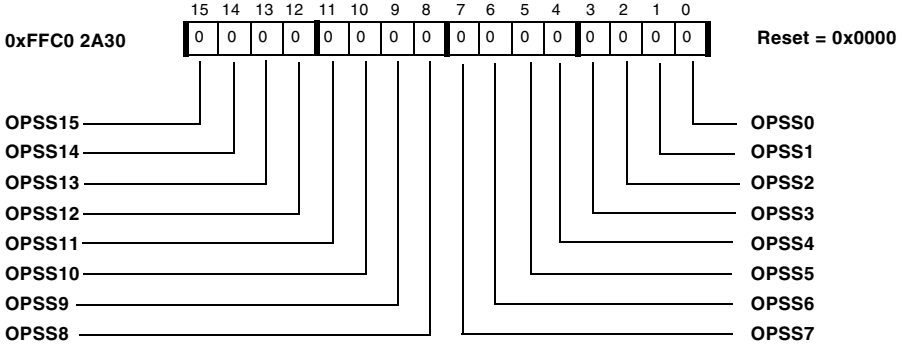


Figure 17-38. Overwrite Protection/Single Shot Transmission Register 1

Overwrite Protection/Single Shot Transmission Register 2 (CAN_OPSS2)

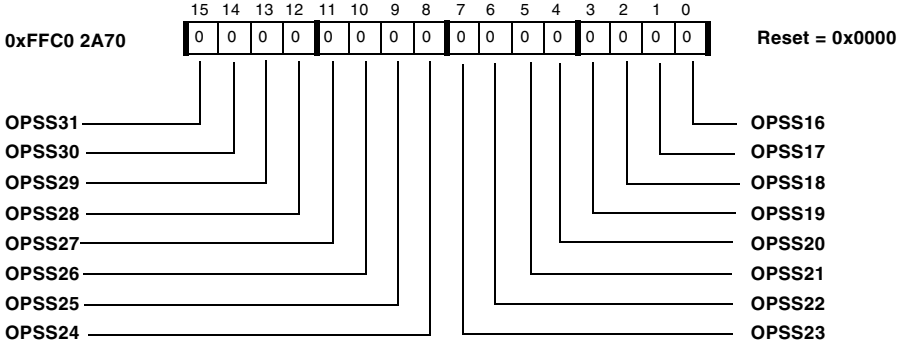


Figure 17-39. Overwrite Protection/Single Shot Transmission Register 2

CAN_TRSx Registers

Transmission Request Set Register 1 (CAN_TRS1)

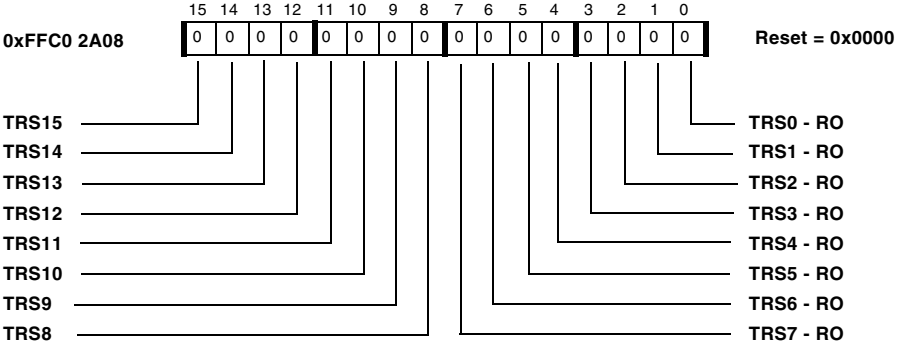


Figure 17-40. Transmission Request Set Register 1

Transmission Request Set Register 2 (CAN_TRS2)

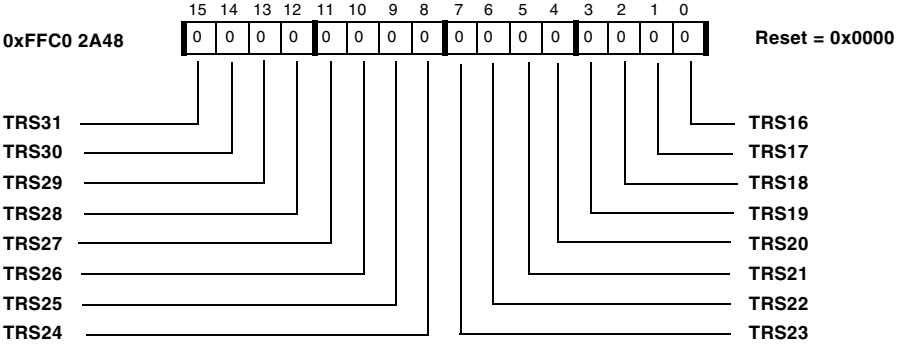


Figure 17-41. Transmission Request Set Register 2

CAN_TRRx Registers

Transmission Request Reset Register 1 (CAN_TRR1)

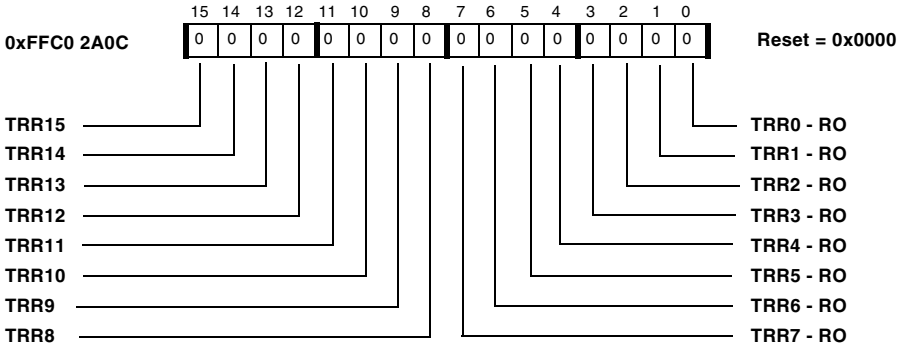


Figure 17-42. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CAN_TRR2)

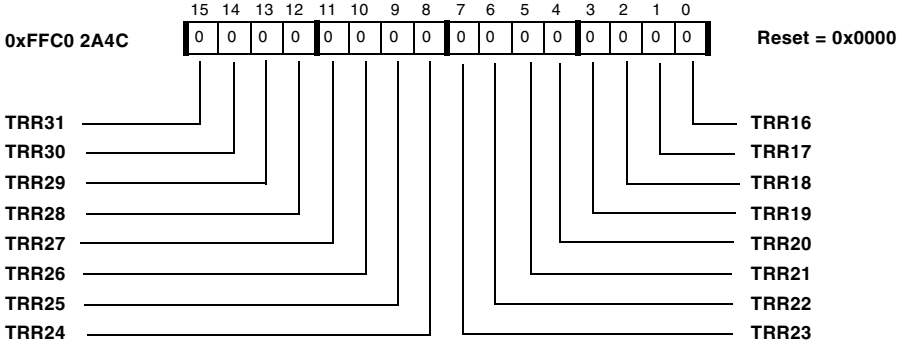


Figure 17-43. Transmission Request Reset Register 2

CAN_AAx Register

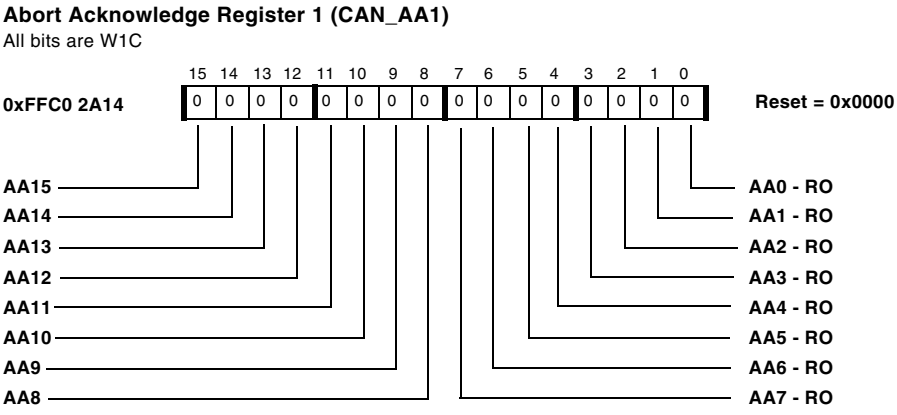


Figure 17-44. Abort Acknowledge Register 1

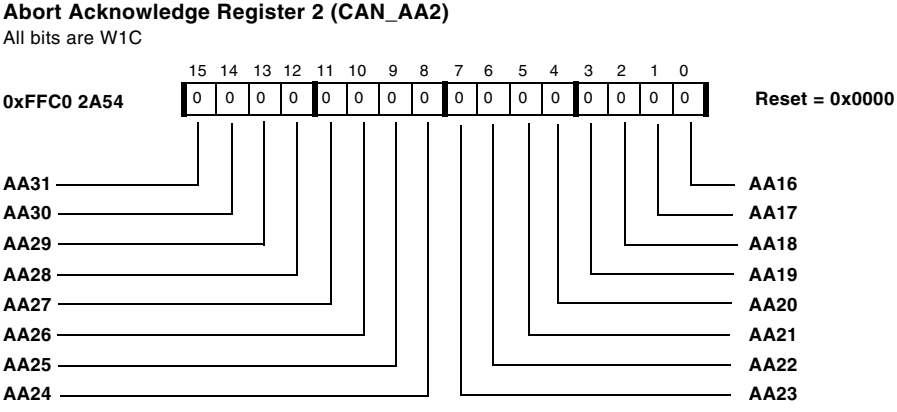


Figure 17-45. Abort Acknowledge Register 2

CAN_TAx Register

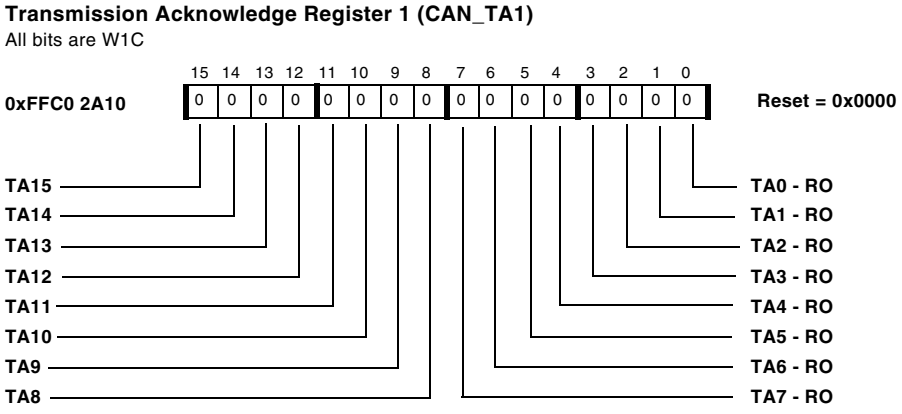


Figure 17-46. Transmission Acknowledge Register 1

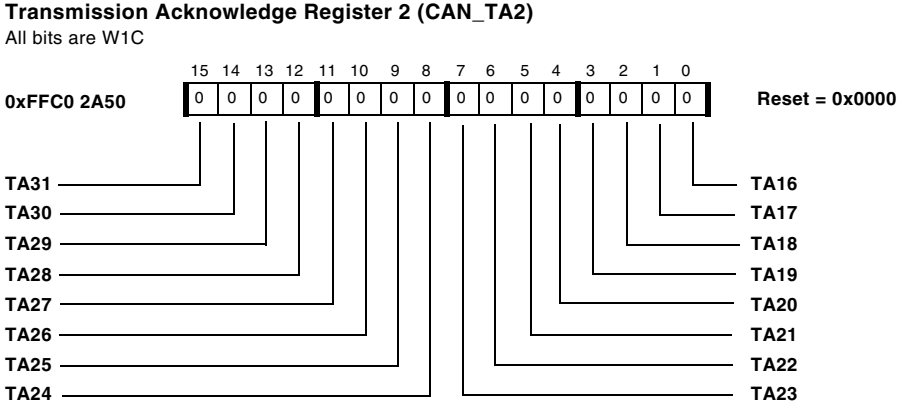


Figure 17-47. Transmission Acknowledge Register 2

CAN_MBTD Register

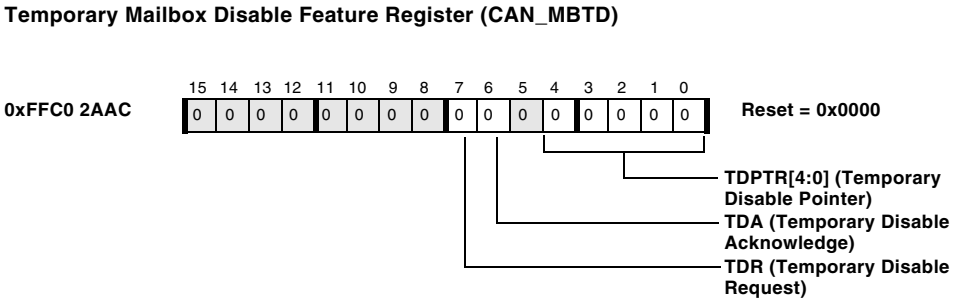


Figure 17-48. Temporary Mailbox Disable Register

CAN_RFHx Registers

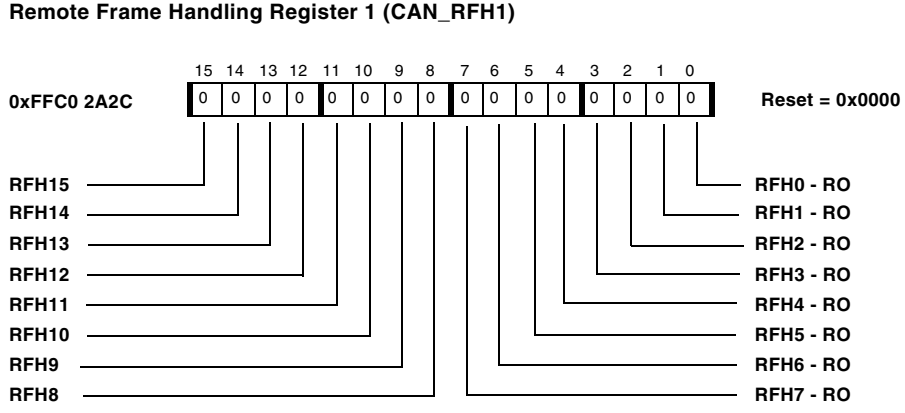


Figure 17-49. Remote Frame Handling Register 1

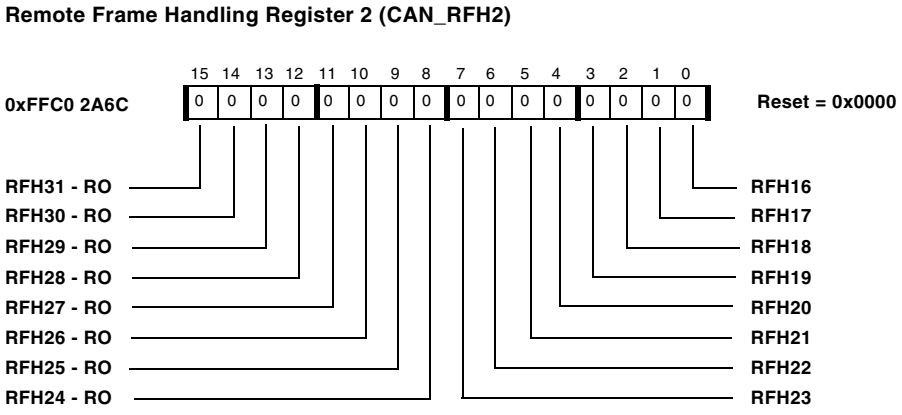


Figure 17-50. Remote Frame Handling Register 2

CAN_MBIMx Registers

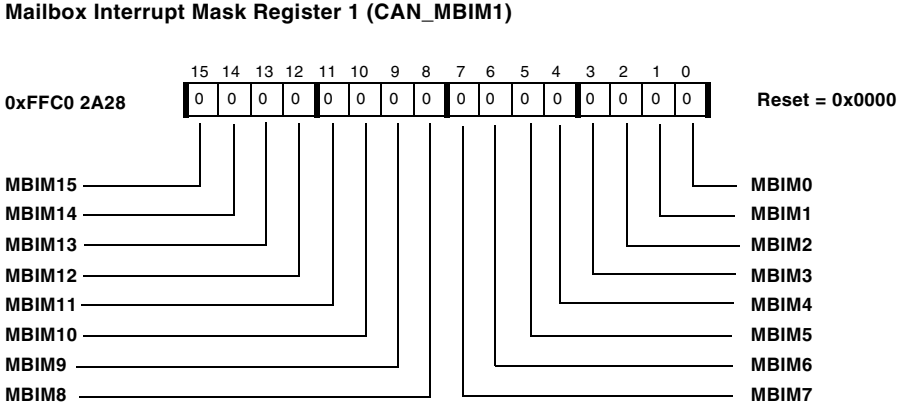


Figure 17-51. Mailbox Interrupt Mask Register 1

Mailbox Interrupt Mask Register 2 (CAN_MBIM2)

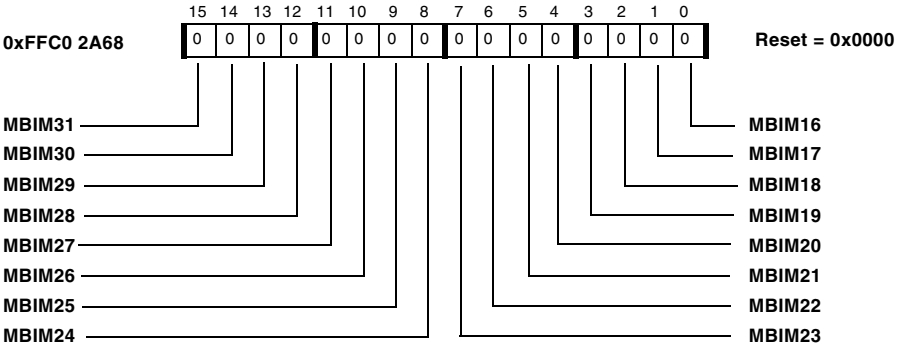


Figure 17-52. Mailbox Interrupt Mask Register 2

CAN_MBTIFx Registers

Mailbox Transmit Interrupt Flag Register 1 (CAN_MBTIF1)

All bits are W1C

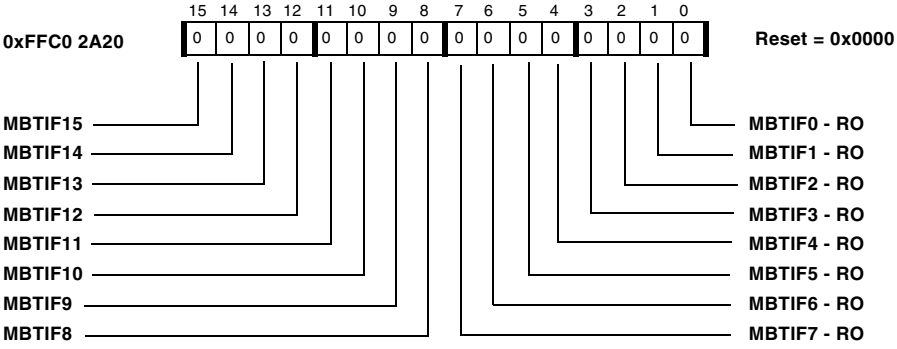


Figure 17-53. Mailbox Transmit Interrupt Flag Register 1

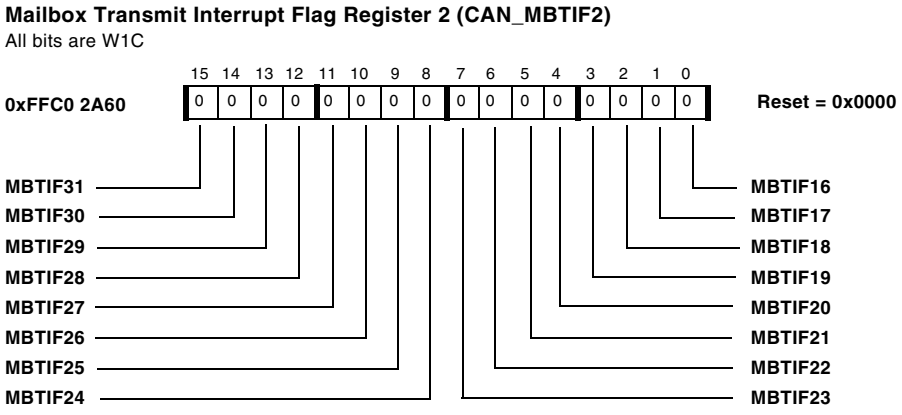


Figure 17-54. Mailbox Transmit Interrupt Flag Register 2

CAN_MBRIFx Registers

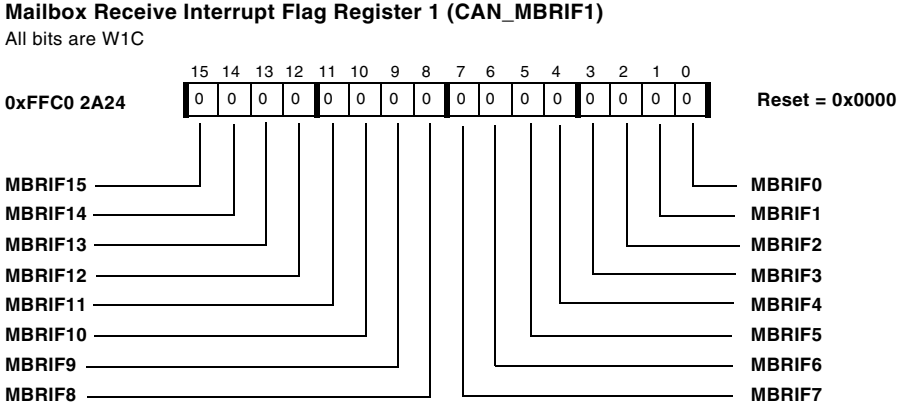


Figure 17-55. Mailbox Receive Interrupt Flag Register 1

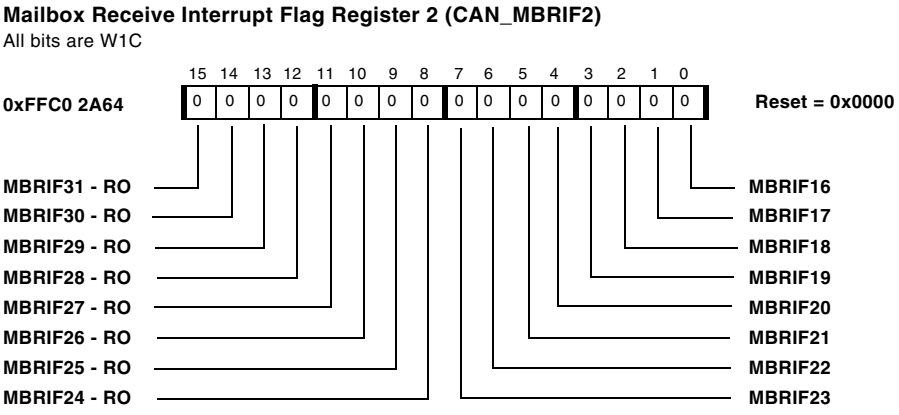


Figure 17-56. Mailbox Receive Interrupt Flag Register 2

Universal Counter Registers

Figure 17-57 through Figure 17-59 show the universal counter registers.

CAN_UCCNF Register

Universal Counter Configuration Mode Register (CAN_UCCNF)

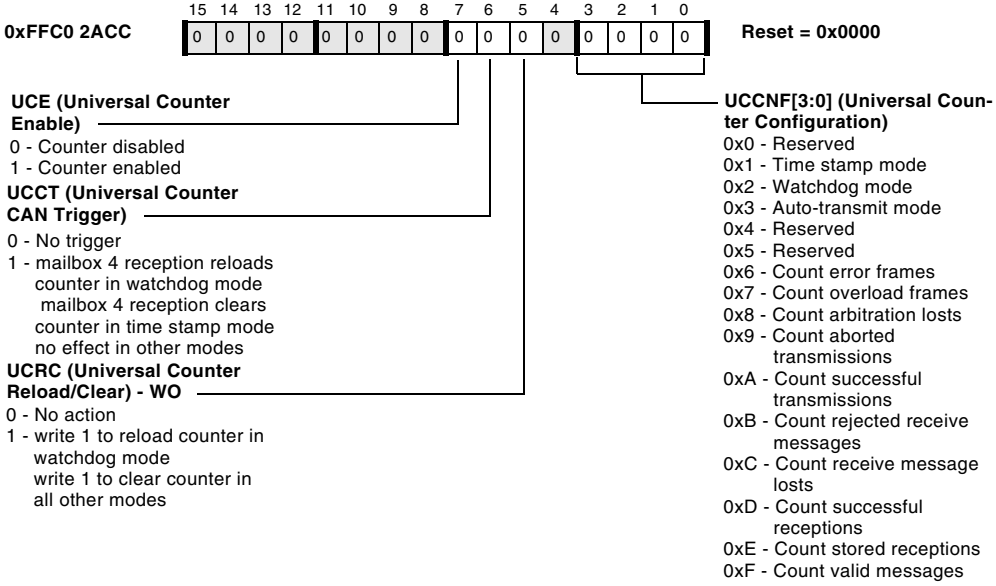


Figure 17-57. Universal Counter Configuration Mode Register

CAN_UCCNT Register

Universal Counter Register (CAN_UCCNT)

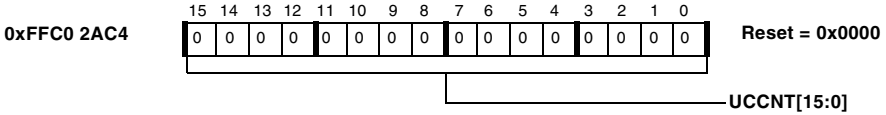


Figure 17-58. Universal Counter Register

CAN_UCRC Register

Universal Counter Reload/Capture Register (CAN_UCRC)

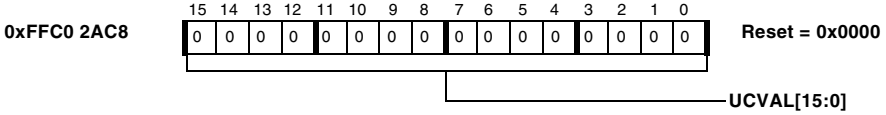


Figure 17-59. Universal Counter Reload/Capture Register

Error Registers

Figure 17-60 through Figure 17-62 show the CAN error registers.

CAN_CEC Register

CAN Error Counter Register (CAN_CEC)

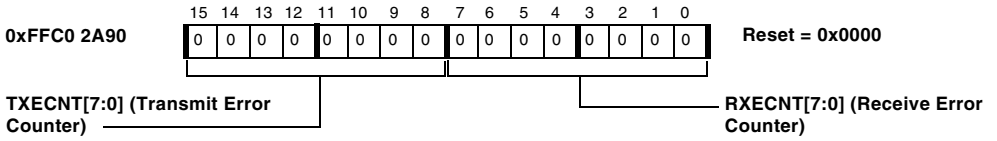


Figure 17-60. CAN Error Counter Register

CAN_ESR Register

Error Status Register (CAN_ESR)
All bits are W1C

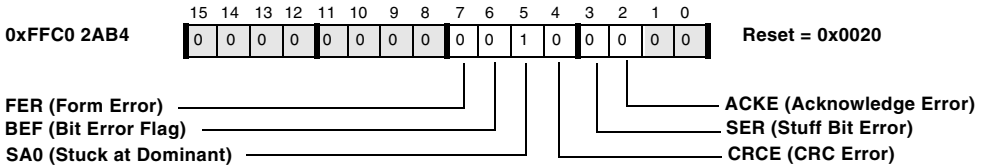


Figure 17-61. Error Status Register

CAN_EWR Register

CAN Error Counter Warning Level Register (CAN_EWR)

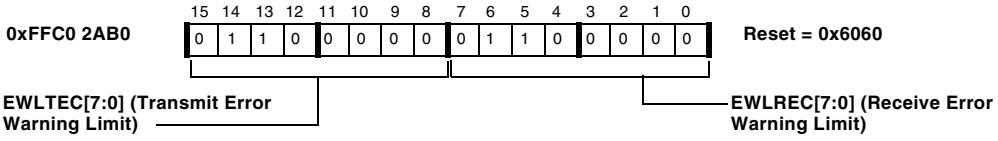


Figure 17-62. CAN Error Counter Warning Level Register

Programming Examples

The following CAN code examples (Listing 17-2 through Listing 17-4) show how to program the CAN hardware and timing, initialize mailboxes, perform transfers, and service interrupts. Each of these code examples assumes that the appropriate header file is included in the source code (that is, `#include <defBF537.h>` for ADSP-BF537 projects).

CAN Setup Code

The following code initializes the port pins to connect to the CAN controller and configures the CAN timing parameters.

Listing 17-2. Initializing CAN

```
Initialize_CAN:
  P0.H = HI(PORT_MUX);  /* CAN pins muxed on Port J */
  P0.L = LO(PORT_MUX);
  R0 = PJCE_CAN(Z);  /* Enable CAN TX/RX pins */
  W[P0] = R0;
  SSYNC;

  /* ================================================
  ** Set CAN Bit Timing
  **
  ** CAN_TIMING - SJW, TSEG2, and TSEG1 governed by:
  ** SJW <= TSEG2 <= TSEG1
  **
  ** ================================================
  */
  P0.H = HI(CAN_TIMING);
  P0.L = LO(CAN_TIMING);
```

```
R0 = 0x0334(Z);  /* SJW = 3, TSEG2 = 3, TSEG1 = 4 */
W[P0] = R0;
SSYNC;

/* ================================================
** CAN_CLOCK  - Calculate Prescaler (BRP)
**
** Assume a 500kbps CAN rate is desired, which means
** the duration of the bit on the CAN bus (tBIT) is
** 2us. Using the tBIT formula from the HRM, solve for
** TQ:
**
** tBIT = TQ x (1 + (TSEG1 + 1) + (TSEG2 + 1))
** 2us  = TQ x (1 + (4 + 1) + (3 + 1))
** 2e-6 = TQ x (1 + 5 + 4)
** TQ = 2e-6 / 10
** TQ = 2e-7
**
** Once time quantum (TQ) is known, BRP can be derived
** from the TQ formula in the HRM. Assume the default
** PLL settings are used for the ADSP-BF537 EZ-KIT,
** which implies that System Clock (SCLK) is 50MHz:
**
** TQ = (BRP+1) / SCLK
** 2e-7 = (BRP+1) / 50e6
** (BRP+1) = 10
** BRP = 9
*/
P0.L = LO(CAN_CLOCK);
R0   = 9(Z);
W[P0] = R0;
SSYNC;

RTS;
```

Initializing and Enabling CAN Mailboxes

Before the CAN can transfer data, the mailbox area must be properly set up and the controller must be initialized properly.

Listing 17-3. Initializing and Enabling Mailboxes

```
CAN_Initialize_Mailboxes:
  P0.H = HI(CAN_MD1);  /* Configure Mailbox Direction */
  P0.L = LO(CAN_MD1);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(MD8));  /* Set MB08 for Transmit */
  BITSET(R0, BITPOS(MD9));  /* Set MB09 for Receive */
  W[P0] = R0;
  SSYNC;

  /* ================================================
  ** Populate CAN Mailbox Area
  **
  ** Mailbox 8 transmits ID 0x411 with 4 bytes of data
  ** Bytes 0 and 1 are a data pattern 0xAABB. Bytes 2
  ** and 3 will be a count value for the number of times
  ** that message is properly sent.
  **
  ** Mailbox 9 will receive message ID 0x007
  **
  ** ================================================
  */

  /* Initialize Mailbox 8 For Transmit */
  R0 = 0x411 << 2;  /* Put Message ID in correct slot */
  P0.L = LO(CAN_MB_ID1(8));  /* Access MB08 ID1 Register */
  W[P0] = R0;  /* Remote frame disabled, 11 bit ID */

  R0 = 0;
  P0.L = LO(CAN_MB_ID0(8));
  W[P0] = R0;  /* Zero Out Lower ID Register */

  R0 = 4;
  P0.L = LO(CAN_MB_LENGTH(8));
  W[P0] = R0;  /* Set DLC to 4 Bytes */
```

```
R0 = 0xAABB(Z);
P0.L = LO(CAN_MB_DATA3(8));
W[P0] = R0;  /* Byte0 = 0xAA, Byte1 = 0xBB */

R0 = 1;
P0.L = LO(CAN_MB_DATA2(8));
W[P0] = R0;  /* Initialize Count to 1 */

/* Initialize Mailbox 9 For Receive */
R0 = 0x007 << 2;  /* Put Message ID in correct slot */
P0.L = LO(CAN_MB_ID1(9));  /* Access MB08 ID1 Register */
W[P0] = R0;       /* Remote frame disabled, 11 bit ID */

R0 = 0;
P0.L = LO(CAN_MB_ID0(9));
W[P0] = R0;  /* Zero Out Lower ID Register */
SSYNC;

/* Enable the Configured Mailboxes */
P0.L = LO(CAN_MC1);
R0 = W[P0](Z);
BITSET(R0, BITPOS(MC8));  /* Enable MB08 */
BITSET(R0, BITPOS(MC9));  /* Enable MB09 */
W[P0] = R0;
SSYNC;
RTS;
```

Initiating CAN Transfers and Processing Interrupts

After the mailboxes are properly set up, transfers can be requested in the CAN controller. This code example initializes the CAN-level interrupts, takes the CAN controller out of configuration mode, requests a transfer, and then waits for and processes CAN TX and RX interrupts. This example assumes that the `CAN_RX_HANDLER` and `CAN_TX_HANDLER` have been properly registered in the system interrupt controller and that the interrupts are enabled properly in the `SIC_IMASK` register.

Listing 17-4. CAN Transfers and Interrupts

```
CAN_SetupIRQs_and_Transfer:
  P0.H = HI(CAN_MBIM1);
  P0.L = LO(CAN_MBIM1);
  R0 = 0;
  BITSET(R0, BITPOS(MBIM8));  /* Enable Mailbox Interrupts */
  BITSET(R0, BITPOS(MBIM9));  /* for Mailboxes 8 and 9 */
  W[P0] = R0;
  SSYNC;

  /* Leave CAN Configuration Mode (Clear CCR) */
  P0.L = LO(CAN_CONTROL);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(CCR));
  W[P0] = R0;

  P0.L = LO(CAN_STATUS);
/* Wait for CAN Configuration Acknowledge (CCA) */
  WAIT_FOR_CCA_TO_CLEAR:
    R1 = W[P0](Z);
    CC = BITTST (R1, BITPOS(CCA));
    IF CC JUMP WAIT_FOR_CCA_TO_CLEAR;
  P0.L = LO(CAN_TRS1);
  R0 = TRS8;   /* Transmit Request MB08 */
  W[P0] = R0;  /* Issue Transmit Request */
  SSYNC;


  Wait_Here_For_IRQs:
    NOP;
    NOP;
    NOP;
    JUMP Wait_Here_For_IRQs;


/* ================================================
** CAN_TX_HANDLER
**
** ISR clears the interrupt request from MB8, writes
** new data to be sent, and requests to send again
```

```
**
** ================================================
*/

CAN_TX_HANDLER:
  [--SP] = (R7:6, P5:5);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P5.H = HI(CAN_MBTIF1);
  P5.L = LO(CAN_MBTIF1);
  R7 = MBTIF8;
  W[P5] = R7;     /* Clear Interrupt Request Bit for MB08 */

  P5.L = LO(CAN_MB_DATA2(8));
  R7 = W[P5](Z);  /* Retrieve Previously Sent Data */

  R6 = 0xFF;      /* Mask Upper Byte to Check Lower */
  R6 = R6 & R7;   /* Byte for Wrap */
  R5 = 0xFF;      /* Check Wrap Condition */

  CC = R6 == R5;  /* Check if Lower Byte Wraps */

  IF CC JUMP HANDLE_COUNT_WRAP;
  R7 += 1;        /* If no wrap, Increment Count */
  JUMP PREPARE_TO_SEND;

  HANDLE_COUNT_WRAP:
    R6 = 0xFF00(Z);  /* Mask Off Lower Byte */
    R7 = R7 & R6;    /* Sets Lower Byte to 0 */
    R6 = 0x0100(Z);  /* Increment Value for Upper Byte */
    R7 = R7 + R6;    /* Increment Upper Byte */

  PREPARE_TO_SEND:
    W[P5] = R7;  /* Set New TX Data */

    P5.L = LO(CAN_TRS1);
    R7 = TRS8;
    W[P5] = R7;    /* Issue New Transmit Request */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:6, P5:5) = [SP++];
  SSYNC;
  RTI;
```

```
/* =================================================
** CAN_RX_HANDLER
**
** ISR clears the interrupt request from MB9, writes
** new data to be sent, and requests to send again
**
** =================================================
*/
CAN_RX_HANDLER:
  [--SP] = (R7:7, P5:4);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P4.H = CAN_RX_WORD;     /* Set Pointer to Storage Element */
  P4.L = CAN_RX_WORD;

  P5.H = HI(CAN_MBRIF1);
  P5.L = LO(CAN_MBRIF1);
  R7 = MBRIF9;
  W[P5] = R7;  /* Clear Interrupt Request Bit for MB09 */

  P5.L = LO(CAN_MB_DATA3(9));
  R7 = W[P5](Z);   /* Read data from mailbox */
  W[P4] = R7;      /* Store data to memory */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:7, P5:4) = [SP++];
  SSYNC;
  RTI;
```

18 SPI-COMPATIBLE PORT CONTROLLER

This chapter describes the serial peripheral interface (SPI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of SPIs for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For SPI DMA channel assignments, refer to Table 7-7 on page 7-107 in Chapter 7, “Direct Memory Access”.

For SPI interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, “System Interrupts”.

To determine how each of the SPIs is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, “General-Purpose Ports”.

For a list of MMR addresses for each SPI, refer to Chapter A, “System MMR Assignments”.

SPI behavior for the ADSP-BF50x that differs from the general information in this chapter can be found in the section “Unique Information for the ADSP-BF50x Processor” on page 18-53.

Overview

The SPI port provides an I/O interface to a wide variety of SPI-compatible peripheral devices.

With a range of configurable options, the SPI port provides a glueless hardware interface with other SPI-compatible devices. SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI-compatible peripheral implementation also supports programmable bit rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Features

The SPI includes these features:

- Full duplex, synchronous serial interface
- Supports 8- or 16-bit word sizes
- Programmable baud rate, clock phase, and polarity
- Supports multimaster environments
- Integrated DMA controller
- Double-buffered transmitter and receiver
- One SPI device select input and multiple chip select outputs
- Programmable shift direction of MSB or LSB first
- Interrupt generation on mode fault, overflow, and underflow
- Shadow register to aid debugging

Typical SPI-compatible peripheral devices that can be used to interface to the SPI-compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays
- Shift registers
- FPGAs with SPI emulation

Interface Overview

Figure 18-1 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCK rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted serially into the other end of the same shift register). The SCK synchronizes the shifting and sampling of the data on the two serial data pins.

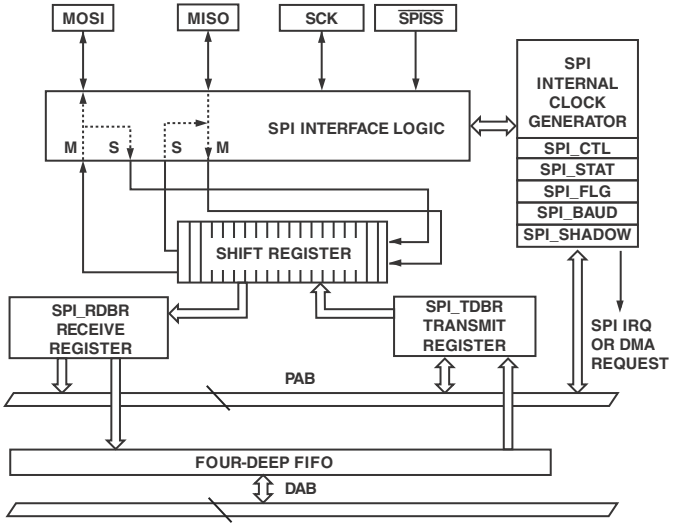


Figure 18-1. SPI Block Diagram

External Interface

SPI Clock Signal (SCK)

The SCK signal is the serial clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of bit rates. The SCK signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCK is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the SPISS input is driven inactive (high).

The SCK is used to shift out and shift in the data driven on the MISO and MOSI lines. Clock polarity and clock phase relative to data are programmable in the SPI_CTL register and define the transfer format.

Master-Out, Slave-In (MOSI) Signal

The master-out, slave-in (MOSI) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI pin transmits data out. If the processor is configured as a slave, the MOSI pin receives data in. In an SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

Master-In, Slave-Out (MISO) Signal

The master-in, slave-out (MISO) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MISO pin receives data in. If the processor is configured as a slave, the MISO pin transmits data out. In an SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 18-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

The processor can be booted through its SPI interface to allow user application code and data to be downloaded before runtime.

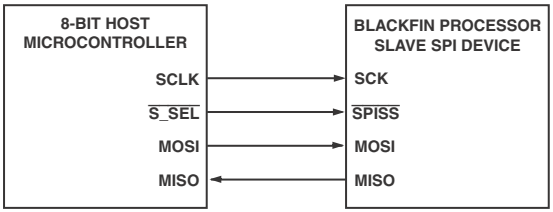


Figure 18-2. ADSP-BF50x Processor as Slave SPI Device

SPI Slave Select Input Signal (SPISS)

The SPISS signal is the SPI slave select input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as an error signal input in a multimaster environment. In multimaster mode, if the SPISS input signal of a master is asserted (driven low), and the PSSE bit in the SPI_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCK period. See Figure 18-3. The minimum time between successive word transfers (T4) is two SCK periods. This is measured from the last active edge of SCK of one word to the first active edge of SCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SCK period. In this case, T1 and T2 will each always be equal to one-half the SCK period.

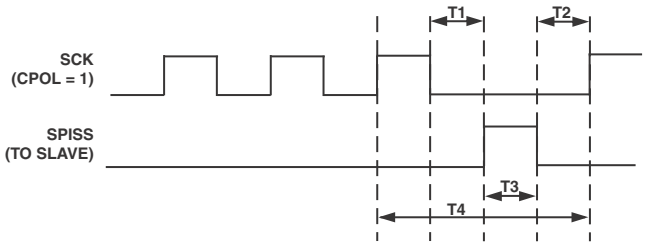


Figure 18-3. SPI Timing

SPI Slave Select Enable Output Signals

When operating in master mode, Blackfin processors may use any GPIO pin to enable individual SPI slave devices by software. In addition, the SPI module provides hardware support to generate up to seven slave select enable signals automatically (depending upon the configuration of the specific processor). See Figure 18-14 on page 18-37 for details.

These signals are always active low in the SPI protocol. Since the respective pins are not driven during reset, it is recommended to pull them up by a resistor.

If enabled as a master, the SPI uses the `SPI_FLG` register to enable general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the port pins that are to be used as SPI slave-select outputs must first be configured as such. To work as SPI output pins, the port pins must be enabled for use by SPI in the appropriate `PORT_MUX` register.

In slave mode, the `SPI_FLG` bits have no effect, and each SPI uses the `SPISS` input as a slave select. Just as in the master mode case, the port pin associated with `SPISS` must first be configured appropriately before use. Figure 18-14 on page 18-37 shows the `SPI_FLG` register diagram.

Slave Select Inputs

If the SPI is in slave mode, 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 multimaster environment. The PSSE bit in SPI_CTL enables this feature. When PSSE = 1, the SPISS input is the master mode error input. Otherwise, SPISS is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a master processor equipped with seven slave selects, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The seven port pins that can be configured as SPI master mode slave-select output pins can be connected to each of the slave SPI device's SPISS pins. In this configuration, the FLSx bits in SPI_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected through SPI ports can be other processors.
3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data

to all seven at the same time. This EMISO feature may be available in some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 18-4 shows one processor as a master with three processors (or other SPI-compatible devices) as slaves.

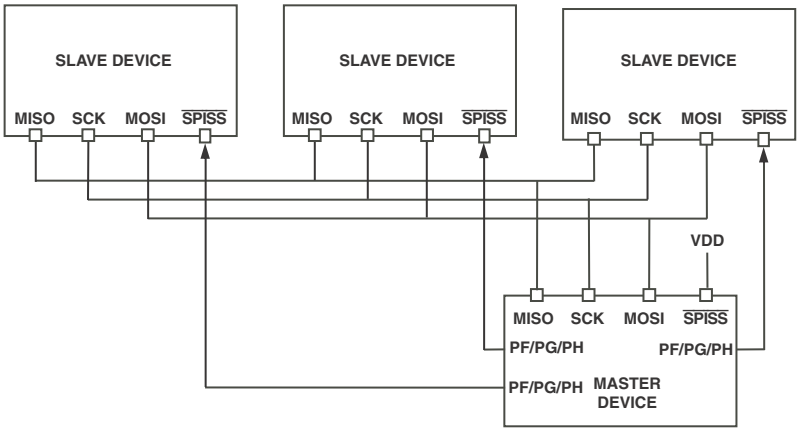


Figure 18-4. Single-Master, Multiple-Slave Configuration

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit in the SPI_STAT register is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit in the SPI_STAT register are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

Internal Interfaces

The SPI has dedicated connections to the processor's peripheral bus (PAB) and DAB.

The low-latency PAB bus is used to map the SPI resources into the system MMR space. For PAB accesses to SPI MMRs, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the peripheral bus are two SCLK cycles.

The DAB bus provides a means for DMA SPI transfers to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory. The SPI peripheral, as a DMA master, is capable of sourcing DMA accesses. The arbitration policy for access to the DAB is described in the *Chip Bus Hierarchy* chapter.

DMA Functionality

The SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data will essentially be ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DAB.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

Description of Operation

The following sections describe the operation of the SPI.

SPI Transfer Protocols

The SPI protocol supports four different combinations of serial clock phase and polarity (SPI modes 0, 1, 2, 3). These combinations are selected using the CPOL and CPHA bits in SPI_CTL as shown in Figure 18-5.

Figure 18-6 on page 18-13 and Figure 18-7 on page 18-13 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the SPISS signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with the most significant bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPI_CTL is allowed. For example, a 16-bit transfer with the least significant bit (LSB) first is another possible configuration.

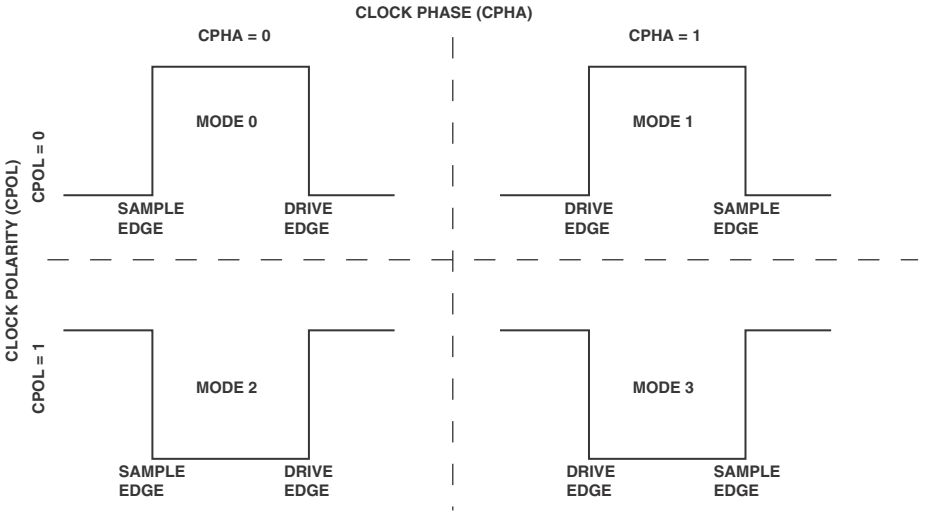


Figure 18-5. SPI Modes of Operation

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, SPISS, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, SPISS may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software through manipulation of the SPI_FLG register.

Figure 18-6 shows the SPI transfer protocol for CPHA = 0. Note SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

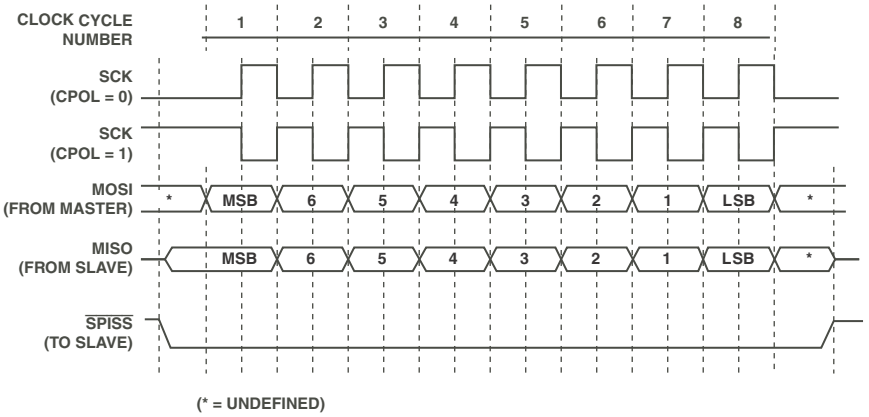


Figure 18-6. SPI Transfer Protocol for CPHA = 0

Figure 18-7 shows the SPI transfer protocol for CPHA = 1. Note SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

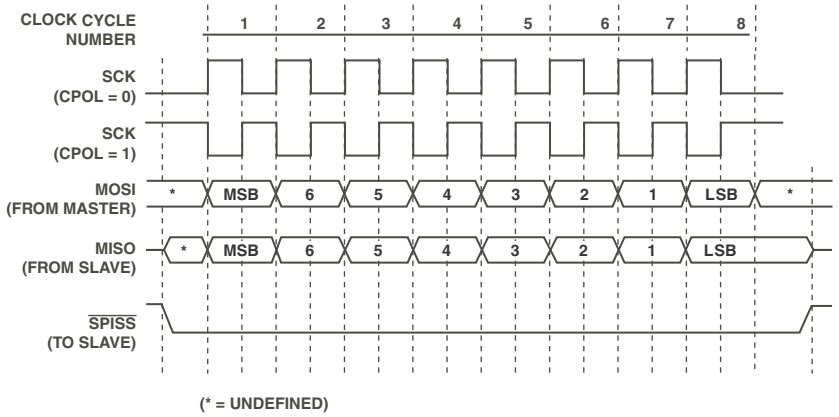


Figure 18-7. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in single master as well as multimaster environments. The MOSI, MISO, and the SCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode has been selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the MISO line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSI and MISO pins when this option is selected.

The WOM bit in the SPI_CTL register controls this option. When WOM is set and the SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic high. The MOSI pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISO pin is three-stated if the data driven out on MISO is a logic high.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal (SPISS). The other SPI device acts as

the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected through their SPI ports, all `MOSI` pins are connected together, all `MISO` pins are connected together, and all `SCK` pins are connected together.

For a multislave environment, the processor can make use of up to seven programmable flags that are dedicated SPI slave select signals for the SPI slave devices.

At reset, the SPI is disabled and configured as a slave.

Clock Signals

The SCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value in the SPI_BAUD register. For slave devices, the value in SPI_BAUD is ignored. When the SPI device is a master, SCK is an output signal. When the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The `SCK` signal is used to shift out and shift in the data driven onto the `MISO` and `MOSI` lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and

clock phase relative to data are programmable in the SPI_CTL register and define the transfer format. See Figure 18-5 on page 18-12.

Interrupt Output

The SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the TIMOD field in the SPI_CTL register. In DMA mode (TIMOD = b#1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = b#11) or read from (TIMOD = b#10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPI_TDBR register is ready to be written to (TIMOD = b#01) or when the SPI_RDBR register is ready to be read from (TIMOD = b#00).

An SPI error interrupt is generated in a master when a mode fault error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = b#11) or an overflow (RBSY when TIMOD = b#10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPI_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control (SPI_CTL) Register" on page 18-35.

Functional Description

The following sections describe the functional operation of the SPI.

Master Mode Operation (Non-DMA)

When the SPI is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to the appropriate port register(s) to properly configure the SPI interface for master mode operation.The required pins are configured for SPI use as slave-select outputs.
2. The core writes to SPI_FLG, setting one or more of the SPI flag select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.
3. The core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.
4. If the CPHA bit in the SPI_CTL register = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI_FLG.
5. The TIMOD bits in SPI_CTL determine the SPI transfer initiate mode. The transfer on the SPI link begins upon either a data write by the core to the SPI_TDBR register or a data read of the SPI_RDBR register.
6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before a shift, the shift register is loaded with the contents of the SPI_TDBR register. At the end of the transfer, the contents of the shift register are loaded into the SPI_RDBR register.
7. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initiate mode.

See Figure 18-8 on page 18-29 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL.

If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros on the MOSI pin. One word is transmitted for each new transfer initiate command. If SZ = 0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty.

If GM = 1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two TIMOD bits of SPI_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of the SPI_RDBR register or a write to the SPI_TDBR register. This is summarized in Table 18-1.

If the SPI port is enabled with TIMOD = b#01 or TIMOD = b#11, the hardware immediately issues a first interrupt or DMA request.

Table 18-1. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
b#00	Transmit and receive	Initiate new single word transfer upon read of SPI_RDBR and previous transfer completed.	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Initiate new single word transfer upon write to SPI_TDBR and previous transfer completed.	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA read of SPI_RDBR, and last transfer completed.	Request DMA reads as long as the SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPI_TDBR, and last transfer completed.	Request DMA writes as long as the SPI DMA FIFO is not full.

Slave Mode Operation (Non-DMA)

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 SPISS select signal to the active state (low), or by the first active edge of the clock (SCK), depending on the state of the CPHA bit in the SPI_CTL register.

These steps illustrate SPI operation in the slave mode:

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The core writes to SPI_CTL to define the mode of the serial link to be the same as the mode set up in the SPI master.
3. To prepare for the data transfer, the core writes data to be transmitted into SPI_TDBR.
4. Once the SPISS falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCK edges, depending upon the states of CPHA and CPOL.
5. Reception/transmission continues until SPISS is released or until the slave has received the proper number of clock cycles.
6. The slave device continues to receive/transmit with each new falling edge transition on SPISS and/or SCK clock edge.

See Figure 18-8 on page 18-29 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros 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 SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and the SPI_RDBR register is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 18-2 are necessary to prepare the device for a new transfer.

Table 18-2. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and receive	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Programming Model

The following sections describe the SPI programming model.

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, which CPHA mode is selected, and which transfer initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPI_TDBR is written to or SPI_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA = 0, the transfer starts as soon as the SPISS input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered

finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPI_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCK. For a slave SPI, RXS is set shortly after the last SCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPI_RDBR is full (TIMOD = b#00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF is cleared shortly after the start of a transfer (SPISS going low for CPHA = 0, first active edge of SCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a transfer (either by writing the SPI_TDBR or reading the SPI_RDBR, depending on TIMOD), and is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPI_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPI_RDBR, because of the latency. Therefore, for SPI_BAUD = 2 or SPI_BAUD = 3, RXS must be set before SPIF to read SPI_RDBR. For larger SPI_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = b#00 mode may be the best operation option. In this mode,

software performs a dummy read from the SPI_RDBR register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPI_TDBR register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the SZ bit in the SPI_CTL register to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the SPI_RDBR register does not initiate another transfer. It is recommended that the SPI port be disabled before the final SPI_RDBR read access. Reading the SPI_SHADOW register is not sufficient, as it does not clear the interrupt request.

In master mode with the CPHA bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in TIMOD = b#00 or TIMOD = b#10 mode. With TIMOD = b#01 or TIMOD = b#11, the interrupt is requested while the transfer is still in progress.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI interface operates as follows.

1. The core writes to the appropriate port register(s) to properly configure the SPI for master mode operation. The appropriate pins can be configured for SPI use as slave-select outputs.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and to configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.

3. The processor core writes to the SPI_FLG register, setting one or more of the SPI flag select bits (FLSx).

4. The processor core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "receive with DMA" (TIMOD = b#10) or "transmit with DMA" (TIMOD = b#11) mode.

5. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI_TDBR register, it initiates a transfer on the SPI link.

6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. For receive transfers, the value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

7. In receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from "1" to "0". The SPI continues receiving words until SPI DMA mode is disabled.

In transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI continues transmitting words until the SPI DMA FIFO is empty.

See Figure 18-9 on page 18-30 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ = 1, the device repeatedly transmits zeros on the MOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode. The TXE underrun condition cannot happen in this mode (master DMA TX mode), because the master SPI will not initiate a transfer if there is no data in the DMA FIFO.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes

to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

Slave Mode DMA Operation

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the SPISS signal to the active-low state or by the first active edge of SCK, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.
3. The processor core writes to the SPI_CTL register to define the mode of the serial link to be the same as the mode set up in the SPI master. The TIMOD field will be configured to select either “receive with DMA” (TIMOD = b#10) or “transmit with DMA” (TIMOD = b#11) mode.

4. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. As the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPI_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

5. In receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues receiving words on SCK edges as long as the slave select input is active.

   In transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues transmitting words on SCK edges as long as the slave select input is active.

See Figure 18-9 on page 18-30 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits zeros on the MISO pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits zeros on the MISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10), or when there is a TXE underflow error condition (when TIMOD = b#11).

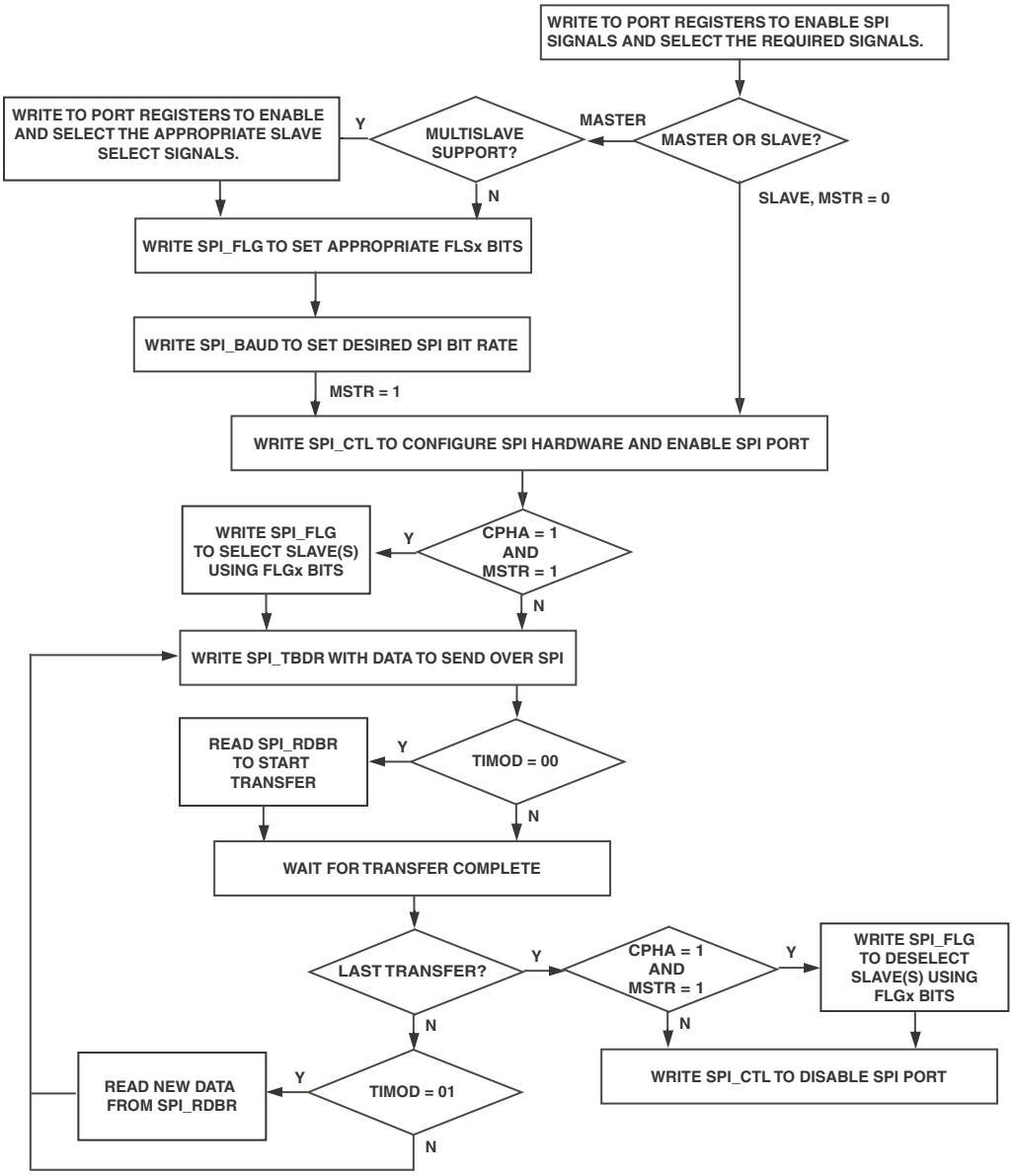


Figure 18-8. Core-Driven SPI Flow Chart

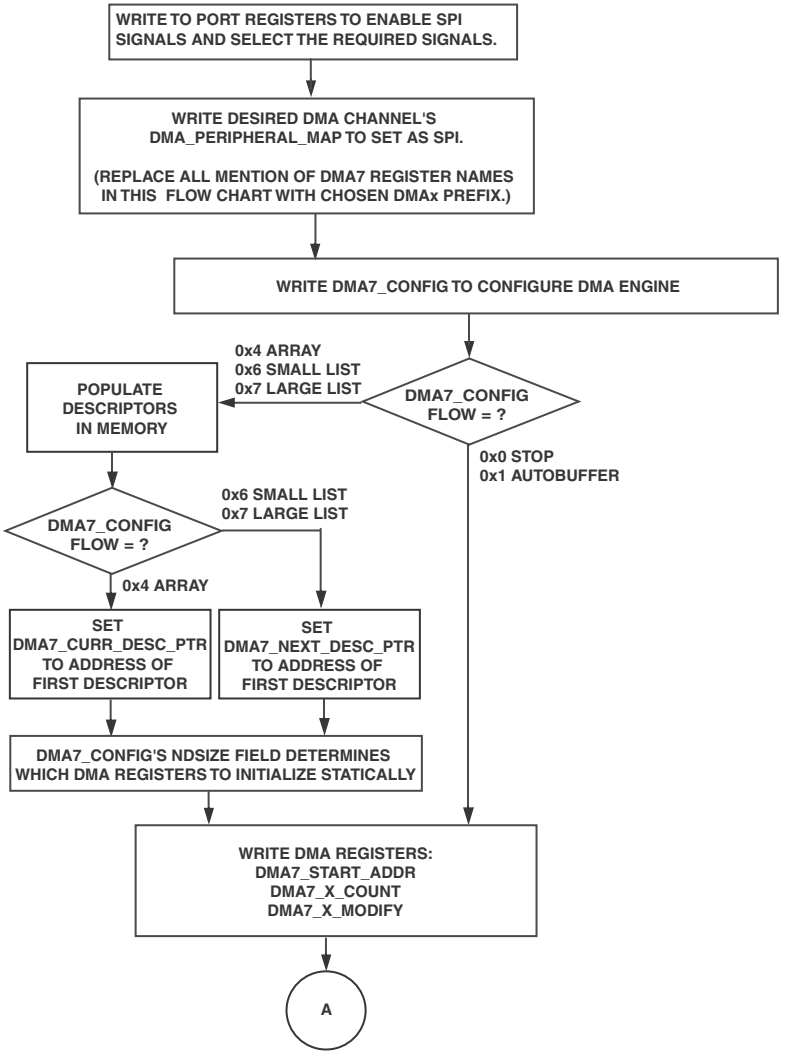


Figure 18-9. SPI DMA Flow Chart (Part 1 of 3)

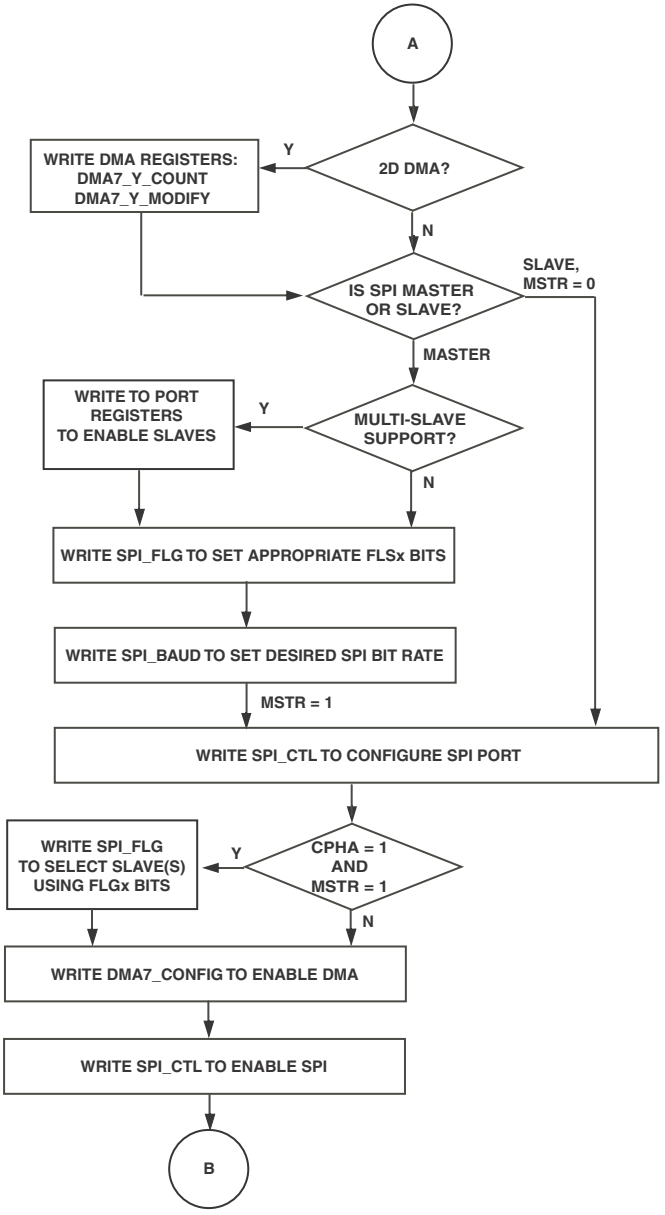


Figure 18-10. SPI DMA Flow Chart (Part 2 of 3)

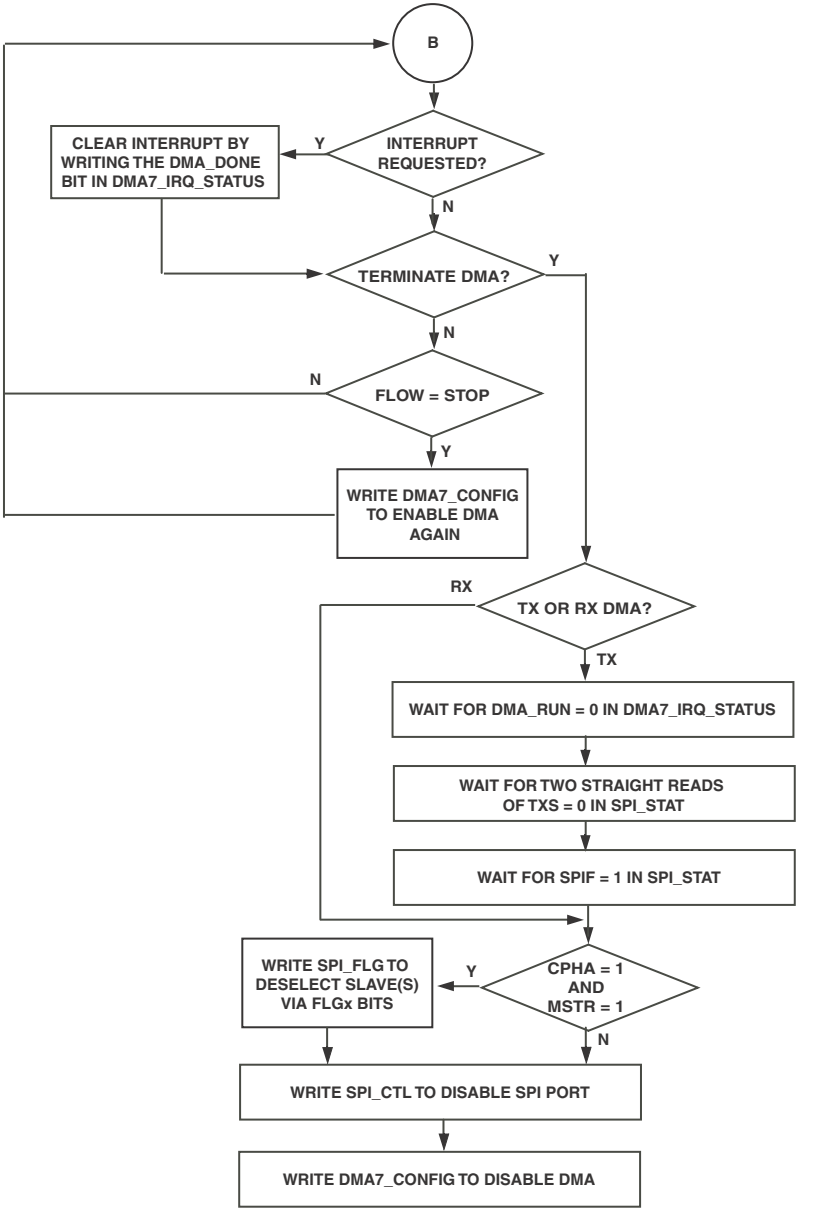


Figure 18-11. SPI DMA Flow Chart (Part 3 of 3)

SPI Registers

The SPI peripheral includes a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPI_BAUD, SPI_CTL, SPI_FLG, and SPI_STAT. Two registers are used for buffering receive and transmit data: SPI_RDBR and SPI_TDBR. The shift register, SFDR, is internal to the SPI module and is not directly accessible.

Table 18-3 shows the functions of the SPI registers. Figure 18-12 through Figure 18-18 provide details.

Table 18-3. SPI Register Mapping

Register Name	Function	Notes
SPI_BAUD	SPI port baud control	Value of “0” or “1” disables the serial clock
SPI_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPI_FLG	SPI port flag	Bits 0 and 8 are reserved
SPI_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPI_CTL
SPI_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPI_CTL)
SPI_RDBR	SPI port receive data buffer	When register is read, hardware events can be triggered
SPI_SHADOW	SPI port data	Register has the same contents as SPI_RDBR, but no action is taken when it is read

SPI Baud Rate (SPI_BAUD) Register

The SPI_BAUD register is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

SCK frequency = (peripheral clock frequency SCLK)/(2 × SPI_BAUD)

Writing a value of “0” or “1” to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

Table 18-4 lists several possible baud rate values for SPI_BAUD.

Table 18-4. SPI Master Baud Rate Example

SPI_BAUD Decimal Value	SPI Clock (SCK) Divide Factor	Baud Rate for SCLK at 100 MHz
0	N/A	N/A
1	N/A	N/A
2	4	25 MHz
3	6	16.7 MHz
4	8	12.5 MHz
65,535 (0xFFFF)	131,070	763 Hz

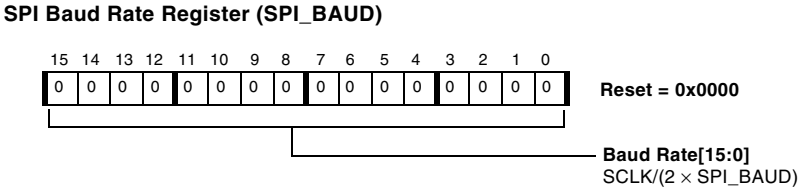


Figure 18-12. SPI Baud Rate Register

SPI Control (SPI_CTL) Register

The SPI_CTL register is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPI_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to b#00, a SPI port transaction is begun when the receive buffer is read. Data from the first read will need to be discarded since the read is needed to initiate the first SPI port transaction. When set to b#01, the transaction is initiated when the transmit buffer is written. A value of b#10 selects DMA receive mode and the first transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPI_RDBR register. A value of 11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPI_TDBR register.

The PSSE bit is used to enable the SPISS input for an external master. When not used, SPISS can be disabled, freeing up a pin for an alternate function.

The EMISO bit enables the MISO pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the SPI_STAT register is set. See “Mode Fault Error (MODF)” on page 18-40.

Figure 18-13 provides the bit descriptions for SPI_CTL.

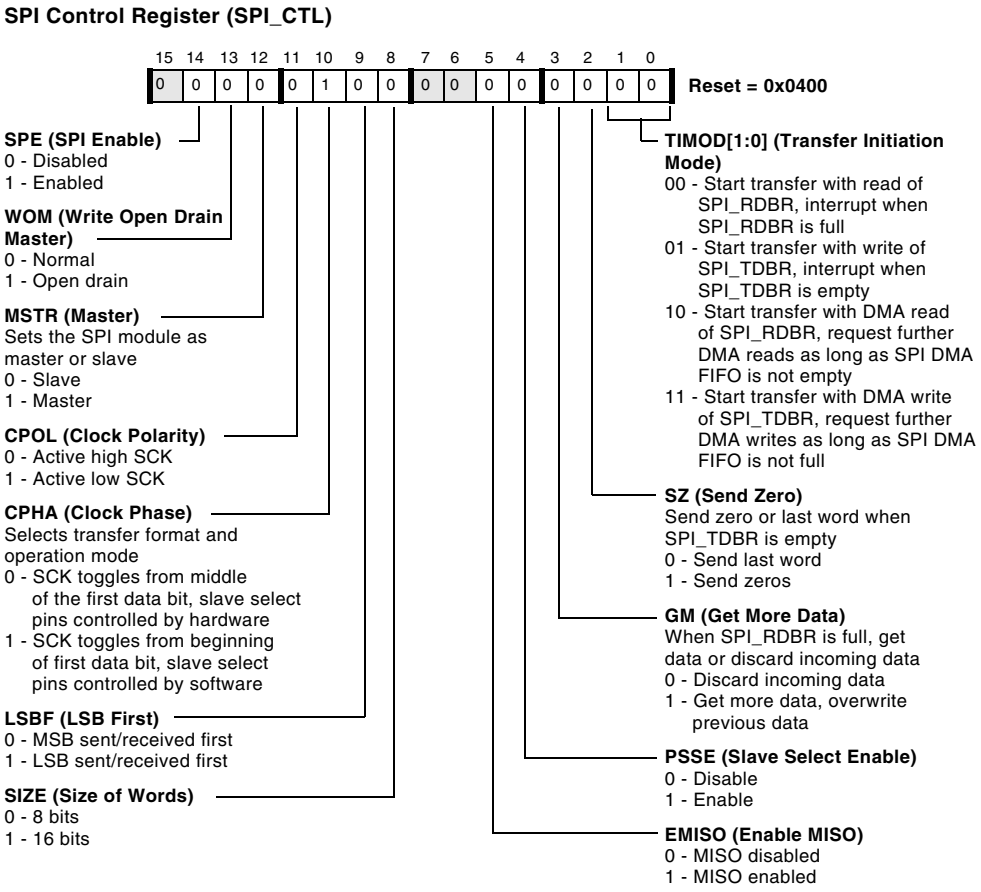


Figure 18-13. SPI Control Register

SPI Flag (SPI_FLG) Register

The SPI_FLG register consists of two sets of bits that function as follows.

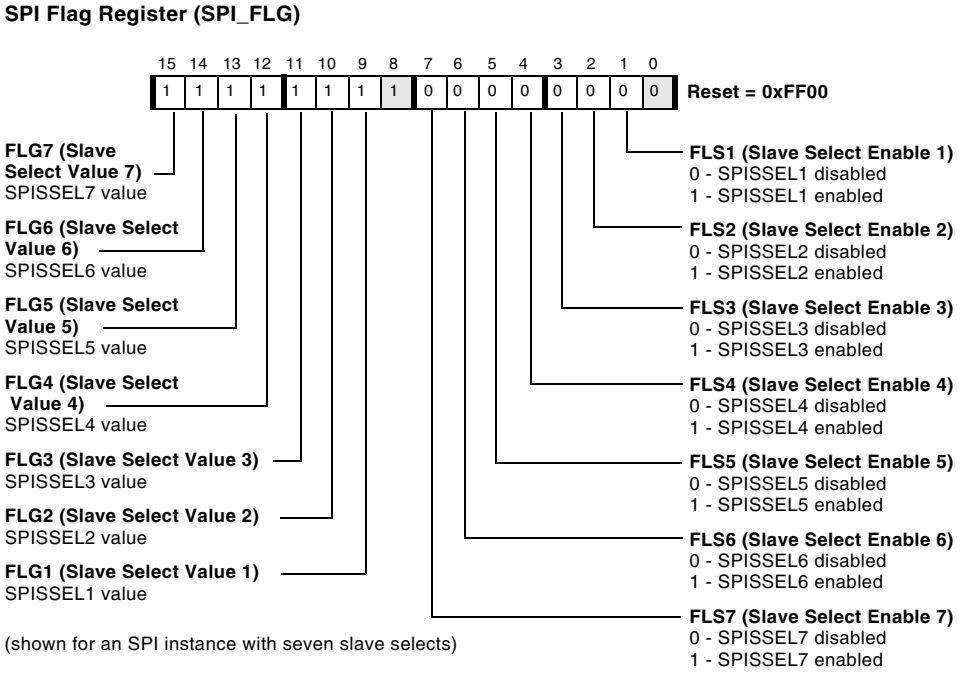


Figure 18-14. SPI Flag Register (example with 7 slave selects)

- Slave select enable (FLSx) bits

  Each FLSx bit corresponds to a general purpose port pin. When an FLSx bit is set, the corresponding port pin is driven as a slave select. For example, if FLS1 is set in SPI_FLG, the port pin corresponding to SPISSEL1 is driven as a slave select.

If the FLSx bit is not set, the general-purpose port registers configure and control the corresponding port pins.

- Slave select value (FLGx) bits

  When a port pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPI_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, setting FLS3 in the SPI_FLG register drives the SPISSEL3 pin as a slave select. Then, clearing FLG3 in the SPI_FLG register drives the pin low, and setting FLG3 drives it high. The pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, the pin remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use the slave select function on a port pin to which it is mapped, it is only necessary to set the appropriate FLS bit in SPI_FLG. It is not necessary to write to an FLG bit, because the SPI hardware automatically drives the port pin.

SPI Status (SPI_STAT) Register

The SPI_STAT register is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

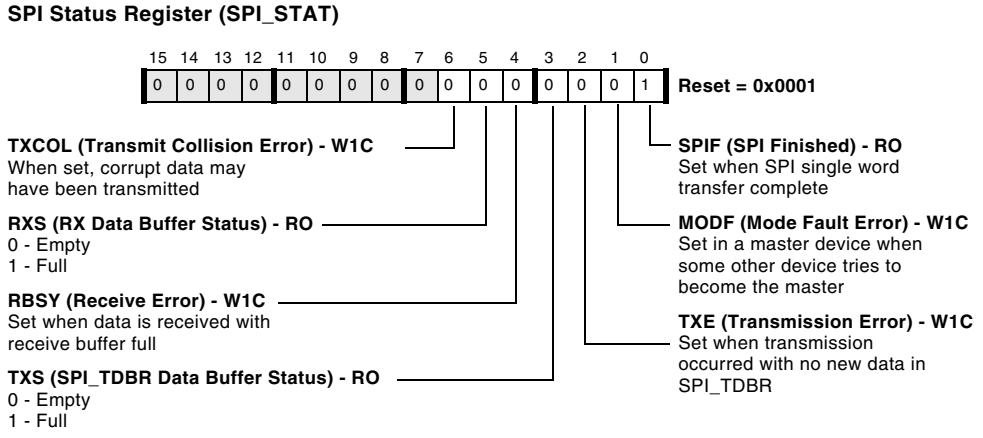


Figure 18-15. SPI Status Register

Some of the bits in SPI_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a "1" to the desired bit position of SPI_STAT. For example, if the TXE bit is set, the user must write a "1" to bit 2 of SPI_STAT to clear the TXE error condition. This allows the user to read SPI_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

See Figure 18-15 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPI_STAT when the SPISS input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPI_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPI_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPI_CTL is cleared, disabling the SPI system
- The MODF status bit in SPI_STAT is set
- An SPI error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the SPISS input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another mode fault error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave select output pins revert to being controlled by the general-purpose I/O port registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the general-purpose I/O port registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the port pins that will be used as slave selects. Programs can do this by

configuring the direction of the port pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as port pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPI_STAT when all the conditions of transmission are met, and there is no new data in SPI_TDBR (SPI_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPI_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPI_STAT register when a new transfer is completed, but before the previous data can be read from SPI_RDBR. The state of the GM bit in the SPI_CTL register determines whether SPI_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPI_STAT when a write to SPI_TDBR coincides with the load of the shift register. The write to SPI_TDBR can be by software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPI_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

SPI Transmit Data Buffer (SPI_TDBR) Register

The SPI_TDBR register is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPI_TDBR is loaded into the internal shift register SFDR. A read of SPI_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPI_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPI_TDBR are repeatedly transmitted. A write to SPI_TDBR is permitted in this mode, and this data is transmitted.

If the SZ control bit in the SPI_CTL register is set, SPI_TDBR may be reset to zero under certain circumstances.

If multiple writes to SPI_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPI_TDBR are transmitted. Multiple writes to SPI_TDBR are possible, but not recommended.

SPI Transmit Data Buffer Register (SPI_TDBR)

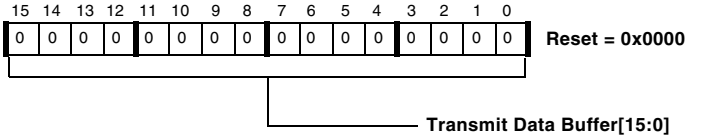


Figure 18-16. SPI Transmit Data Buffer Register

SPI Receive Data Buffer (SPI_RDBR) Register

The SPI_RDBR register is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPI_RDBR. During a DMA receive operation, the data in SPI_RDBR is automatically read by the DMA controller. When SPI_RDBR is read by software, the RXS bit in the

SPI_STAT register is cleared and an SPI transfer may be initiated (if TIMOD = b#00).

SPI Receive Data Buffer Register (SPI_RDBR)
Read Only

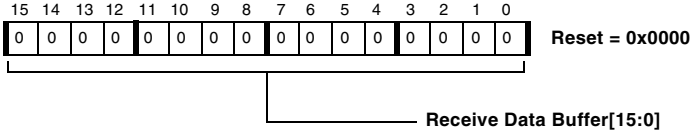


Figure 18-17. SPI Receive Data Buffer Register

SPI RDBR Shadow (SPI_SHADOW) Register

The SPI_SHADOW register is provided for use in debugging software. This register is at a different address than the receive data buffer, SPI_RDBR, but its contents are identical to that of SPI_RDBR. When a software read of SPI_RDBR occurs, the RXS bit in SPI_STAT is cleared and an SPI transfer may be initiated (if TIMOD = b#00 in SPI_CTL). No such hardware action occurs when the SPI_SHADOW register is read. The SPI_SHADOW register is read-only.

SPI RDBR Shadow Register (SPI_SHADOW)
Read Only

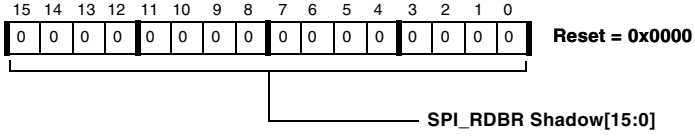


Figure 18-18. SPI RDBR Shadow Register

Programming Examples

This section includes examples (Listing 18-1 through Listing 18-8) for both core-generated and DMA-based transfers. Each code example assumes that the appropriate processor header files are included.

Core-Generated Transfer

The following core-driven master-mode SPI example shows how to initialize the hardware, signal the start of a transfer, handle the interrupt and issue the next transfer, and generate a stop condition.

Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 18-1. SPI Register Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7);      /* FLS7 */
  W[P0] = R0;           /* Enable slave-select output pin */

  P0.H = hi(SPI_BAUD);
  P0.L = lo(SPI_BAUD);
  R0.L = 0x208E;        /* Write to SPI Baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133 MHz, SPI clock ~= 8 kHz
*/

/* Setup SPI Control Register */
/***********************************************
```

```
* TIMOD [1:0] = 00 : Transfer On RDBR Read.
* SZ [2]      = 0 : Send Last Word When TDBR Is Empty
* GM [3]      = 1 : Overwrite Previous Data If RDBR Is Full
* PSSE [4]    = 0 : Disables Slave-Select As Input (Master)
* EMISO [5]   = 0 : MISO Disabled For Output (Master)
* [7] and [6] = 0 : RESERVED
* SIZE [8]    = 1 : 16 Bit Word Length Select
* LSBF [9]    = 0 : Transmit MSB First
* CPHA [10]   = 0 : Hardware Controls Slave-Select Outputs
* CPOL [11]   = 1 : Active Low SCK
* MSTR [12]   = 1 : Device Is Master
* WOM [13]    = 0 : Normal MOSI/MISO Data Output (No Open Drain)
* SPE [14]    = 1 : SPI Module Is Enabled
* [15]        = 0 : RESERVED
*********************************************/
 P0.H = hi(SPI_CTL) ;
 P0.L = lo(SPI_CTL) ;
 R0 = 0x5908;
 W[P0] = R0.L; ssync;   /* Enable SPI as MASTER */
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following a dummy read of `SPI_RDBR`. Typically, known data which is desired to be transmitted to the slave is preloaded into the `SPI_TDBR`. In the following code, `P1` is assumed to point to the start of the 16-bit transmit data buffer and `P2` is assumed to point to the start of the 16-bit receive data buffer. In addition, the user must ensure appropriate interrupts are enabled for SPI operation.

Listing 18-2. Initiate Transfer

```
Initiate_Transfer:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITCLR (R0,0xF);      /* FLG7 */
  W[P0] = R0;           /* Drive 0 on enabled slave-select pin */

  P0.H = hi(SPI_TDBR);  /* SPI Transmit Register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++] (z);     /* Get First Data To Be Transmitted
  And Increment Pointer */
  W[P0] = R0;            /* Write to SPI_TDBR */

  P0.H = hi(SPI_RDBR);
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z); /* Dummy read of SPI_RDBR kicks off transfer */
```

Post Transfer and Next Transfer

Following the transfer of data, the SPI generates an interrupt, which is serviced if the interrupt is enabled during initialization. In the interrupt routine, software must write the next value to be transmitted prior to reading the byte received. This is because a read of the `SPI_RDBR` initiates the next transfer.

Listing 18-3. SPI Interrupt Handler

```
SPI_Interrupt_Handler:
Process_SPI_Sample:
  P0.H = hi(SPI_TDBR);     /* SPI transmit register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++](z);         /* Get next data to be transmitted */
```

```
  W[P0] = R0.l;            /* Write that data to SPI_TDBR */

Kick_Off_Next:
  P0.H = hi(SPI_RDBR);     /* SPI receive register */
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z);   /* Read SPI receive register (also kicks off
next transfer) */
  W[P2++] = R0;            /* Store received data to memory */
  RTI;                     /* Exit interrupt handler */
```

Stopping

In order for a data transfer to end after the user has transferred all data, the following code can be used to stop the SPI. Note that this is typically done in the interrupt handler to ensure the final data has been sent in its entirety.

Listing 18-4. Stopping SPI

```
Stopping_SPI:
  P0.H = hi(SPI_CTL);
 P0.L = lo(SPI_CTL);
  R0 = W[P0];
  BITCLR(R0, 14);         /* Clear SPI enable bit */
  W[P0] = R0.L; ssync;    /* Disable SPI */
```

DMA-Based Transfer

The following DMA-driven master-mode SPI autobuffer example shows how to initialize DMA, initialize SPI, signal the start of a transfer, and generate a stop condition.

DMA Initialization Sequence

The following code initializes the DMA to perform a 16-bit memory read DMA operation in autobuffer mode, and generates an interrupt request when the buffer has been sent. This code assumes that `P1` points to the start of the data buffer to be transmitted and that `NUM_SAMPLES` is a defined macro indicating the number of elements being sent.

Listing 18-5. DMA Initialization

```
Initialize_DMA:     /* Assume DMA7 as the channel for SPI DMA */
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R0 = 0x1084(z);   /* Autobuffer mode, IRQ on complete, linear
16-bit, mem read */
  w[P0] = R0;

  P0.H = hi(DMA7_START_ADDR);
  P0.L = lo(DMA7_START_ADDR);
  [p0] = p1;     /* Start address of TX buffer */

  P0.H  = hi(DMA7_X_COUNT);
  P0.L  = lo(DMA7_X_COUNT);
  R0    = NUM_SAMPLES;
  w[p0] = R0;   /* Number of samples to transfer */

  R0 = 2;
  P0.H = hi(DMA7_X_MODIFY);
  P0.L = lo(DMA7_X_MODIFY);
  w[p0] = R0;   /* 2 byte stride for 16-bit words */

  R0 = 1;       /* single dimension DMA means 1 row */
  P0.H = hi(DMA7_Y_COUNT);
  P0.L = lo(DMA7_Y_COUNT);
  w[p0] = R0;
```

SPI Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 18-6. SPI Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7);    /* FLS7 */
  W[P0] = R0;         /* Enable slave-select output pin */

  P1.H = hi(SPI_BAUD);
  P1.L = lo(SPI_BAUD);
  R0.L = 0x208E;      /* Write to SPI baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133MHz, SPI clock ~= 8kHz */

  /* Setup SPI Control Register */
/************************************************
 * TIMOD [1:0] = 11 : Transfer on DMA TDBR write
 * SZ [2]      =  0 : Send last word when TDBR is empty
 * GM [3]      =  1 : Discard incoming data if RDBR is full
 * PSSE [4]    =  0 : Disables slave-select as input (master)
 * EMISO [5]   =  0 : MISO disabled for output (master)
 * [7] and [6] =  0 : RESERVED
 * SIZE [8]    =  1 : 16 Bit word length select
 * LSBF [9]    =  0 : Transmit MSB first
 * CPHA [10]   =  0 : SCK starts toggling at START of first data
bit
 * CPOL [11]   =  1 : Active HIGH serial clock
 * MSTR [12]   =  1 : Device is master
 * WOM [13]    =  0 : Normal MOSI/MISO data output (no open
drain)
```

```
 * SPE [14]    =  1 : SPI module is enabled
 * [15]        =  0 : RESERVED
 ************************************************/
  /* Configure SPI as MASTER */
  R1 = 0x190B(z);     /* Leave disabled until DMA is enabled */
  P1.L = lo(SPI_CTL);
  W[P1] = R1; ssync;
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following enabling of SPI. However, the DMA must be enabled before enabling the SPI.

Listing 18-7. Starting a Transfer

```
Initiate_Transfer:
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R2 = w[P0](z);
  BITSET (R2, 0);   /*Set DMA enable bit */
  w[p0] = R2.L;     /* Enable TX DMA */

  P4.H = hi(SPI_CTL);
  P4.L = lo(SPI_CTL);
  R2=w[p4](z);
  BITSET (R2, 14);  /* Set SPI enable bit */
  w[p4] = R2;       /* Enable SPI */
```

Stopping a Transfer

In order for a data transfer to end after the DMA has transferred all required data, the following code is executed in the SPI DMA interrupt handler. The example code below clears the DMA interrupt, then waits for the DMA engine to stop running. When the DMA engine has

completed, SPI_STAT is polled to determine when the transmit buffer is empty. If there is data in the SPI Transmit FIFO, it is loaded as soon as the TXS bit clears. A second consecutive read with the TXS bit clear indicates the FIFO is empty and the last word is in the shift register. Finally, polling for the SPIF bit determines when the last bit of the last word has been shifted out. At that point, it is safe to shut down the SPI port and the DMA engine.

Listing 18-8. Stopping a Transfer

```
SPI_DMA_INTERRUPT_HANDLER:
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = 1 ;
  W[P0] = R0 ;  /* Clear DMA interrupt */

  /* Wait for DMA to complete */
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = DMA_RUN;      /* 0x08 */

CHECK_DMA_COMPLETE:  /* Poll for DMA_RUN bit to clear */
  R3 = W[P0] (Z);
  R1 = R3 & R0;
  CC = R1 == 0;
  IF !CC JUMP CHECK_DMA_COMPLETE;

  /* Wait for TXS to clear */
  P0.L = lo(SPI_STAT);
  P0.H = hi(SPI_STAT);
  R1 = TXS;   /* 0x08 */

Check_TXS:    /* Poll for TXS = 0 */
  R2 = W[P0] (Z);
```

```
    R2 = R2 & R1;
    CC = R0 == 0;
    IF !CC JUMP Check_TXS;

    R2 = W[P0] (Z);  /* Check if TXS stays clear for 2 reads */
    R2 = R2 & R1;
    CC = R0 == 0;
    IF !CC JUMP Check_TXS;

/* Wait for final word to transmit from SPI */
Final_Word:
    R0 = W[P0](Z);
    R2 = SPIF;   /* 0x01 */
    R0 = R0 & R2;
    CC = R0 == 0;
    IF CC JUMP Final_Word;

Disable_SPI:
    P0.L = lo(SPI_CTL);
    P0.H = hi(SPI_CTL);
    R0 = W[P0] (Z);
    BITCLR (R0,0xe);   /* Clear SPI enable bit */
    W[P0] = R0;        /* Disable SPI */

Disable_DMA:
    P0.L = lo(DMA7_CONFIG);
    P0.H = hi(DMA7_CONFIG);
    R0 = W[P0](Z);
    BITCLR (R0,0x0);   /* Clear DMA enable bit */
    W[P0] = R0;        /* Disable DMA */

    RTI;   /* Exit Handler */
```

Unique Information for the ADSP-BF50x Processor

None.

19 SPORT CONTROLLER

This chapter describes the synchronous serial peripheral port (SPORT). Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of SPORTs for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For SPORT DMA channel assignments, refer to Table 7-7 on page 7-107 in Chapter 7, "Direct Memory Access".

For SPORT interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the SPORTs is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each SPORT, refer to Chapter A, "System MMR Assignments".

SPORT behavior for the ADSP-BF50x that differs from the general information in this chapter can be found at the end of this chapter in the

section “Unique Information for the ADSP-BF50x Processor” on page 19-76.

Overview

Unlike the SPI interface which has been designed for SPI-compatible communication only, the SPORT modules support a variety of serial data communication protocols, for example:

- A-law or µ-law companding according to G.711 specification
- Multichannel or time-division-multiplexed (TDM) modes
- Stereo audio I^2S mode
- H.100 telephony standard support

In addition to these standard protocols, the SPORT module provides modes to connect to standard peripheral devices, such as ADCs or codecs, without external glue logic. With support for high data rates, independent transmit and receive channels, and dual data paths, the SPORT interface is a perfect choice for direct serial interconnection between two or more processors in a multiprocessor system. Many processors provide compatible interfaces, including processors from Analog Devices and other manufacturers.

Each SPORT has its own set of control registers and data buffers.

Features

A SPORT can operate at up to ½ the system clock (SCLK) rate for an internally generated or external serial clock. The SPORT external clock must always be less than the SCLK frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

A SPORT offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.
- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.
- Provides two synchronous transmit and two synchronous receive data signals and buffers to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 19-29 for more information.)
- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.
- Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.
- Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.

- Provides direct memory access transfer to and from memory under DMA master control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Has a multichannel mode for TDM interfaces. A SPORT can receive and transmit data selectively from a time-division-multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 (= 1023 – 128). Note the multichannel select registers and the WSIZE register control which subset of the 128 channels within the active region can be accessed.

Interface Overview

A SPORT provides an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only. Each SPORT has one group of signals (primary data, secondary data, clock, and frame sync) for transmit and a second set of signals for receive. The receive and transmit functions are programmed separately. A SPORT is a full duplex device, capable of simultaneous data transfer in both directions. A SPORT can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

Figure 19-1 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX shift register. The bits in the shift register are shifted out on the DTPRI/DTSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLK pin. The receive portion of the SPORT accepts data from the DRPRI/DRSEC pin synchronous to the serial clock on the RSCLK pin. When an entire word

is received, the data is optionally expanded, then automatically transferred to the SPORT_RX register, and then into the RX FIFO where it is available to the processor. Table 19-1 shows the signals for each SPORT.

Table 19-1. SPORT Signals

Pin	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

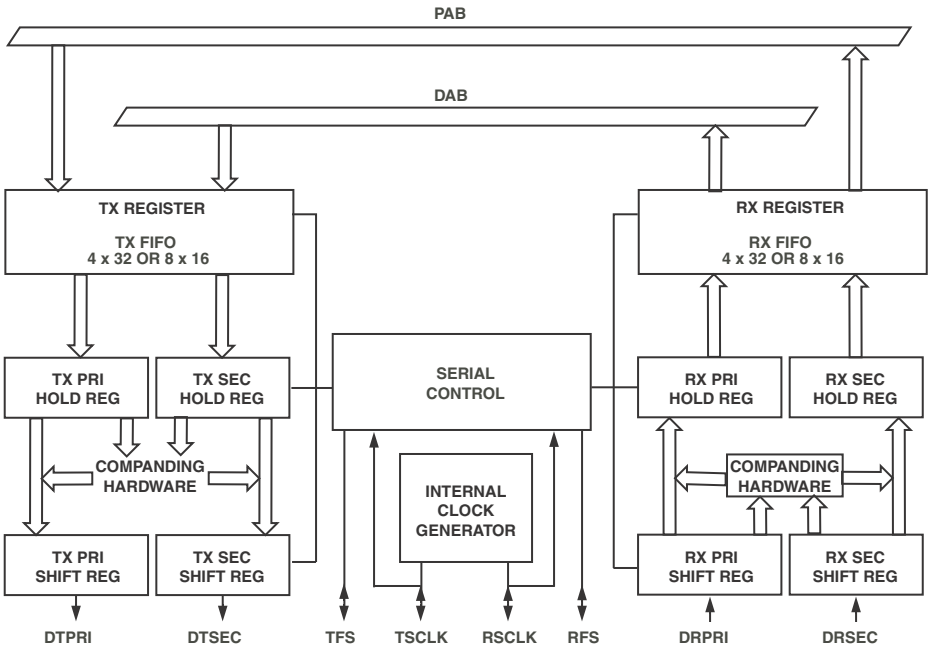


Figure 19-1. SPORT Block Diagram[1, 2, 3]

1 All wide arrow data paths are 16- or 32-bits wide, depending on SLEN. for SLEN = 2 to 15, a 16-bit data path with 8-deep fifo is used. for SLEN = 16 to 31, a 32-bit data path with 4-deep fifo is used.
2 TX register is the bottom of the TX fifo, RX register is the top of the RX fifo.
3 In multichannel mode, the TFS pin acts as transmit data valid (TDV). For more information, see "Multichannel Operation" on page 19-15.

A SPORT receives serial data on its DRPRI and DRSEC inputs and transmits serial data on its DTPRI and DTSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTPRI and DTSEC) are synchronous to the transmit clock (TSCLK). For receive, the data bits (DRPRI and DRSEC) are synchronous to the receive clock (RSCLK). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFS and TFS are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins, if enabled by a specific processor port configuration, provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary signals is interleaved in main memory and can be retrieved by setting a stride in the data address generators (DAG) unit. For more information about DAGs, see the *Data Address Generators* chapter in the *Blackfin Processor Programming Reference*. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor's powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

Figure 19-2 shows a possible port connection for a device with at least two SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices 1, 2, …N.

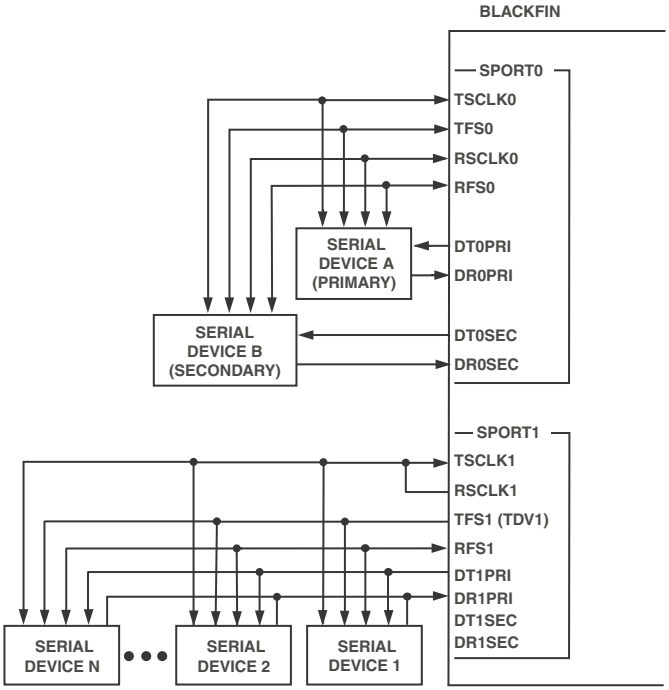


Figure 19-2. Example SPORT Connections
(SPORT0 is Standard Mode, SPORT1 is Multichannel Mode)[1, 2]

1 In multichannel mode, TFS functions as a transmit data valid (TDV) output. See "Multichannel Operation" on page 19-15.

2 Although shown as an external connection, the TSCLK1/RSCLK1 connection is internal in multichannel mode. See "Multichannel Operation" on page 19-15.

Figure 19-3 shows an example of a stereo serial device with three transmit and two receive channels connected to a processor with two SPORTs.

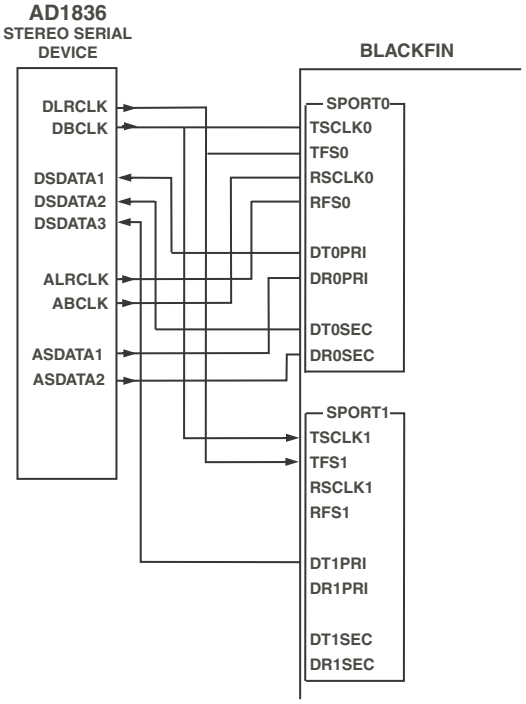


Figure 19-3. Stereo Serial Connection

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Description of Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT_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 SPORT_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORT_TCR1 register. Each bit is shifted out on the driving edge of TSCLK. The driving edge of TSCLK can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed.

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORT_TCR1 register and RSPEN in the SPORT_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORT_TCLKDIV, SPORT_RCLKDIV, SPORT_TFSDIVx, and SPORT_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFS/RFS and TSCLK/RSCLK outputs, if these signals are configured to be generated internally.

When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. A SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORT_TCR1 and/or RSPEN in SPORT_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in "SPORT Registers" on page 19-45. All control and status bits in the SPORT registers are active high unless otherwise noted.

Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or

SPORT_TCR2 changes the operation of the frame sync pin to a left/right clock as required for I2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 19-4 on page 19-14 and Figure 19-5 on page 19-15 show timing diagrams for stereo serial mode operation.

Table 19-2 shows several modes that can be configured using bits in SPORT_TCR1 and SPORT_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

ADSP-BF50x SPORTs are designed such that, in I^2S master mode, LRCLK is held at the last driven logic level and does not transition, to provide an edge, after the final data word is driven out. Therefore, while transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a ADSP-BF50x SPORT.

Table 19-2. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1
RCKFE	1	0	0

Table 19-2. Stereo Serial Settings (Cont'd)

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 19-2 provide glue-less interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I^2S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 19-26) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a "right" channel. Thus, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the "right" channel, thus implying that it is low for the "left" channel. This is the default setting.

The RRFST and TRFST bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRSEC and DTSEC pins are useful extensions of the SPORT which pair well with stereo serial mode. Multiple I^2S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 19-3, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

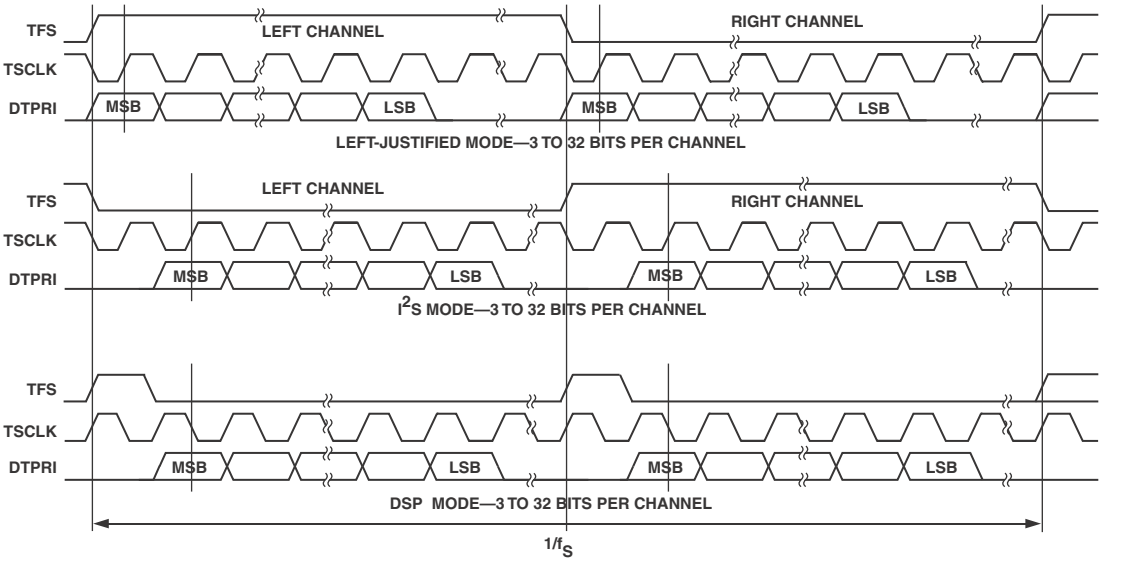


Figure 19-4. SPORT Stereo Serial Modes, Transmit[1, 2, 3]

1 DSP mode does not identify channel.
2 TFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 TSCLK frequency is normally 64 x TFS but may be operated in burst mode.

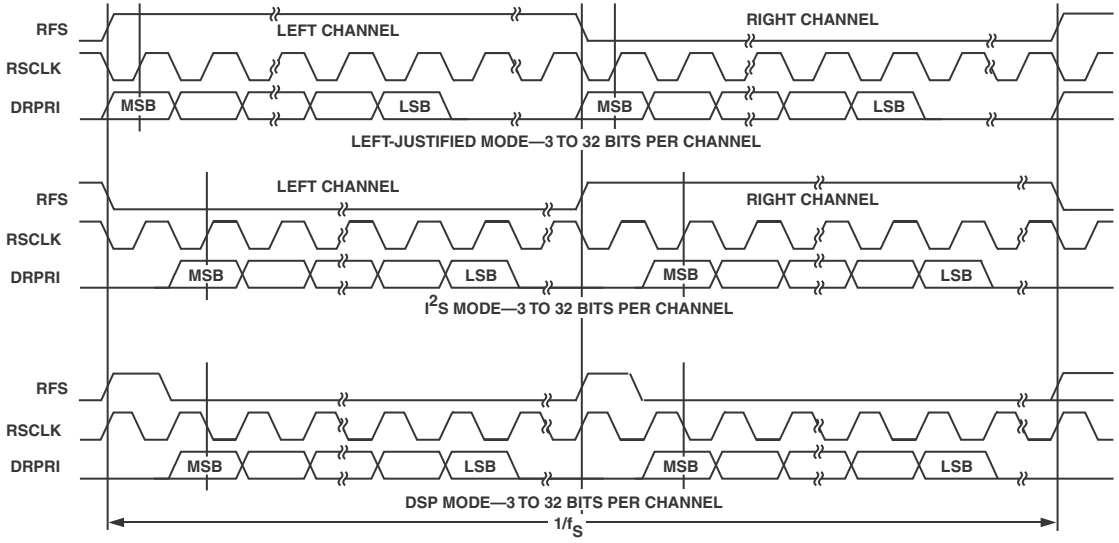


Figure 19-5. SPORT Stereo Serial Modes, Receive[1, 2, 3]

1 DSP mode does not identify channel.
2 RFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 RSCLK frequency is normally 64 × RFS but may be operated in burst mode.

Multichannel Operation

The SPORT offers a multichannel mode of operation which allows the SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bitstream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024

total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORT_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORT_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

In multichannel mode, RSCLK can either be provided externally or generated internally by the SPORT, and this signal is used for both transmit and receive functions. Leave TSCLK disconnected if the SPORT is used only in multichannel mode. If RSCLK is externally or internally provided, the signal is internally distributed to both the receiver and transmitter circuitry.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORT_TX or SPORT_RX operation for multichannel mode. This is especially important in “DMA data unpacked mode,” since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORT_MCMC2) must be enabled prior to enabling SPORT_TX or SPORT_RX operation. When disabling the SPORT from multichannel operation, first

disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 19-6 shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- RFS signals start of frame
- TFS is used as "transmit data valid" for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See "Timing Examples" on page 19-39 for more examples.

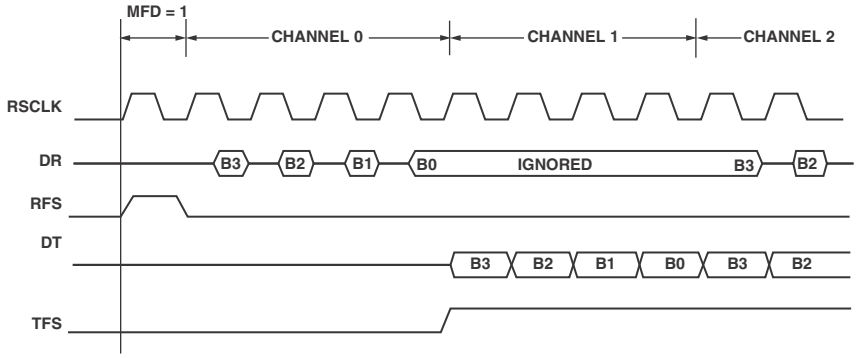


Figure 19-6. Multichannel Operation

Multichannel Enable

Setting the MCMEN bit in the SPORT_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 19-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 19-3. Multichannel Mode Configuration

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Independent
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0
RCKFE	TCKFE	Set or clear both to same value

Table 19-3. Multichannel Mode Configuration (Cont'd)

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFS indicates the beginning of the channel 0 data word.

Since RFS is used by both the SPORT_TX and SPORT_RX channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in SPORT_RCR1 and SPORT_TCR1, and in SPORT_RCR2 and SPORT_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORT_RX operation is not enabled.

In multichannel mode, RFS timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFS signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFS signal serves as an

output-enabled signal for the data transmit pin. The SPORT drives TFS in multichannel mode whether or not ITFS is cleared. The TFS pin in multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFS pin indicates an active channel.

Once the initial RFS is received, and a frame transfer has started, all other RFS signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFS may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFS signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers. See Figure 19-7.

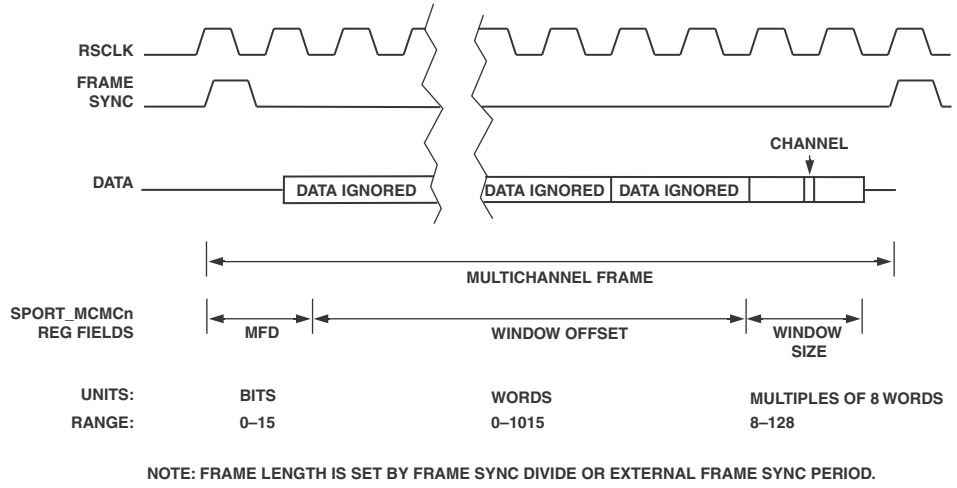


Figure 19-7. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit MFD field in SPORT_MCMC2 specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (WSIZE[3:0]) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active

window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

Number of words in active window = 8 × (WSIZE + 1)

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

Other Multichannel Fields in SPORT_MCMC2

The FSDR bit in the SPORT_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge on TFS causes data to be transmitted on the positive edge of the TSCLK—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a multibit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels. The SPORT_MRCSn and SPORT_MTCSn multichannel select registers are used to enable and disable individual channels; the SPORT_MRCSn registers specify the active receive channels, and the SPORT_MTCSn registers specify the active transmit channels.

Four registers make up each multichannel select register. Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit). See Figure 19-8.

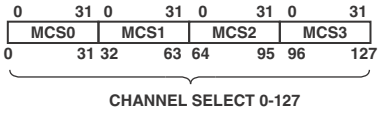


Figure 19-8. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the SPORT_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the SPORT_RX buffer. Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or µ-law companding is selected with the TDTYPE field in the SPORT_TCR1 register and the RDTYPE field in the SPORT_RCR1 register, and applies to all active channels. (See "Companding" on page 19-29 for more information about companding.)

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORT_MCMC2 multichannel configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguration is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words (unless the secondary side is enabled). The data to be

transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the channel register to make sure that the active window is not being serviced. If the channel count is 0, then the multichannel select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- TFSR/RFSR set (frame syncs required)
- LTFS/LRFS set (active low frame syncs)
- Set for external clock
- MCMEN set (multichannel mode selected)
- MFD = 0 (no frame delay between frame sync and first data bit)
- SLEN = 7 (8-bit words)
- FSDR = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2× Clock Recovery Control

The SPORT can recover the data rate clock from a provided 2× input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the

SPORT_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 00 chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

Functional Description

The following sections provide a functional description of the SPORT.

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORT_TCLKDIV and SPORT_RCLKDIV.

TSCLK frequency = (SCLK frequency)/(2 × (SPORT_TCLKDIV + 1))

RSCLK frequency = (SCLK frequency)/(2 × (SPORT_RCLKDIV + 1))

If the value of SPORT_TCLKDIV or SPORT_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLK or RSCLK that follows the next leading edge of TFS or RFS.

When an internal frame sync is selected (ITFS = 1 in the SPORT_TCR1 register or IRFS = 1 in the SPORT_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORT_TCLKDIV or SPORT_RCLKDIV has changed. The second frame sync will cause the update.

The SPORT_TFSDIV and SPORT_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or

RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

# of transmit serial clocks between frame sync assertions = TFSDIV + 1

# of receive serial clocks between frame sync assertions = RFSDIV + 1

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

SPORT TFS frequency = (TSCLK frequency)/(SPORT_TFSDIV + 1)

SPORT RFS frequency = (RSCLK frequency)/(SPORT_RFSDIV + 1)

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORT_TCR2 or SPORT_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for exact timing specifications.

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORT_TCR2 and SPORT_RCR2 registers of each SPORT determines the word length according to this formula:

Serial Word Length = SLEN + 1

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORT_RCR1 and SPORT_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORT_TCR1 register and the RDTYPE field of the SPORT_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 19-4.

Table 19-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORT_TCR1 Data Formatting	SPORT_RCR1 Data Formatting
00	Normal operation	Zero fill
01	Reserved	Sign extend
10	Compand using μ-law	Compand using μ-law
11	Compand using A-law	Compand using A-law

These formats are applied to serial data words loaded into the SPORT_RX and SPORT_TX buffers. SPORT_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 SPORT supports the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORT_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORT_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLK) and a receive clock signal (RSCLK). The clock signals are configured by the TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers. Serial clock frequency is configured in the SPORT_TCLKDIV and SPORT_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORT_TCR1 configuration register and the IRCLK bit in the SPORT_RCR1 configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the processor, and the TSCLK or RSCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORT_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLK or RSCLK pins, and the serial clock divisors in the SPORT_TCLKDIV/SPORT_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the system clock or with each other. The system clock must have a higher frequency than RSCLK and TSCLK.

When the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for details.

The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame sync) and RFS (receive frame sync). A variety of framing options are available; these options are configured in the SPORT configuration registers (SPORT_TCR1, SPORT_TCR2, SPORT_RCR1 and SPORT_RCR2). The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (transmit frame sync required select) and RFSR (receive frame sync required select) control bits determine whether frame sync signals are required. These bits are located in the SPORT_TCR1 and SPORT_RCR1 registers.

When `TFSR` = 1 or `RFSR` = 1, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the `SPORT_TX` hold register before the previous word is shifted out and transmitted.

When `TFSR` = 0 or `RFSR` = 0, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.

Figure 19-9 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORT_TCR1 and SPORT_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers.

See "Timing Examples" on page 19-39 for more timing examples.

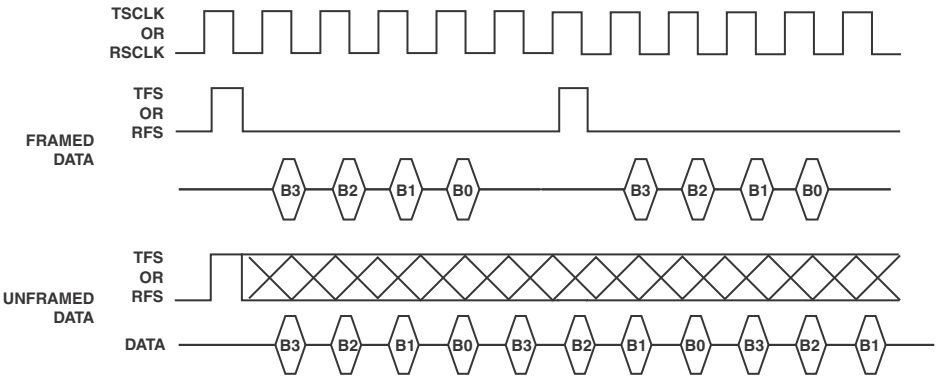


Figure 19-9. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine the frame sync source.

When ITFS = 1 or IRFS = 1, the corresponding frame sync signal is generated internally by the SPORT, and the TFS pin or RFS pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the SPORT_TFSDIV or SPORT_RFSDIV register.

When ITFS = 0 or IRFS = 0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pin, and the frame sync divisors in the SPORT_TFSDIV/SPORT_RFSDIV registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine frame sync logic levels:

- When LTFS = 0 or LRFS = 0, the corresponding frame sync signal is active high.
- When LTFS = 1 or LRFS = 1, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LTFS and LRFS bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORT_TCR1 register selects the falling edge of TSCLK to drive data and internally generated frame syncs and selects the rising edge of TSCLK to sample externally

generated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLK to drive data and internally generated frame syncs and selects the falling edge of TSCLK to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORT_RCR1 register selects the falling edge of RSCLK to drive internally generated frame syncs and selects the rising edge of RSCLK to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLK to drive internally generated frame syncs and selects the falling edge of RSCLK to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORT_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 19-10, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

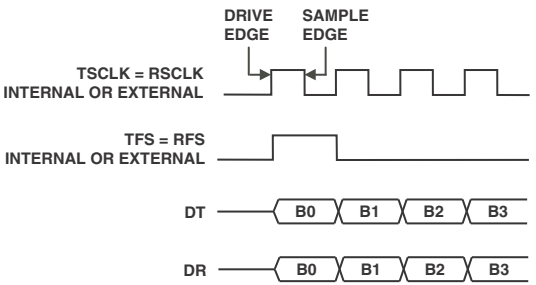


Figure 19-10. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 19-11, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

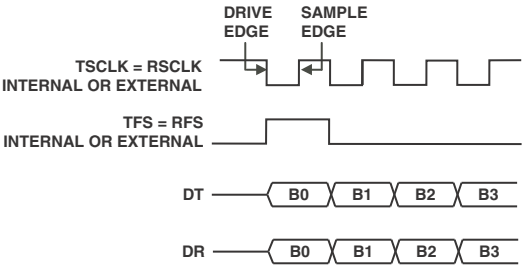


Figure 19-11. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORT_TCR1 and SPORT_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early

framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN ≥ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 19-12 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORT_TCR1 or SPORT_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See “Timing Examples” on page 19-39 for more examples.

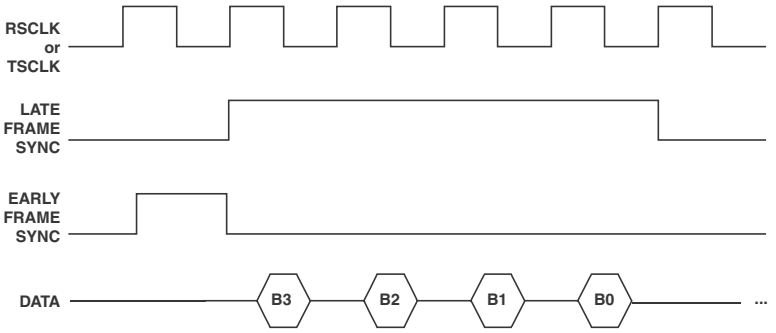


Figure 19-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS) is output only when the SPORT_TX buffer has data ready to transmit. The data-independent transmit frame sync select bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORT_TCR1 register configures this option.

When DITFS = 0, the internally generated TFS is only output when a new data word has been loaded into the SPORT_TX buffer. The next TFS is generated once data is loaded into SPORT_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the SPORT_TX buffer. Whatever data is present in SPORT_TX is transmitted again with each assertion of TFS. The TUVF (transmit underflow status) bit in the SPORT_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORT_TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPORT_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

Peripheral Bus Errors

The SPORT generates a peripheral bus error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_TX)
- Writing a read-only register (for example, SPORT_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 19-31, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 19-35, and "Frame Syncs in Multichannel Mode" on page 19-19). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing signals are active high (LRFS = 0 and LTFS = 0).

Figure 19-13 on page 19-40 through Figure 19-18 on page 19-42 show framing for receiving data.

In Figure 19-13 and Figure 19-14, the normal framing mode is shown for non-continuous data (any number of TSCLK or RSCLK cycles between words) and continuous data (no TSCLK or SCLK cycles between words).

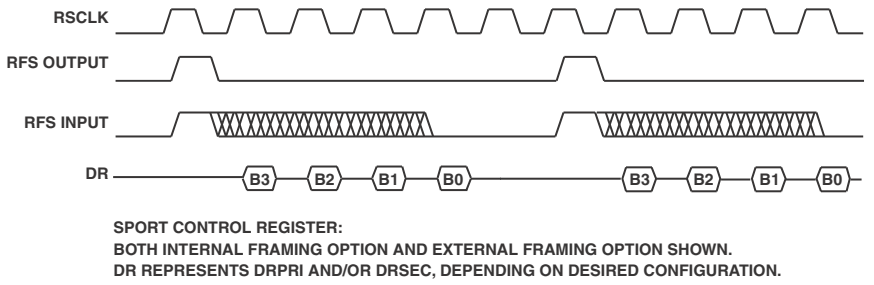


Figure 19-13. SPORT Receive, Normal Framing

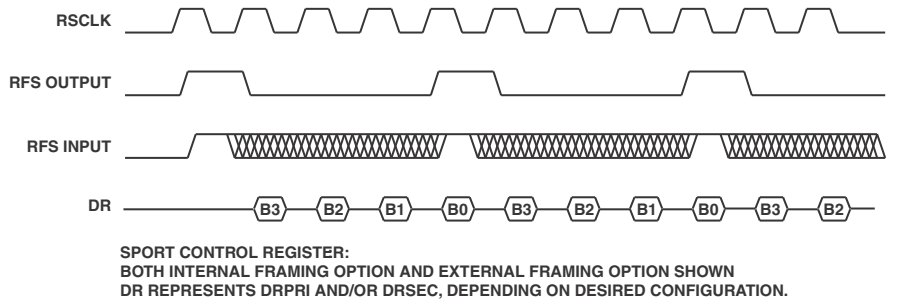


Figure 19-14. SPORT Continuous Receive, Normal Framing

Figure 19-15 and Figure 19-16 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note the output meets the input timing requirement; therefore, with two SPORT channels used, one SPORT channel could provide RFS for the other SPORT channel.

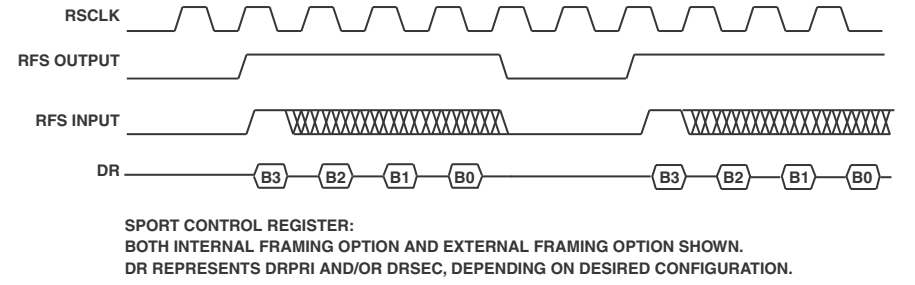


Figure 19-15. SPORT Receive, Alternate Framing

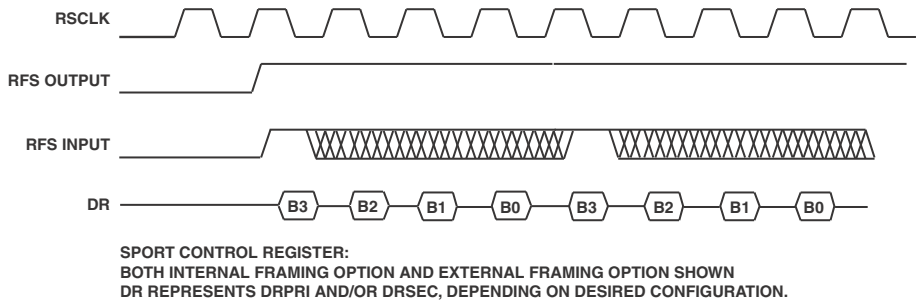


Figure 19-16. SPORT Continuous Receive, Alternate Framing

Figure 19-17 and Figure 19-18 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one RSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

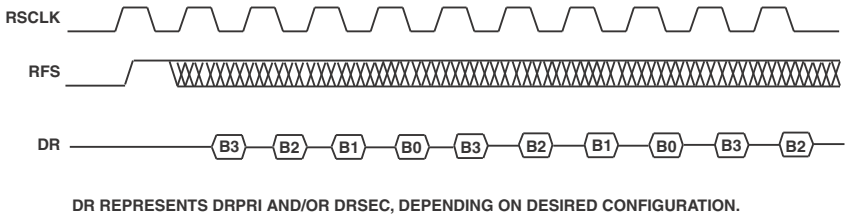


Figure 19-17. SPORT Receive, Unframed Mode, Normal Framing

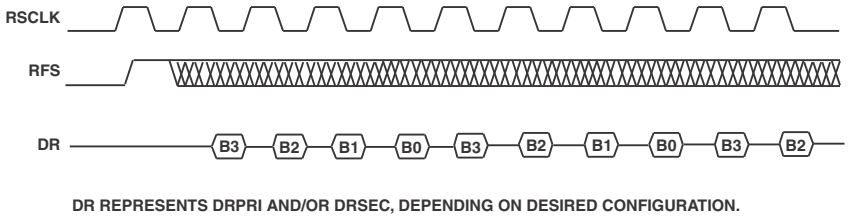


Figure 19-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 19-19 through Figure 19-24 show framing for transmitting data and are very similar to Figure 19-13 through Figure 19-18.

In Figure 19-19 and Figure 19-20, the normal framing mode is shown for non-continuous data (any number of TSCLK cycles between words) and continuous data (no TSCLK cycles between words). Figure 19-21 and Figure 19-22 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFS output meets the RFS input timing requirement.

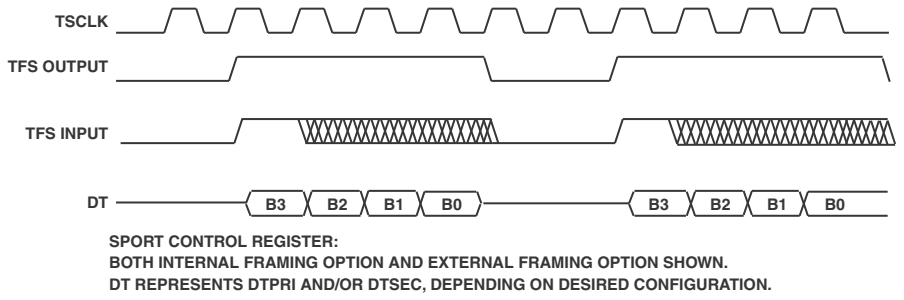


Figure 19-19. SPORT Transmit, Normal Framing

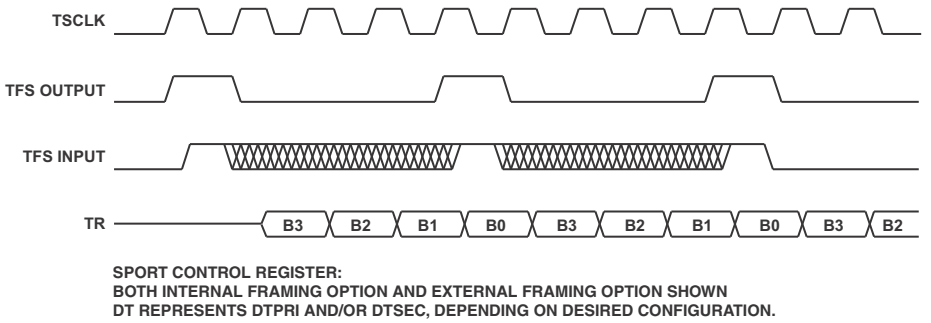


Figure 19-20. SPORT Continuous Transmit, Normal Framing

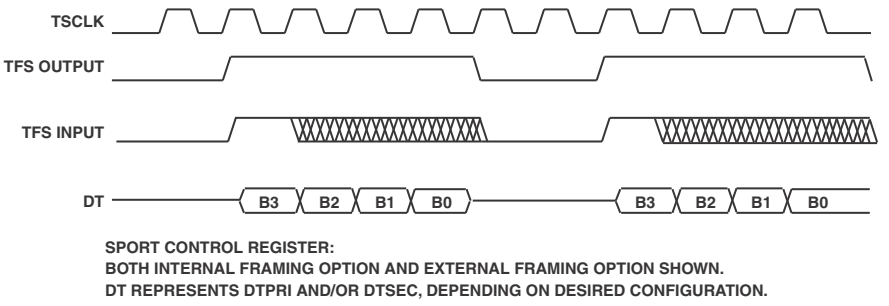


Figure 19-21. SPORT Transmit, Alternate Framing

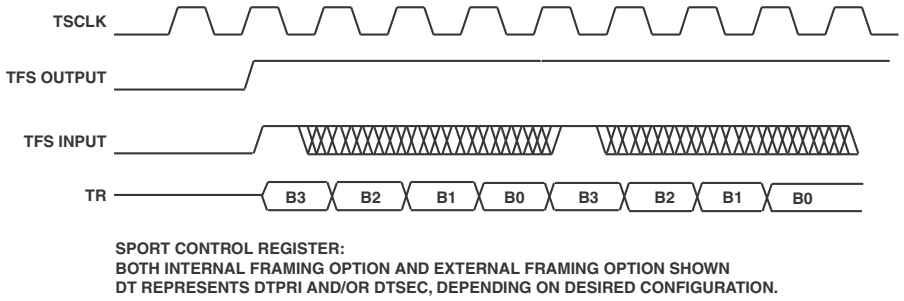


Figure 19-22. SPORT Continuous Transmit, Alternate Framing

Figure 19-23 and Figure 19-24 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

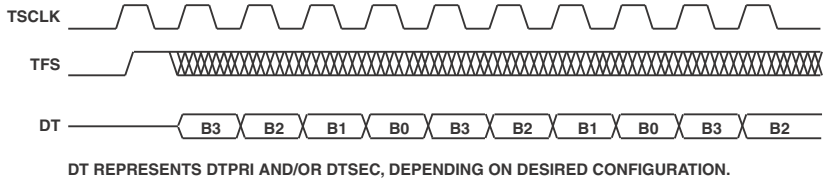


Figure 19-23. SPORT Transmit, Unframed Mode, Normal Framing

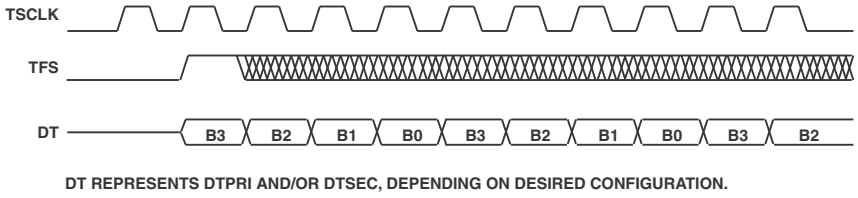


Figure 19-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. Table 19-5 provides an overview of the available control registers.

Table 19-5. SPORT Register Mapping

Register Name	Function	Notes
SPORT_TCR1	Primary transmit configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_TCR2	Secondary transmit configuration register	
SPORT_TCLKDIV	Transmit clock divider register	Ignored if external SPORT clock mode is selected
SPORT_TFSDIV	Transmit frame sync divider register	Ignored if external frame sync mode is selected
SPORT_TX	Transmit data register	See description of FIFO buffering at "SPORT Transmit Data (SPORT_TX) Register" on page 19-57
SPORT_RCR1	Primary receive configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_RCR2	Secondary receive configuration register	
SPORT_RCLK_DIV	Receive clock divider register	Ignored if external SPORT clock mode is selected
SPORT_RFSDIV	Receive frame sync divider register	Ignored if external frame sync mode is selected
SPORT_RX	Receive data register	See description of FIFO buffering at "SPORT Receive Data (SPORT_RX) Register" on page 19-59
SPORT_STAT	Receive and transmit status	
SPORT_MCM1	Primary multichannel mode configuration register	Configure this register before enabling the SPORT

Table 19-5. SPORT Register Mapping (Cont'd)

Register Name	Function	Notes
SPORT_MCM2	Secondary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORT_MRCSn	Receive channel selection registers	Select or deselect channels in a multichannel frame
SPORT_MTCSn	Transmit channel selection registers	Select or deselect channels in a multichannel frame
SPORT_CHNL	Currently serviced channel in a multichannel frame	

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). The SPORT_TX register writes are always enabled; SPORT_RX, SPORT_CHNL, and SPORT_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORT_TCR1 and SPORT_TCR2, shown in Figure 19-25 and Figure 19-26.

A SPORT is enabled for transmit if bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_TCR1 is not written except for bit 0 (TSPEN). For example,

```
write (SPORT_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORT_TCR1, 0xFF01) ;   /* ignored, no effect */
write (SPORT_TCR1, 0xFFF0) ;   /* SPORT disabled, SPORT_TCR1
                                  still equal to 0x0000 */
```

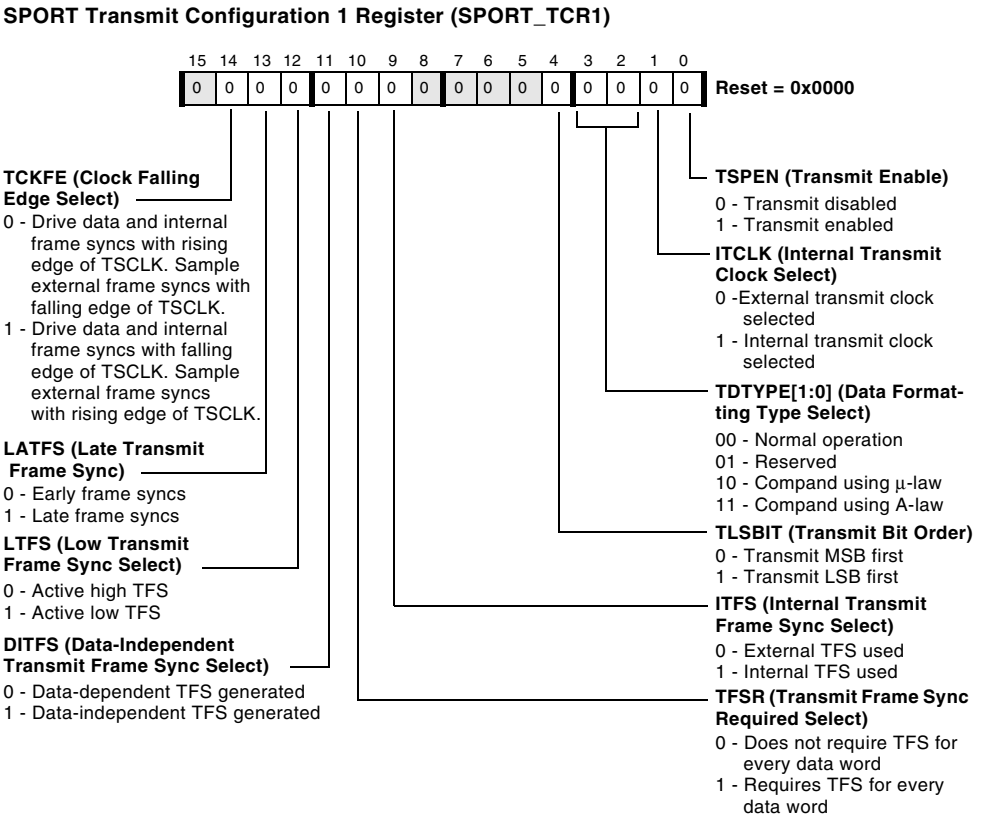


Figure 19-25. SPORT Transmit Configuration 1 Register

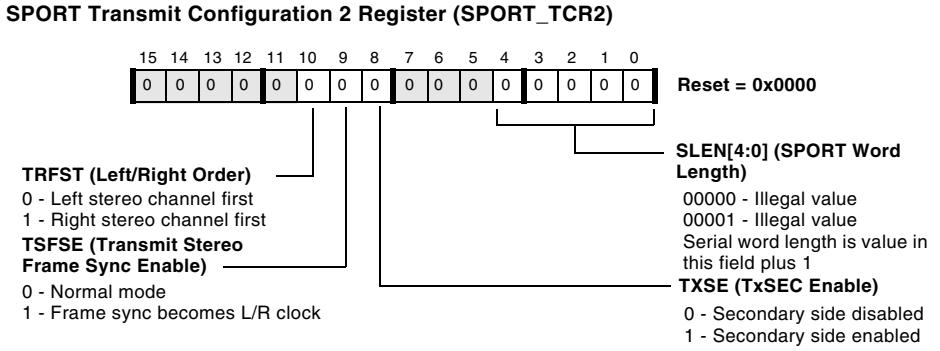


Figure 19-26. SPORT Transmit Configuration 2 Register

Additional information for the SPORT_TCR1 and SPORT_TCR2 transmit configuration register bits includes:

- **Transmit enable** (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting TSPEN causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting TSPEN.

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting TSPEN. Set all DMA control registers before setting TSPEN.

  Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_TCR1 with all of the necessary bits, including TSPEN.

- **Internal transmit clock select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit order select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

  Serial Word Length = SLEN + 1;

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The SPORT always transfers the SLEN+1 lower bits from the transmit buffer.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider

must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal transmit frame sync select**. (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit frame sync required select**. (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transmit frame sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent transmit frame sync select**. (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORT_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORT_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORT_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the SPORT_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORT_TX register.
- **Low transmit frame sync select**. (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).

- **Late transmit frame sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec enable**. (TXSE). This bit enables the transmit secondary side of the SPORT (if set).
- **Stereo serial enable**. (TSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (TRFST). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORT_RCR1 and SPORT_RCR2, shown in Figure 19-27 and Figure 19-28.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

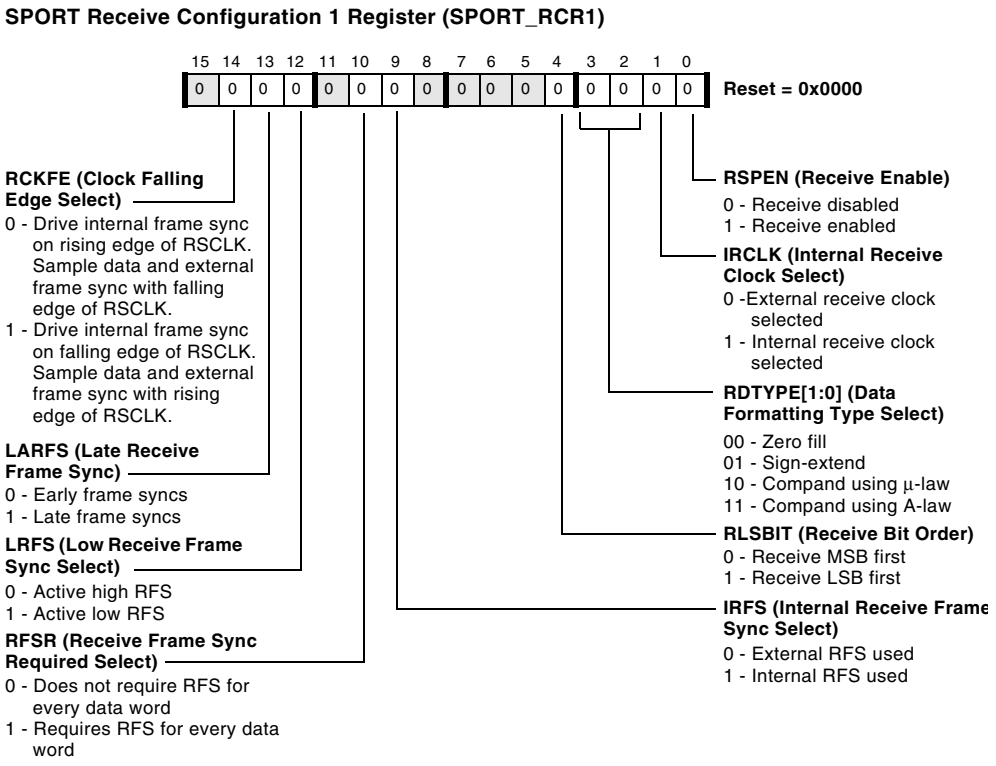


Figure 19-27. SPORT Receive Configuration 1 Register

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_RCR1 is not written except for bit 0 (RSPEN). For example,

```
write (SPORT_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORT_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORT_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORT_RCR1
                                 still equal to 0x0000 */
```

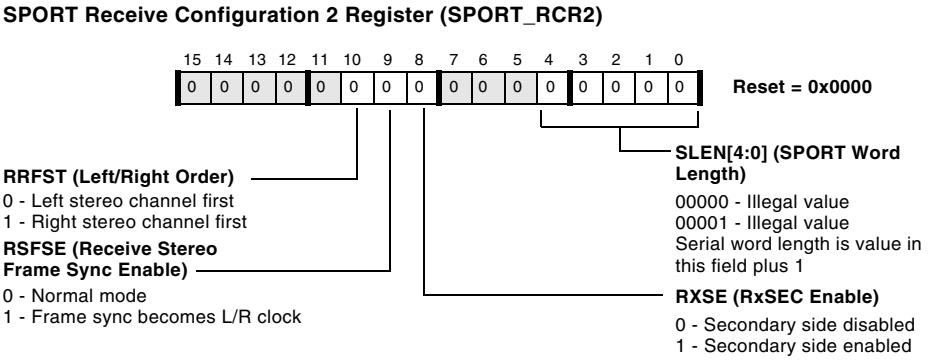


Figure 19-28. SPORT Receive Configuration 2 Register

Additional information for the SPORT_RCR1 and SPORTRCR2 receive configuration register bits:

- **Receive enable**. (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

  Setting RSPEN enables the SPORT receiver, which can generate a SPORT RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN.

  Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_RCR1 with all of the necessary bits, including RSPEN.

- **Internal receive clock select**. (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
- Bit order select. (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal receive frame sync select**. (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- **Receive frame sync required select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a receive frame sync for every data word.

- **Low receive frame sync select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late receive frame sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec enable**. (RXSE). This bit enables the receive secondary side of the SPORT (if set).
- **Stereo serial enable**. (RSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (RRFST). If this bit is set, the right channel is received first in stereo serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers.

SPORT Transmit Data (SPORT_TX) Register

The SPORT_TX register is a write-only register. Reads produce a peripheral bus error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length ≤ 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 19-29. The SPORT_TX register is shown in Figure 19-30.

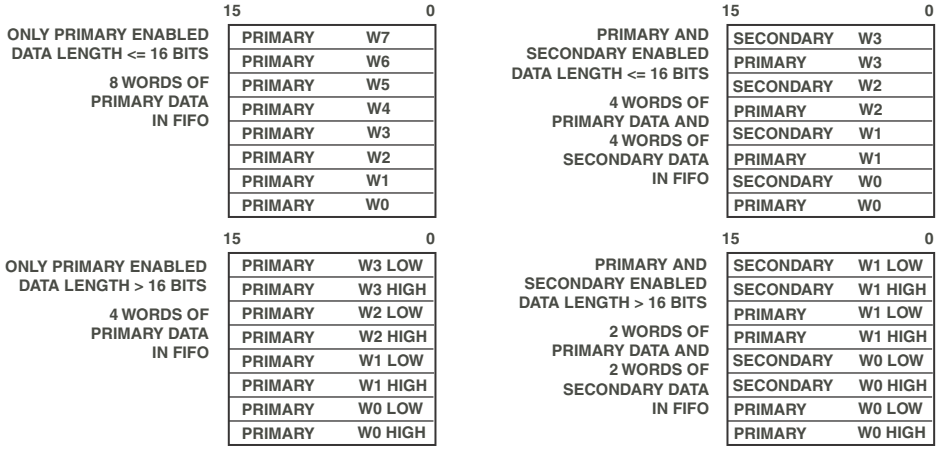


Figure 19-29. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that peripheral bus/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/peripheral bus writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on TXSE and SLEN, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the DTPRI and DTSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled.

The transmit underflow status bit (TUVF) is set in the SPORT_STAT register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORT_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORT_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT_STAT register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORT_STAT register are updated upon writes from the core processor, even when the SPORT is disabled.

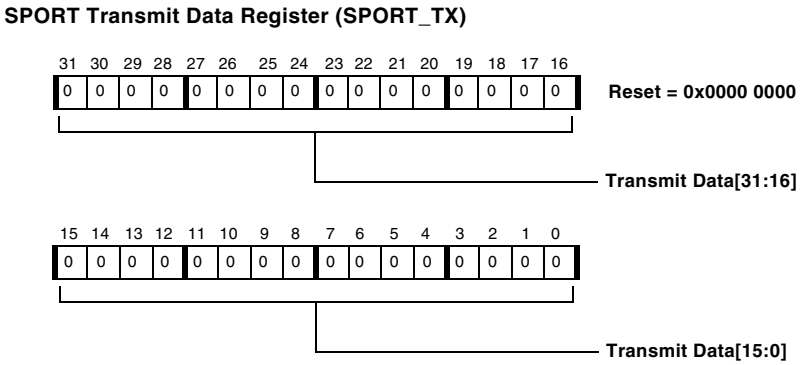


Figure 19-30. SPORT Transmit Data Register

SPORT Receive Data (SPORT_RX) Register

The SPORT_RX register is a read-only register. Writes produce a peripheral bus error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length ≤ 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using peripheral bus/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO are shown in Figure 19-31. The SPORT_RX register is shown in Figure 19-32.

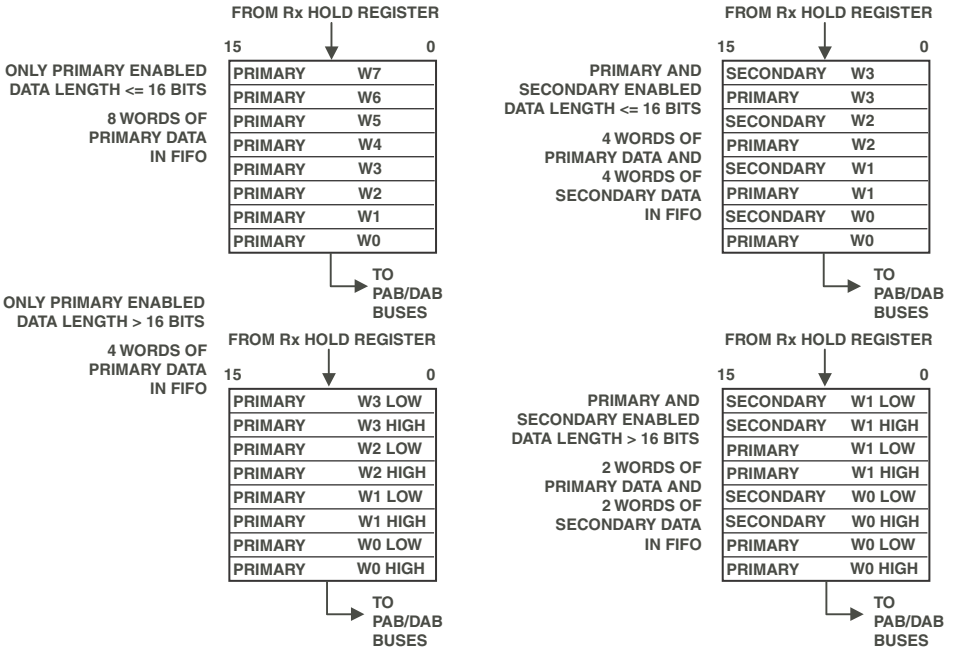


Figure 19-31. SPORT Receive FIFO Data Ordering

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/peripheral bus reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRPRI pin is loaded into the RX primary shift register, while data from the DRSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORT_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORT_STAT register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORT_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT RX.

SPORT Receive Data Register (SPORT_RX)

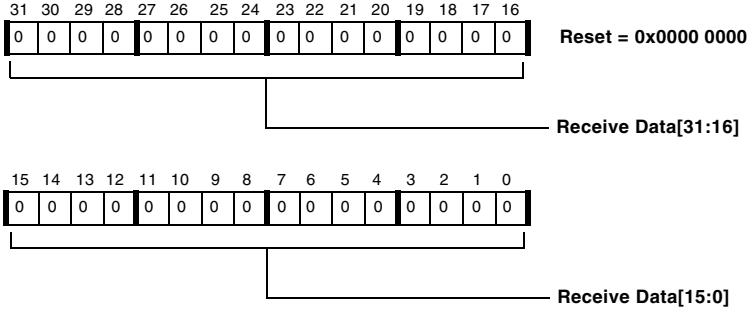


Figure 19-32. SPORT Receive Data Register

SPORT Status (SPORT_STAT) Register

The SPORT_STAT register is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. This register is shown in Figure 19-33.

The TXF bit in the SPORT_STAT register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.

The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever SPORT_TX is empty by clearing the DITFS control bit in the SPORT_TCR1 register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TXEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT_STAT register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing RXEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RXEN = 0. The RUVF bit is updated even when the SPORT is disabled.

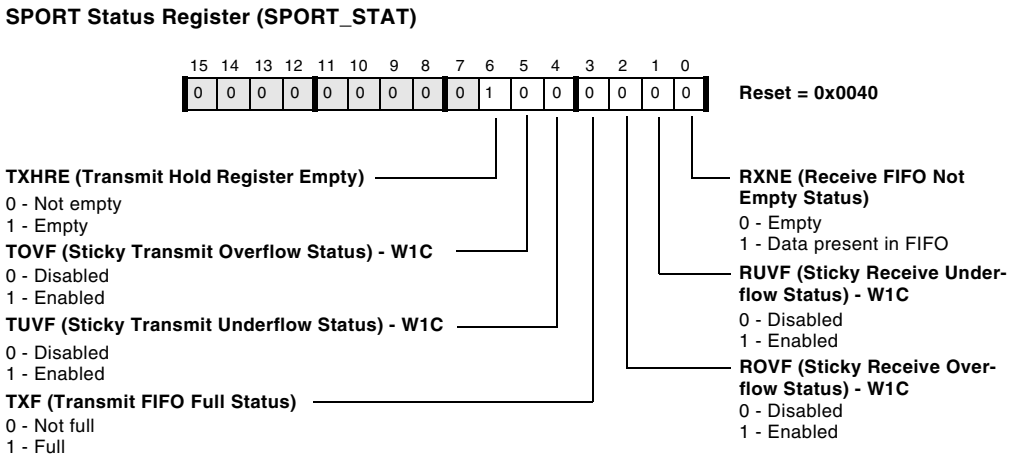


Figure 19-33. SPORT Status Register

SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORT_TCLKDIV register, shown in Figure 19-34, and the SPORT_RCLKDIV register, shown in Figure 19-35).

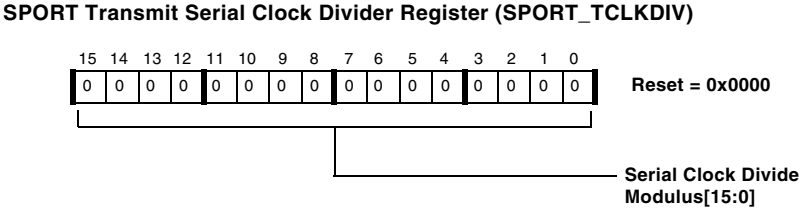


Figure 19-34. SPORT Transmit Serial Clock Divider Register

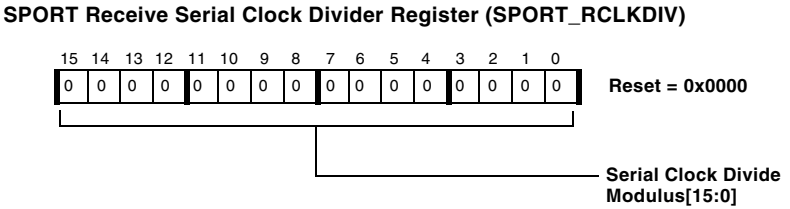


Figure 19-35. SPORT Receive Serial Clock Divider Register

SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers

The 16-bit SPORT_TFSDIV and SPORT_RFSDIV registers specify how many transmit or receive clock cycles are counted before generating a TFS or RFS pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks. These registers are shown in Figure 19-36 and Figure 19-37.

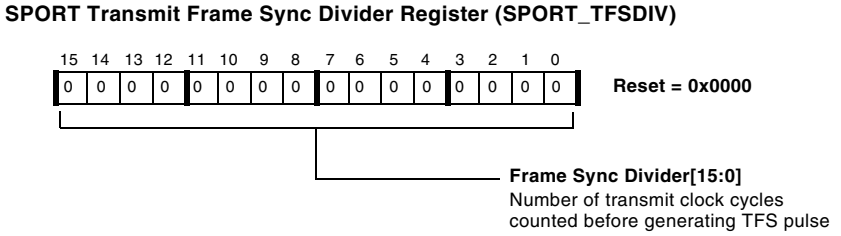


Figure 19-36. SPORT Transmit Frame Sync Divider Register

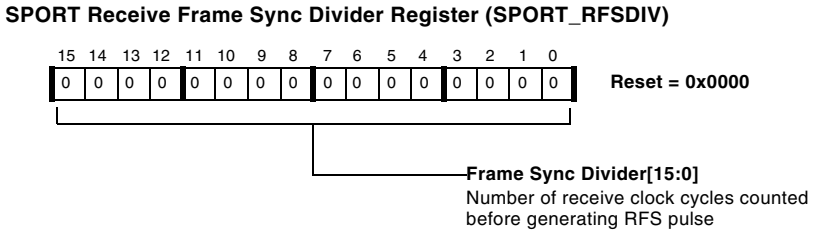


Figure 19-37. SPORT Receive Frame Sync Divider Register

SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers

There are two multichannel configuration registers for each SPORT, shown in Figure 19-38 and Figure 19-39. These registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

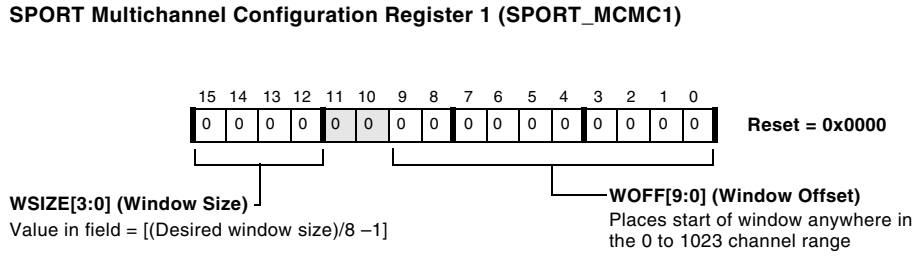


Figure 19-38. SPORT Multichannel Configuration Register 1

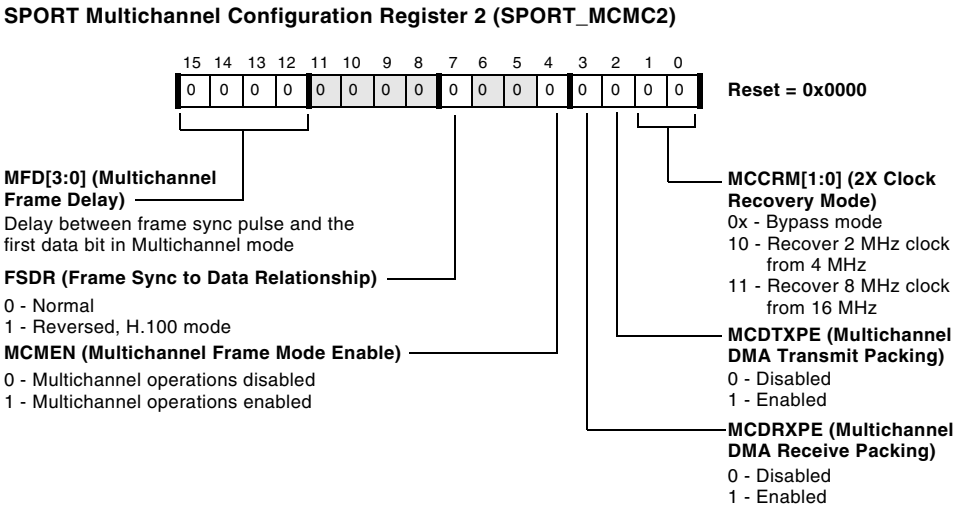


Figure 19-39. SPORT Multichannel Configuration Register 2

SPORT Current Channel (SPORT_CHNL) Register

The 10-bit `CHNL` field in the `SPORT_CHNL` register indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The `CHNL[9:0]` field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between `RSCLK` and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind. See Figure 19-40.

SPORT Current Channel Register (SPORT_CHNL)
RO

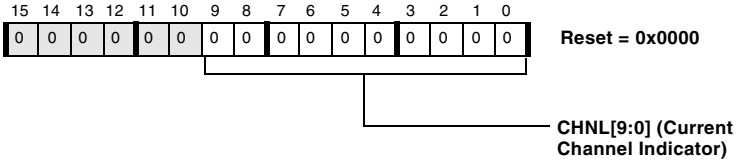


Figure 19-40. SPORT Current Channel Register

SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers

The SPORT_MRCSn registers (shown in Figure 19-41) are used to enable and disable individual channels. They specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. When the secondary receive side is enabled by the RXSE bit, both inputs are processed on enabled channels. Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data on either channel.

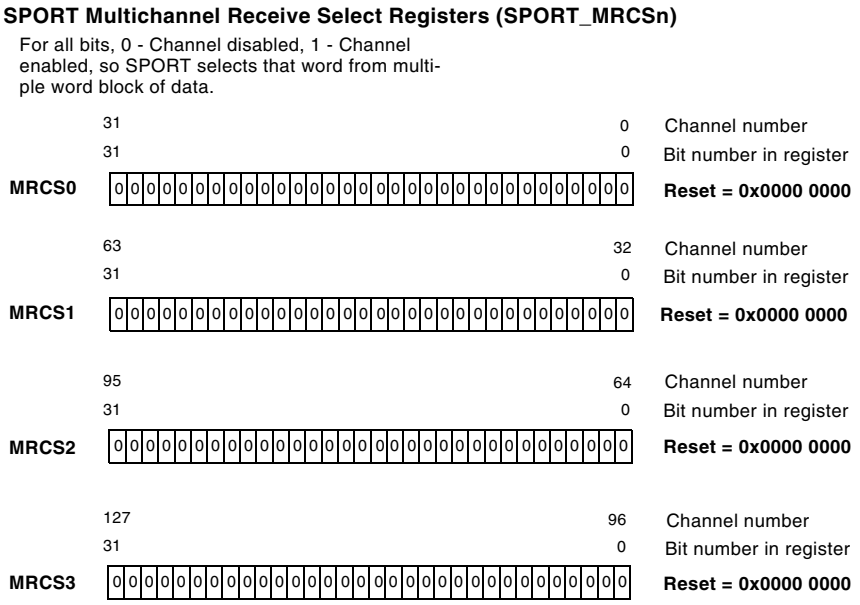


Figure 19-41. SPORT Multichannel Receive Select Registers

SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers

The SPORT_MTCSn registers (shown in Figure 19-42) are used to enable and disable individual channels. They specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. When the secondary transmit side is enabled by the TXSE bit, both sides transmit a word on the enabled channel. Clearing the bit in the SPORT_MTCSn register

causes a SPORT controllers' data transmit pins to three-state during the time slot of that channel.

SPORT Multichannel Transmit Select Registers (SPORT_MTCSn)
For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

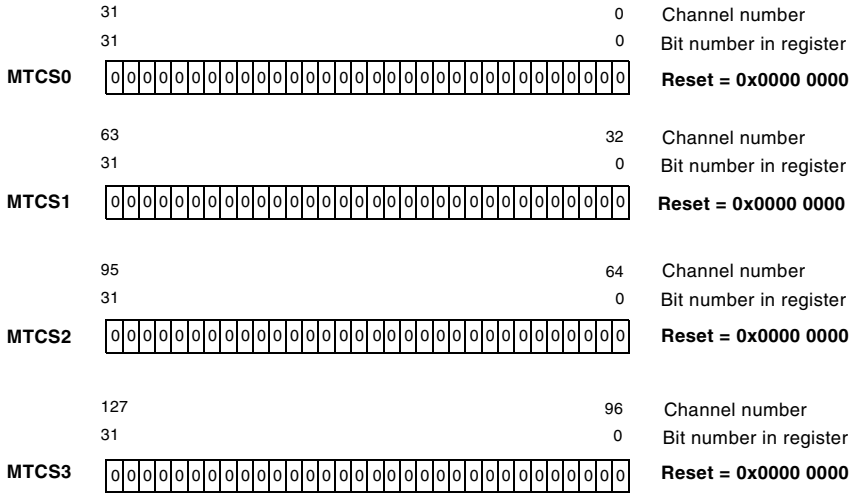


Figure 19-42. SPORT Multichannel Transmit Select Registers

Programming Examples

This section shows an example of typical usage of the SPORT peripheral in conjunction with the DMA controller. See Listing 19-1 through Listing 19-4. These listings assume a processor with at least two SPORTs, SPORT0 and SPORT1.

The SPORT is usually employed for high-speed, continuous serial transfers. The example reflects this, in that the SPORT is set-up for auto-buffered, repeated DMA transfers.

Because of the many possible configurations, the example uses generic labels for the content of the SPORT's configuration registers (`SPORT_RCRn`

and SPORT_TCRn) and the DMA configuration. An example value is given in the comments, but for the meaning of the individual bits the user is referred to the detailed explanation in this chapter.

The example configures both the receive and the transmit section. Since they are completely independent, the code uses separate labels.

SPORT Initialization Sequence

The SPORT's receiver and transmitter are configured, but they are not enabled yet.

Listing 19-1. SPORT Initialization

```
Program_SPORT_TRANSMITTER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);

/* Configure Clock speeds */
  R1 = SPORT_TCLK_CONFIG;  /* Divider SCLK/TCLK (value 0 to
65535) */
  W[P0 + (SPORT0_TCLKDIV - SPORT0_TCR1)] = R1;
                        /* TCK divider register */
  /* number of Bitclocks between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_TFSDIV_CONFIG;
  W[P0 + (SPORT0_TFSDIV - SPORT0_TCR1)] = R1;
                       /* TFSDIV register */

/* Transmit configuration */
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_TRANSMIT_CONF_2;
```

```
  W[P0 + (SPORT0_TCR2 - SPORT0_TCR1)] = R1;
  /* Configuration register 1   (for instance 0x4E12 for inter-
nally generated clk and framesync) */
  R1 = SPORT_TRANSMIT_CONF_1;
  W[P0] = R1;
  ssync;
/* NOTE: SPORT0 TX NOT enabled yet (bit 0 of TCR1 must be zero) */

Program_SPORT_RECEIVER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);

/* Configure Clock speeds */
  R1 = SPORT_RCLK_CONFIG;  /* Divider SCLK/RCLK (value 0 to
65535) */
  W[P0 + (SPORT0_RCLKDIV - SPORT0_RCR1)] = R1;   /* RCK divider
register */
  /* number of Bitclock between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_RFSDIV_CONFIG;
  W[P0 + (SPORT0_RFSDIV - SPORT0_RCR1)] = R1;
                              /* RFSDIV register */

/* Receive configuration */
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_RECEIVE_CONF_2;
  W[P0 + (SPORT0_RCR2 - SPORT0_RCR1)] = R1;
  /* Configuration register 1   (for instance 0x4410 for external
clk and framesync) */
  R1 = SPORT_RECEIVE_CONF_1;
```

```
  W[P0] = R1;
  ssync;  /* NOTE: SPORT0 RX NOT enabled yet (bit 0 of RCR1 must
be zero) */
```

DMA Initialization Sequence

Next the DMA channels for receive (channel3 in this example) and for transmit (channel4 in this example) are set up for auto-buffered, one-dimensional, 32-bit transfers. Again, there are other possibilities, so generic labels have been used, with a particular value shown in the comments.

Note that the DMA channels can be enabled at the end of the configuration since the SPORT is not enabled yet. However, if preferred, the user can enable the DMA later, immediately before enabling the SPORT. The only requirement is that the DMA channel be enabled before the associated peripheral is enabled to start the transfer.

Listing 19-2. DMA Initialization

```
Program_DMA_Controller:

/* Receiver (DMA channel 3) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA3_CONFIG);
  P0.h = hi(DMA3_CONFIG);

  /* Configuration (for instance 0x108A for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_RECEIVE_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* rx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
```

```
  R1 = (length(rx_buf)/4)(z);
  W[P0 + (DMA3_X_COUNT - DMA3_CONFIG)] = R1;
                                        /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA3_X_MODIFY - DMA3_CONFIG)] = R1;
                                       /* X_modify register */

           /* start_address register points to memory buffer
               to be filled */
  R1.l = rx_buf;
  R1.h = rx_buf;
  [P0 + (DMA3_START_ADDR - DMA3_CONFIG)] = R1;

  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
set bit 0 */
  W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */

/* Transmitter (DMA channel 4) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA4_CONFIG);
  P0.h = hi(DMA4_CONFIG);
  /* Configuration (for instance 0x1088 for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_TRANSMIT_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* tx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(tx_buf)/4)(z);
  W[P0 + (DMA4_X_COUNT - DMA4_CONFIG)] = R1;
             /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
```

```
W[P0 + (DMA4_X_MODIFY - DMA4_CONFIG)] = R1;
            /* X_modify register */

/* start_address register points to memory buffer to be
   transmitted from */
R1.l = tx_buf;
R1.h = tx_buf;
[P0 + (DMA4_START_ADDR - DMA4_CONFIG)] = R1;

BITSET(R0,0);  /* R0 still contains value of CONFIG register -
                  set bit 0 */
W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */
```

Interrupt Servicing

The receive channel and the transmit channel will each generate an interrupt request if so programmed. The following code fragments show the minimum actions that must be taken. Not shown is the programming of the core and system event controllers.

Listing 19-3. Servicing an Interrupt

```
RECEIVE_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA3_IRQ_STATUS);
  P0.l = lo(DMA3_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */

  RETI = [SP++];
  rti;
```

```
TRANSMIT_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA4_IRQ_STATUS);
  P0.l = lo(DMA4_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */

  RETI = [SP++];
  rti;
```

Starting a Transfer

After the initialization procedure outlined in the previous sections, the receiver and transmitter are enabled. The core may just wait for interrupts.

Listing 19-4. Starting a Transfer

```
/* Enable Sport0 RX and TX */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Receiver (set bit 0) */

  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Transmitter (set bit 0) */
```

```
/* dummy wait loop (do nothing but waiting for interrupts) */
wait_forever:
 jump wait_forever;
```

Unique Information for the ADSP-BF50x Processor

None.

20 PARALLEL PERIPHERAL INTERFACE

This chapter describes the parallel peripheral interface (PPI). Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF50x

For details regarding the number of PPIs for the ADSP-BF50x product, please refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

For PPI DMA channel assignments, refer to Table 7-7 on page 7-107 in Chapter 7, "Direct Memory Access".

For PPI interrupt vector assignments, refer to Table 4-3 on page 4-19 in Chapter 4, "System Interrupts".

To determine how each of the PPIs is multiplexed with other functional pins, refer to Table 9-1 on page 9-4 through Table 9-3 on page 9-6 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each PPI, refer to Chapter A, "System MMR Assignments".

PPI behavior for the ADSP-BF50x that differs from the general information in this chapter can be found in the section "Unique Information for the ADSP-BF50x Processor" on page 20-37.

Overview

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

Features

The PPI includes these features:

- Half duplex, bidirectional parallel port
- Supports up to 16 bits of data
- Programmable clock and frame sync polarities
- ITU-R 656 support
- Interrupt generation on overflow and underrun

Typical peripheral devices that can be interfaced to the PPI port:

- A/D converters
- D/A converters
- LCD panels
- CMOS sensors
- Video encoders
- Video decoders

Interface Overview

Figure 20-1 shows a block diagram of the PPI.

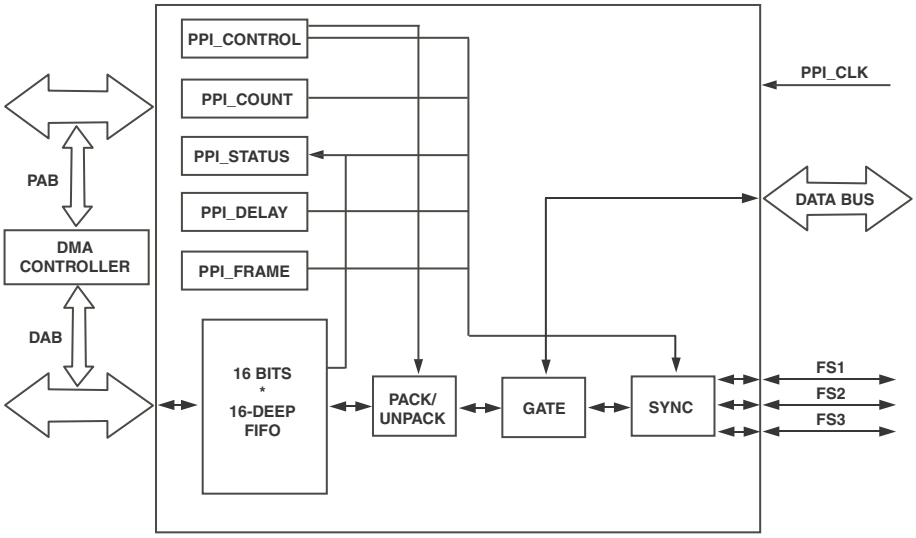


Figure 20-1. PPI Block Diagram

The PPI_CLK pin accepts an external clock input. It cannot source a clock internally.

When the PPI_CLK is not free-running, there may be additional latency cycles before data gets received or transmitted. In RX and TX modes, there may be at least 2 cycles latency before valid data is received or transmitted.

The PPI_CLK not only supplies the PPI module itself, but it also can clock one or more GP Timers to work synchronously with the PPI. Depending on PPI operation mode, the PPI_CLK can either equal or invert the TMRCLK input. For more information, see the *General-Purpose Timers* chapter.

Description of Operation

Table 20-1 shows all the possible modes of operation for the PPI.

Table 20-1. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	0
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	0
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	0
RX mode, ITU-R 656, active field only	embedded	0	00	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, vertical blanking only	embedded	0	00	10	0 or 1	0	0
RX mode, ITU-R 656, entire field	embedded	0	00	01	0 or 1	0	0
TX mode, 0 frame syncs	0	1	00	00	0 or 1	0 or 1	0
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	0
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	0

Table 20-1. PPI Possible Operating Modes (Cont'd)

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS1 assertion	3	1	01	11	0 or 1	0 or 1	0
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS2 assertion	3	1	11	11	0 or 1	0 or 1	0

Functional Description

The following sections describe the function of the PPI.

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 20-2, and Figure 20-3 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the horizontal (H), vertical (V), and field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The start of active video (SAV) and end of active video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a

0-to-1 transition of H. An entire field of video is comprised of active video + horizontal blanking (the space between an EAV and SAV code) and vertical blanking (the space where V = 1). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between field 1 and field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

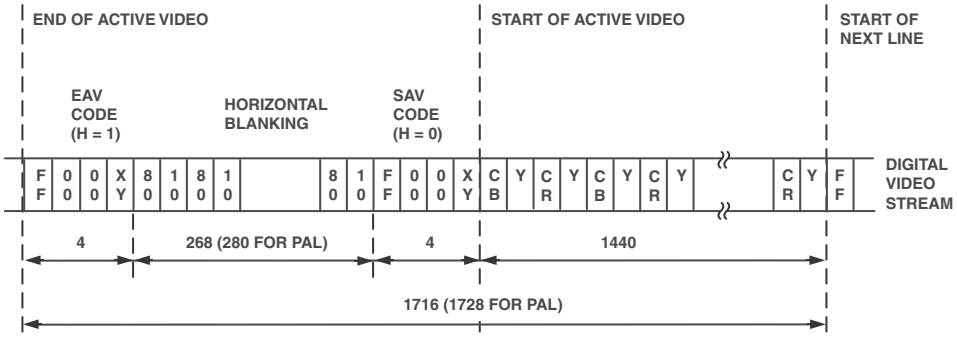
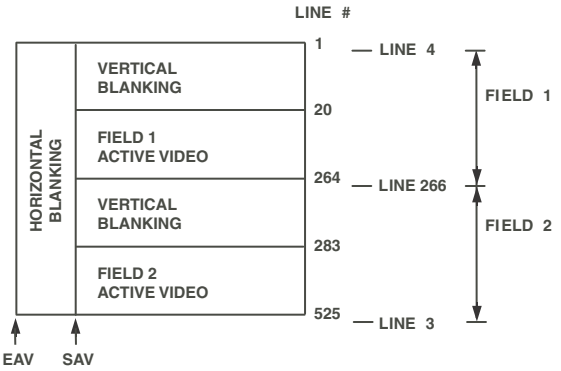
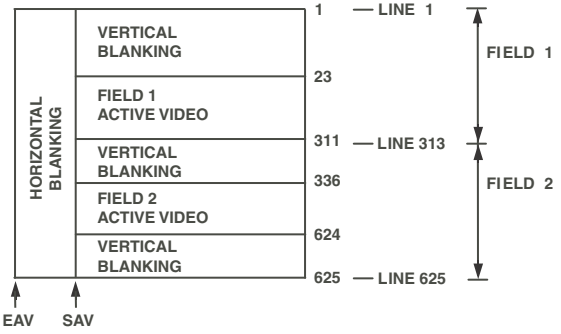


Figure 20-2. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 20-3. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 20-2. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY status word, which, aside from the F (field), V (vertical blanking) and H (horizontal blanking) bits, contains four protection bits for single-bit error detection and correction. Note F and V are only allowed to change as part of EAV sequences (that is, transition from H = 0 to H = 1). The bit definitions are as follows:

- F = 0 for field 1
- F = 1 for field 2

- V = 1 during vertical blanking
- V = 0 when not in vertical blanking
- H = 0 at SAV
- H = 1 at EAV
- P3 = V XOR H
- P2 = F XOR H
- P1 = F XOR V
- P0 = F XOR V XOR H

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the V and F codes can be used to delimit fields and frames.

Table 20-2. Control Byte Sequences for 8-Bit and 10-Bit ITU-R 656 Video

	8-bit Data								**10-bit Data**	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 20-4 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock `CLK` is either provided by the video source or supplied externally by the system.

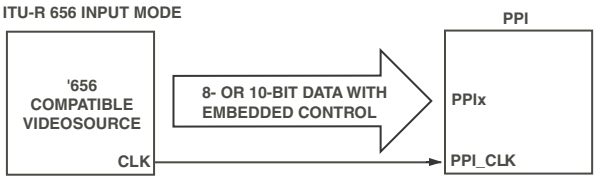


Figure 20-4. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 20-5 shows these three submodes.

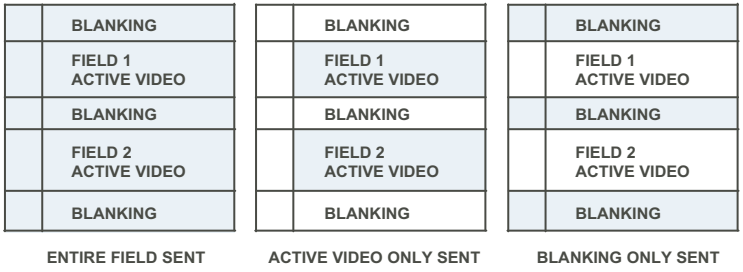


Figure 20-5. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes active video as well as control byte sequences and ancillary data that may be embedded in horizontal and vertical blanking intervals. Data transfer starts immediately after synchronization to field 1 occurs,

but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a "loopback" feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the PPI_FRAME MMR.

Vertical Blanking Interval (VBI) Only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the vertical blanking interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (active plus vertical blanking) per frame in the PPI_FRAME MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of field 1, which doesn't necessarily correspond to the start of vertical blanking. For instance, in 525/60 systems, the start of field 1 (F = 0) corresponds to line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of field 1. Consequently, up to two fields might be ignored (for example, if field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signaling is embedded in the datastream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that

occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the field status bit is toggled in the PPI_STATUS register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The general-purpose PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 20-3 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed functions.

Table 20-3. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input
RX mode, 2 or 3 external frame syncs	Input	Input	Input (if used)	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output

Table 20-3. General-Purpose PPI Modes (Cont'd)

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
TX mode, 2 external frame syncs	Input	Input	Not used	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Output

Figure 20-6 illustrates the general flow of the general purpose PPI modes. The top of the diagram shows an example of RX mode with one external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter reinitializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 20-6 shows an example of TX mode, one internal frame sync. After PPI_FS1 is asserted, there is a latency of one PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

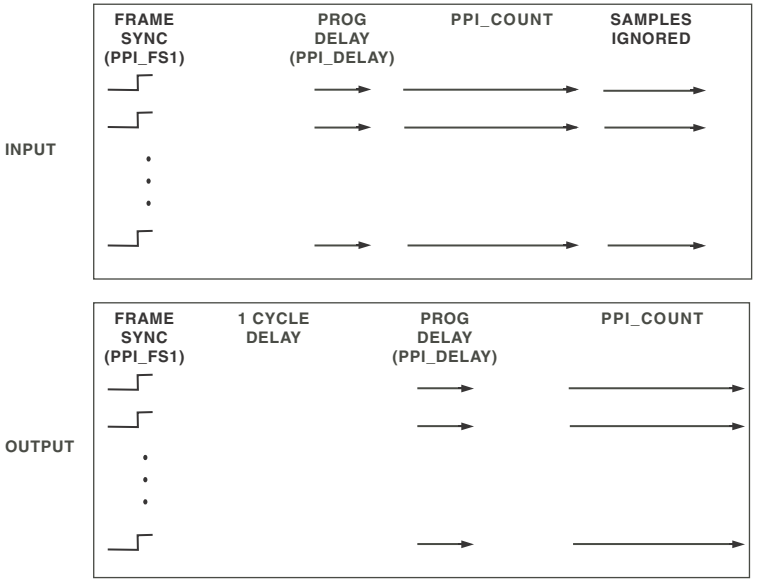


Figure 20-6. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 20-1 on page 20-4 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.
- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K byte memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least two PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The frame syncs are level-sensitive signals. The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 20-7 shows a typical illustration of the system setup for this mode.

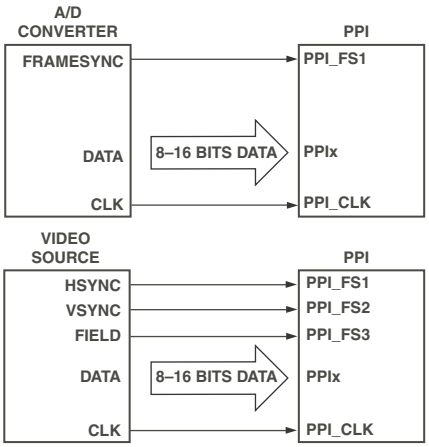


Figure 20-7. RX Mode, External Frame Syncs

The 3-sync mode shown at the bottom of Figure 20-7 supports video applications that use hardware signaling (HSYNC, VSYNC, FIELD) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is PPI_FS1 = HSYNC, PPI_FS2 = VSYNC, PPI_FS3 = FIELD. Please refer to "Frame Synchronization in GP Modes" on page 20-19 for more information about frame syncs in this mode.

A 2-sync mode is supported by not enabling the PPI_FS3 pin. See the *Product Specific Implementation* section for information on how this is achieved on this processor.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read

from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 20-8 shows a sample application for this mode.

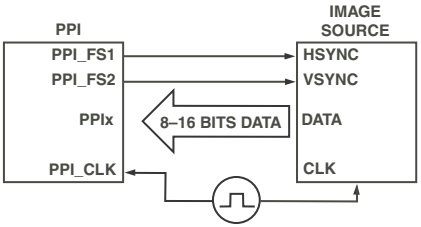


Figure 20-8. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 20-1 on page 20-4 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 20-9 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

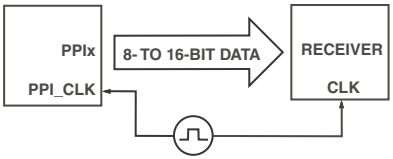


Figure 20-9. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 20-10 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 `PPI_CLK` cycles, plus the value programmed in `PPI_DELAY`, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

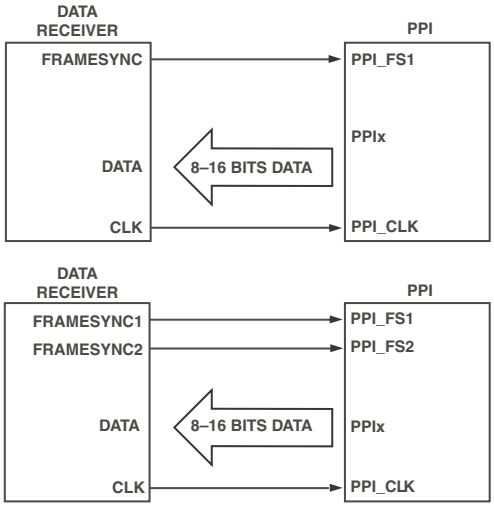


Figure 20-10. TX Mode, 1, or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 20-11 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 20-11. A 2-sync mode is implicitly supported by leaving PPI_FS3 unconnected in this case.

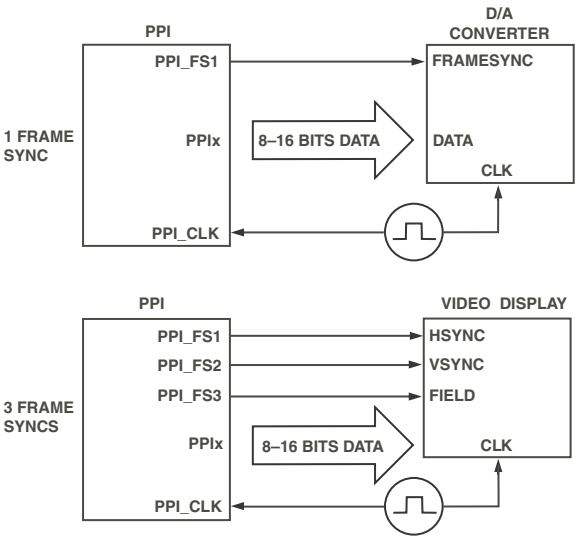


Figure 20-11. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in general purpose modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes With Internal Frame Syncs

In modes with internal frame syncs, PPI_FS1 and PPI_FS2 link directly to the pulsewidth modulation (PWM) circuits of general purpose timers. See

the *General-Purpose Timers* chapter for information on how this is achieved on this processor. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing TIMERx registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by PPI_CLK, not by SCLK (as during conventional timer PWM operation). If PPI_FS2 is not used in the configured PPI mode, its corresponding timer operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 20-22.
2. Configure the width and period for each frame sync signal via the appropriate TIMER_WIDTH and TIMER_PERIOD registers.
3. Set up the appropriate TIMER_CONFIG register(s) for PWM_OUT mode. This includes setting CLK_SEL to 1 and TIN_SEL to 1 for each timer involved.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable the appropriate timer(s).

It is important to guarantee proper frame sync polarity between the PPI and timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in the appropriate TIMER_CONFIG register(s). Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in the appropriate TIMER_CONFIG register(s).

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the appropriate timer(s) (using TIMER_DISABLE).

Modes With External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such modes the timers associated with the PPI_FS1 and PPI_FS2 pins can still be used for a purpose not involving the actual pin. However, timer access to a TMRx pin is disabled when the PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, the associated timer is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR1 pin becomes available for timer use as well). For more information on configuring and using the timers, please refer to the *General-Purpose Timers* chapter.

In RX mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMER_CONFIG. Then

the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see the *General-Purpose Timers* chapter.

Programming Model

The following sections describe the PPI programming model.

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, please refer to the *Direct Memory Access* chapter.

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of (PPI_CLK) × (16 bits/transfer). In modes where data lengths are greater than eight bits, only one element can be clocked in per PPI_CLK cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and PACK_EN = 1 (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA error occurs. In fact, the specification of the DMA_XCOUNT and DMA_YCOUNT MMRs allows for flexible data interrupt points. For example, assume the DMA registers XMODIFY = YMODIFY = 1. Then, if a data frame

contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 1 (the DI_SEL bit is located in DMA_CONFIG) interrupts on every row transferred, for the entire frame.
- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 0 interrupts only on the completion of the frame (when 240 rows of 320 bytes have been transferred).
- Setting XCOUNT = 38,400 (320 x 120), YCOUNT = 2, and DI_SEL = 1 causes an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

The general procedure for setting up DMA operation with the PPI follows.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.
5. If internally generated frame syncs are used, write to the TIMER_ENABLE register to enable the timers linked to the PPI frame syncs.

Figure 20-12 shows a flow diagram detailing the steps on how to configure the PPI for the various modes of operation.

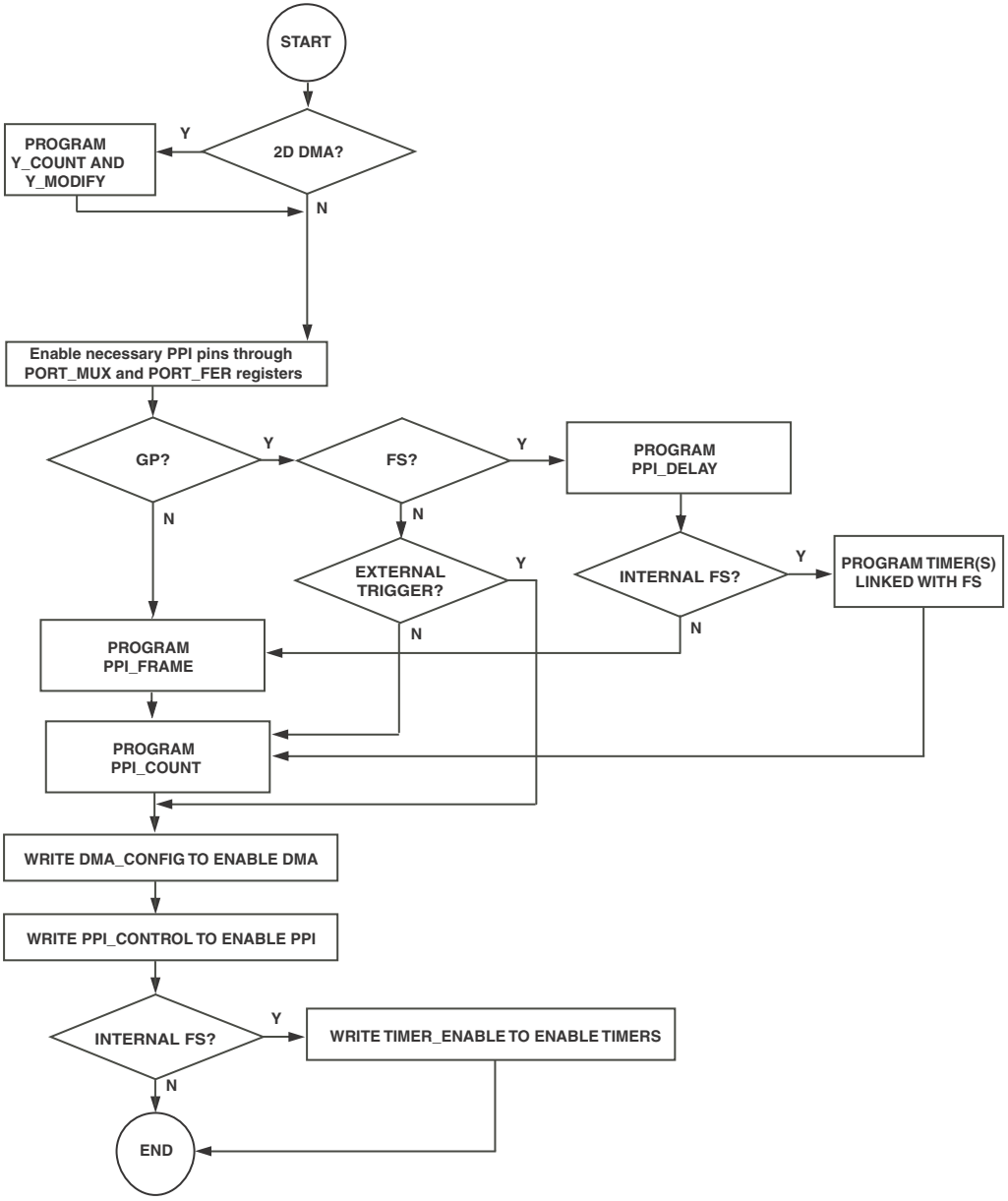


Figure 20-12. PPI Flow Diagram

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (PPI_CONTROL), the PPI status register (PPI_STATUS), the delay count register (PPI_DELAY), the transfer count register (PPI_COUNT), and the lines per frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI Control Register (PPI_CONTROL)

The PPI_CONTROL register configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 20-13 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any pins unused by the PPI as a result of the DLEN setting are free for use in their other functions.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

PPI Control Register (PPI_CONTROL)

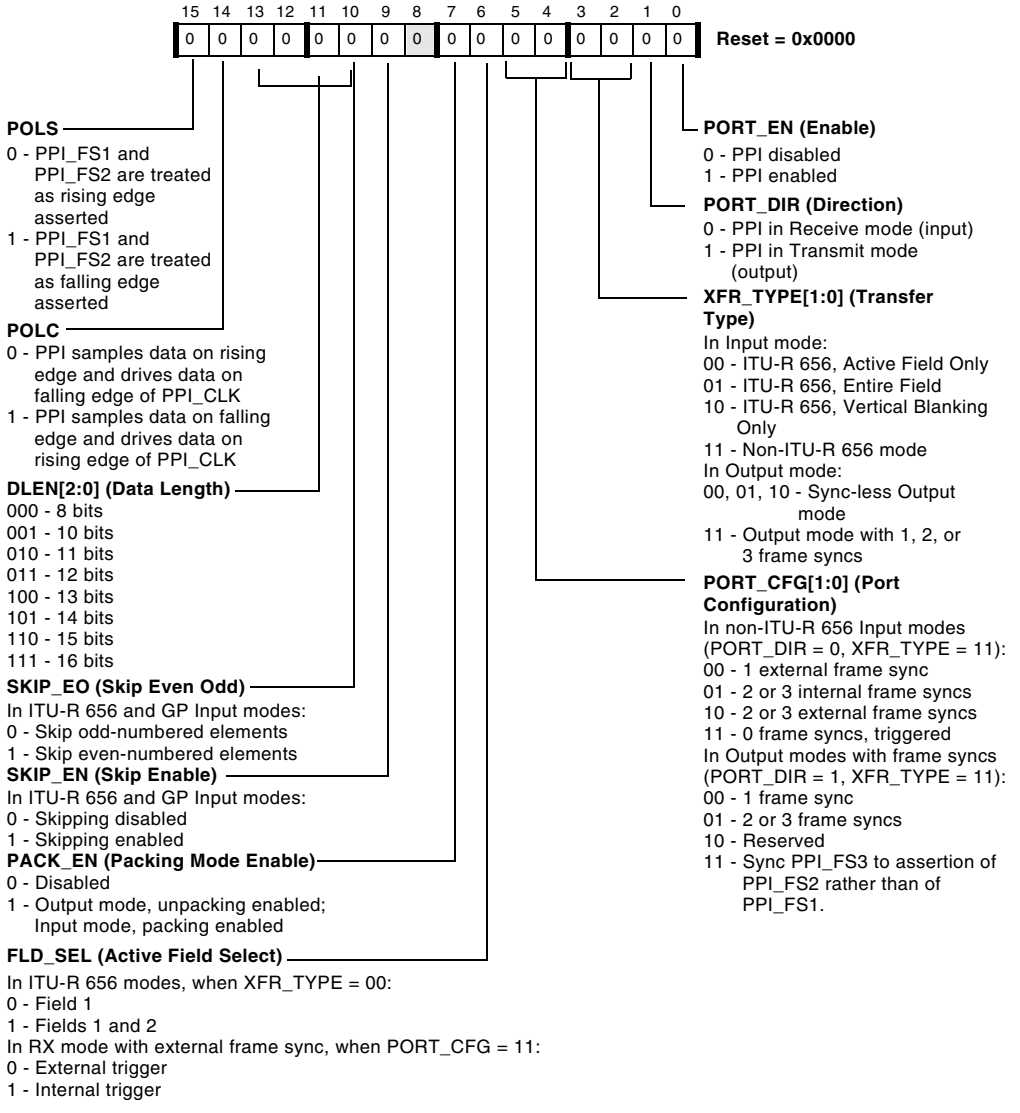


Figure 20-13. PPI Control Register

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is eight bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of ten bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of eight bits also results in a 16-bit input word, with the upper eight bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:
`0xCE, 0xFA, 0xFE, 0xCA....`

- With PACK_EN set:

  This is read into the PPI, configured for an 8-bit port width:
`0xCE, 0xFA, 0xFE, 0xCA...`

- This is transferred onto the DMA bus:
`0xFACE, 0xCAFE,...`

- With PACK_EN cleared:

  This is read into the PPI:
`0xCE, 0xFA, 0xFE, 0xCA,...`

- This is transferred onto the DMA bus:

```
0x00CE, 0x00FA, 0x00FE, 0x00CA,...
```

For TX modes, setting PACK_EN enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

```
0xFACE CAFE....
```

(0xFA and 0xCA are the two most significant bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMA'ed to the PPI:

```
0xFACE, 0xCAFE,...
```

- This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

```
0xCE, 0xFA, 0xFE, 0xCA,...
```

- With PACK_EN cleared:

  This is DMA'ed to the PPI:

```
0xFACE, 0xCAFE,...
```

- This is transferred out through the PPI, configured for an 8-bit port width:

```
0xCE, 0xFE,...
```

The FLD_SEL bit is used primarily in the active field only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only field 1 of each video frame, or both fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 20-1 on page 20-4 for the possible operating modes for the PPI.

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 20-1 on page 20-4 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

When configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 20-19 for more information.

PPI Status Register (PPI_STATUS)

The PPI_STATUS register, shown in Figure 20-14, contains bits that provide information about the current operating state of the PPI.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

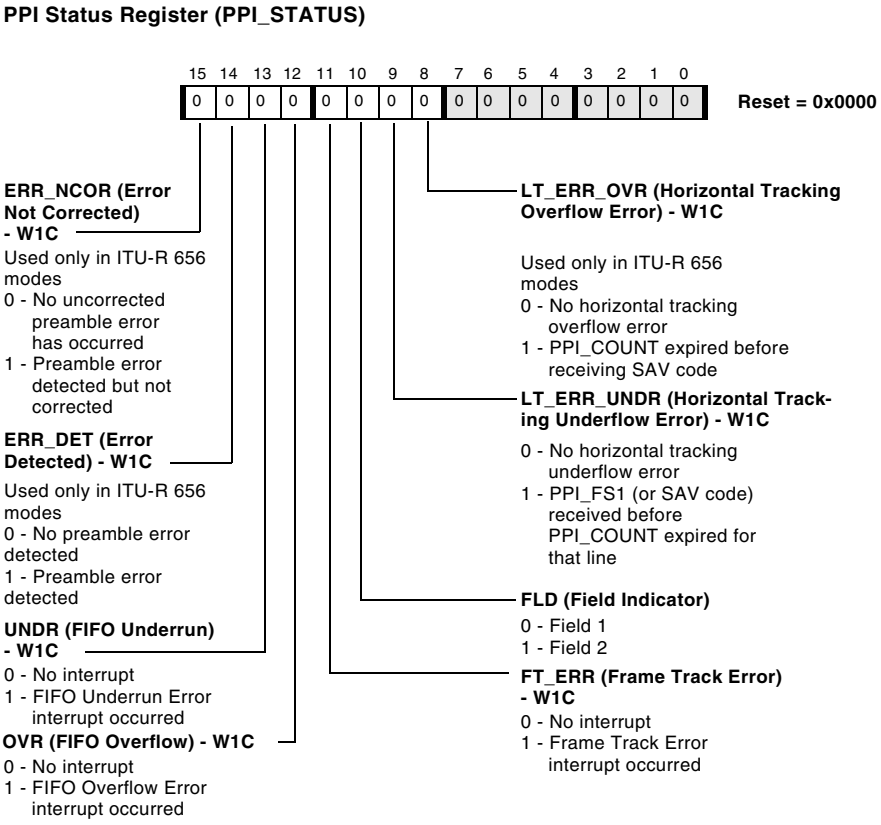


Figure 20-14. PPI Status Register

The FT_ERR bit is sticky and indicates, when set, that a frame track error has occurred. In this condition, the programmed number of lines per frame in PPI_FRAME does not match up with the “frame start detect” condition (see the information note on page 20-34). A frame track error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for input modes only. The state of FLD reflects the current state of the F or PPI_FS3

signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO overflow error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO underrun error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The LT_ERR_OVR and LT_ERR_UNDR bits are sticky and indicate, when set, that a line track error has occurred. These bits are valid for RX modes with recurring frame syncs only. If one of these bits is set, the programmed number of samples in PPI_COUNT did not match up with the actual number of samples counted between assertions of PPI_FS1 (for general-purpose modes) or start of active video (SAV) codes (for ITU-R 656 modes). If the PPI error interrupt is enabled in the SIC_IMASK register, an interrupt request is generated when one of these bits is set.

The LT_ERR_OVR flag signifies that a horizontal tracking overflow has occurred, where the value in PPI_COUNT was reached before a new SAV code was received. This flag does not apply for non ITU-R 656 modes; in this case, once the value in PPI_COUNT is reached, the PPI simply stops counting until receiving the next PPI_FS1 frame sync.

The LT_ERR_UNDR flag signifies that a horizontal tracking underflow has occurred, where a new SAV code or PPI_FS1 assertion occurred before the value in PPI_COUNT was reached.

PPI Delay Count Register (PPI_DELAY)

The PPI_DELAY register, shown in Figure 20-15, can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

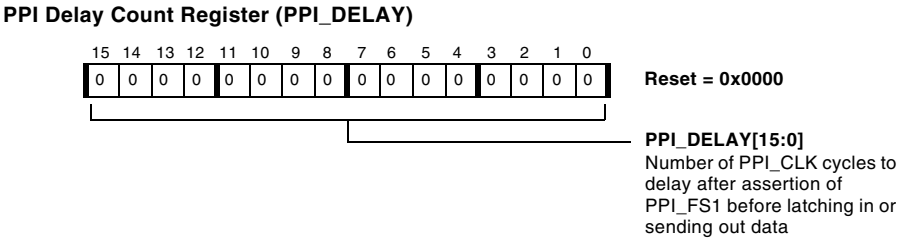


Figure 20-15. PPI Delay Count Register

PPI Transfer Count Register (PPI_COUNT)

The PPI_COUNT register, shown in Figure 20-16, is used in all modes except “RX mode with 0 frame syncs, external trigger” and “TX mode with 0 frame syncs.” For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the "horizontal" interval specified by PPI_FS1.

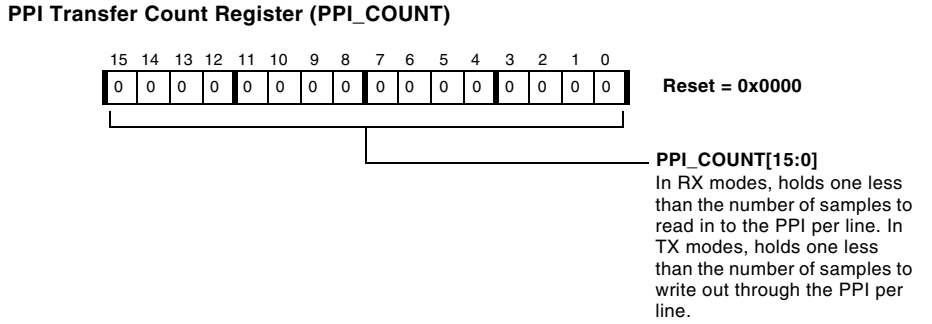


Figure 20-16. PPI Transfer Count Register

PPI Lines Per Frame Register (PPI_FRAME)

The PPI_FRAME register, shown in Figure 20-17, is used in all TX and RX modes with two or three frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as field 1 and field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original "frame start" each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a frame track error results, and the FT_ERR bit in PPI_STATUS is set.

However, the PPI still automatically reinitializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the field indicator. This occurs at the start of field 1.

In RX mode with three external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with three external frame syncs, and only two syncs are needed, configure the PPI for 3-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

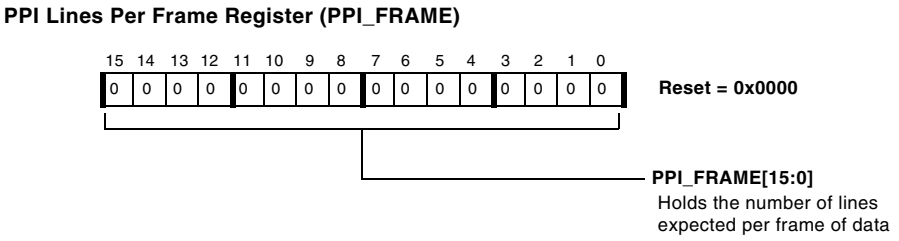


Figure 20-17. PPI Lines Per Frame Register

Programming Examples

The PPI can be configured to receive data from a video source in several RX modes. The following programming examples (Listing 20-1 through Listing 20-5) describe the ITU-R 656 entire field input mode.

Listing 20-1. Configure DMA Registers

```
config_dma:
/*Assumes PPI is mapped to DMA channel 0.*/
  /* DMA0_START_ADDR */
  R0.L = rx_buffer;
  R0.H = rx_buffer;
  P0.L = lo(DMA0_START_ADDR);
  P0.H = hi(DMA0_START_ADDR);
  [P0] = R0;

  /* DMA0_CONFIG */
  R0.L = DI_EN | WNR;
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  W[P0] = R0.L;

  /* DMA0_X_COUNT */
  R0.L = 256;
  P0.L = lo(DMA0_X_COUNT);
  P0.H = hi(DMA0_X_COUNT);
  W[P0] = R0.L;

  /* DMA0_X_MODIFY */
  R0.L = 0x0001;
  P0.L = lo(DMA0_X_MODIFY);
  P0.H = hi(DMA0_X_MODIFY);
  W[P0] = R0.L;
  ssync;
config_dma.END:  RTS;
```

Listing 20-2. Configure PPI Registers

```
config_ppi:

 /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = 0x0004;
  W[P0] = R0.L;
  ssync;

config_ppi.END:  RTS;
```

Listing 20-3. Enable DMA

```
  /* DMA0_CONFIG */
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 20-4. Enable PPI

```
  /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 20-5. Clear DMA Completion Interrupt

```
/* DMA0_IRQ_STATUS */
P2.L = lo(DMA0_IRQ_STATUS);
P2.H = hi(DMA0_IRQ_STATUS);
R2.L = W[P2];
BITSET(R2,0);
W[P2] = R2.L;
ssync;
```

Unique Information for the ADSP-BF50x Processor

None.

21 REMOVABLE STORAGE INTERFACE

This chapter describes the ADSP-BF50x Blackfin processor Removable Storage Interface (RSI) and includes the following sections:

Overview

ADSP-BF50x Blackfin processors provide an RSI interface for multimedia cards (MMC), secure digital memory cards (SD), secure digital input/output cards (SDIO) and consumer electronic ATA devices (CE-ATA). All of these storage solutions use similar interface protocols. The main difference between MMC and SD support is the initialization sequence. The main difference between SD and SDIO support is the use of interrupt and read wait signals for SDIO. CE-ATA devices require handling of larger block sizes of 4K bytes and implement a device interrupt scheme known as the command completion signal (CCS).

Features of the RSI interface include:

- Support for a single SD or SDIO card
- Support for one or more MMC cards (sharing the same interface)
- Support for 1- and 4-bit SD modes (SPI mode is not supported)
- Support for 1-, 4-, and 8-bit MMC modes (SPI mode is not supported)
- Support for 4- and 8-bit CE-ATA devices
- Programmable clock frequency generated from SCLK
- SDIO interrupt and read wait features
- Command Completion Signal recognition and disable for CE-ATA device support
- High-capacity card support such as SDHC implemented within software
- 512-bit transmit/receive FIFO
- DMA channel with 32-bit DMA Access Bus

Interface Overview

The RSI interface handles the multimedia and secure digital card functions. This includes clock generation, power management, command transfer, and data transfer. The bus interface converts 16-bit PAB accesses to 32-bit register accesses to the memory-mapped registers, and generates interrupt requests to the processor core and system. The RSI has two interrupt signals (IRQ0 and IRQ1) that are fed to the system interrupt controller (SIC) IRQ10 and IRQ55, respectively.

The RSI block has 22 individual status bits contained within the RSI_STATUS register that can be configured to generate an interrupt. The status bits may be mapped to either of the two interrupts fed to the system interrupt controller, allowing for greater flexibility in system configuration. In order for an interrupt to be generated on IRQ0, the interrupt should be enabled by setting the corresponding bit in the RSI_MASK0 register. Interrupts that are required to be generated on IRQ1 are enabled by setting the corresponding bit in the RSI_MASK1 register. In addition to status flags within the RSI_STATUS register being capable of generating interrupts, each of the flags in the RSI_ESTAT register are also capable of generating an interrupt. Interrupts for the RSI_ESTAT flags are enabled by setting the corresponding bit in the RSI_EMASK register and are sent to the SIC via IRQ10.

The 32-bit DAB bus allows for efficient transfer of data, both to and from internal memory, via DMA channel 4 that is shared with the SPORT0 TX. The peripheral used by this DMA channel is determined by the peripheral that is enabled via the pin multiplexing.

The RSI (Figure 21-1) is a 10-pin interface consisting of:

- RSI_CLK: The clock signal applied to the card from the RSI. All transfers on the command and data signals are synchronous to this signal. The frequency is variable between zero and the maximum clock frequency. Refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for maximum supported clock frequencies.
- RSI_CMD: A bi-directional command signal used for command transfer and card initialization. The RSI drives this signal to send commands to the cards, and the card drives the signal to send responses back to the RSI. This signal is configurable for both push-pull mode and open-drain mode. MMC cards are the only

cards to support open-drain mode. This allows multiple MMC cards to share the data and command signals on the RSI interface and allows for the initialization sequence to take place on all cards.

- `RSI_DATA7-0`: These are configurable bi-directional data channels used for all data transfers both to and from the card. The data bus width can be configured as 1-, 4-, or 8-bit.

Although multiple MMC cards may be bused together to the single RSI interface, it is not possible to bus together an MMC card with an SD or SDIO such that they share the command and or data signals. Multiple MMC cards bused together respond to `CMD1` and `CMD2` commands simultaneously using the open drain drivers. For other card types, broadcast commands with a response must not be issued if the command or data signals are shared between cards.

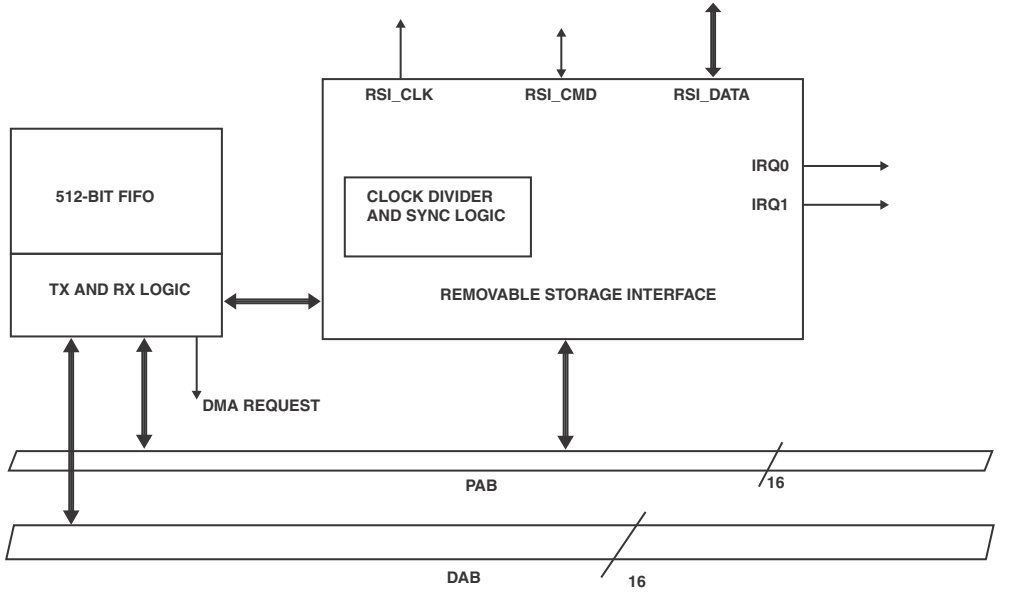


Figure 21-1. RSI Block Diagram

Table 21-1 and Table 21-2 list the RSI interface pins functional operations for all supported protocol modes.

Table 21-1. RSI Protocol Interface

Signal Name	MMC (1-bit)	MMC (4-bit)	MMC (8-bit)	CE-ATA (4-bit)	CE-ATA (8-bit)	Direction
RSI_DATA7	Not Used	Not Used	Dat7	Not Used	Dat7	Bi-dir.
RSI_DATA6	Not Used	Not Used	Dat6	Not Used	Dat6	Bi-dir.
RSI_DATA7	Not Used	Not Used	Dat7	Not Used	Dat7	Bi-dir.
RSI_DATA6	Not Used	Not Used	Dat6	Not Used	Dat6	Bi-dir.
RSI_DATA5	Not Used	Not Used	Dat5	Not Used	Dat5	Bi-dir.
RSI_DATA4	Not Used	Not Used	Dat4	Not Used	Dat4	Bi-dir.
RSI_DATA3	Not Used / Card Detect	Dat3/ Card Detect	Dat3/ Card Detect	Dat3	Dat3	Bi-dir.
RSI_DATA2	Not Used	Dat2	Dat2	Dat2	Dat2	Bi-dir.
RSI_DATA1	Not Used	Dat1	Dat1	Dat1	Dat1	Bi-dir.
RSI_DATA0	Dat0	Dat0	Dat0	Dat0	Dat0	Bi-dir.
RSI_CMD	Command/ Response	Command/ Response	Command/ Response	Command/ Response/ CCS/ CCSD	Command/ Response/ CCS/ CCSD	Bi-dir.
RSI_CLK	CLK	CLK	CLK	CLK	CLK	Output

Table 21-2. RSI Protocol Interface

Signal Name	SD (1-bit)	SD (4-bit)	SDIO (1-bit)	SDIO (4-bit)	Direction
RSI_DATA7	Not Used	Not Used	Not Used	Not Used	Bi-directional
RSI_DATA6	Not Used	Not Used	Not Used	Not Used	Bi-directional
RSI_DATA5	Not Used	Not Used	Not Used	Not Used	Bi-directional
RSI_DATA4	Not Used	Not Used	Not Used	Not Used	Bi-directional
RSI_DATA3	Not Used/ Card Detect	Dat3/ Card Detect	Not Used/ Card Detect	Dat3/ Card Detect	Bi-directional
RSI_DATA2	Not Used	Dat2	Read Wait	Dat2/ Read Wait	Bi-directional
RSI_DATA1	Not Used	Dat1	Interrupt	Dat1/ Interrupt	Bi-directional
RSI_DATA0	Dat0	Dat0	Dat0	Dat0	Bi-directional
RSI_CMD	Command/ Response	Command/ Response	Command	Command	Bi-directional
RSI_CLK	CLK	CLK	CLK	CLK	Output

Description of Operation

The RSI controller is a fast, synchronous peripheral that uses various protocols to communicate with MMC, SD, and SDIO cards as well as CE-ATA hard drives. The RSI is compatible with the following protocols:

- MMC (Multimedia Card) bus protocol
- SD (Secure Digital) bus protocol
- SDIO (Secure Digital Input Output) bus protocol
- CE-ATA (Consumer Electronic ATA)

The RSI does not support the SPI bus protocol.

Communication is via a master and slave type configuration, whereby the RSI is the master and the card is the slave device. The RSI communicates with the device via a message-based bus protocol in which the host sends commands serially via the RSI_CMD signal. Certain commands require the card to provide a response back to the host. This response is also sent serially via the RSI_CMD signal.

Data transfers, both to and from the card, occur via RSI_DATAx signals. The number of data lines used for the data transfer can be configured to 1, 4, or 8 using RSI_DATA0, RSI_DATA3-0, or RSI_DATA7-0, respectively. All transfers over the RSI_CMD and RSI_DATAx signals are transferred synchronously to the RSI_CLK.

Commands, responses, and data transfers are protected from transmission errors with the use of cyclic redundancy codes (CRC). A CRC7 code is generated for every command sent by the host and for almost every response returned by the card on the RSI_CMD signal. A CRC16 code is used to protect block data transfers sent over the RSI_DATAx signals. In 4- and 8-bit bus configurations, the CRC16 is calculated for each individual data signal.

When a device connected to the RSI is first powered and detected by the host or has been reset, the device must first be identified and initialized by the host. This allows the software to determine whether the device is compatible with the RSI controller and the implemented software drivers. This phase in the procedure is known commonly as the *card identification mode*.

When a device is in card identification mode, the host may be required to perform the following actions:

- Reset the device
- Validate the device operating voltage range

- Identify the device type,
- Assign/request a relative card address (RCA)

Only once a device has been assigned an RCA will the device then transition to a stand-by state, where it is then known to be in *data transfer mode*. Only once the device has entered this mode may data transfers then take place. All communication during the card identification phase between the host and the attached device occur via the RSI_CMD signal. The maximum clock frequencies during this identification phase may typically be far lower than the cards maximum operating frequency for data transfers.

Once the device is in data transfer mode, communication may take place via the RSI_CMD and the RSI_DATAx signals. The card may be interrogated to then identify further supported features such as supported bus widths, maximum supported clock frequency, and the device capacity. At this point the bus width may then be altered and the supplied clock frequency increased.

Data may be written to the device or read from the device using the following two methods:

- Stream reads and writes
- Block reads and writes

Stream transfers result in a continual stream of data being transferred until a specific command is sent to the device by the RSI informing the device to stop the transfer. There may be additional maximum operating frequency limitations imposed by the device for stream read and write operations. In addition, stream write operations may have restrictions that are dependent on writable block boundaries.

Block-based transfers result in a block of a pre-configured size being transferred. The size of a block is dependent upon the device and can be obtained by reading registers contained on the device that are read during the device detection procedure.

Functional Description

The following sections describe the functions and features of the RSI controller as well as the MMC, SD, SDIO, and CE-ATA protocols. For detailed information on timing parameters and protocol requirements, refer to the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* and the following standards and specifications:

- MMCA System Specification
- JESD84 series of JEDEC standards
- SD Specifications Part 1 Physical Layer Specification
- SD Specifications Part 1 Physical Layer Simplified Specification
- SD Specifications Part E1 SDIO Specification

RSI Clock Configuration

The RSI is a fast, synchronous peripheral with a programmable clock frequency that is supplied via the RSI_CLK signal. The interface between the RSI and the PAB/DAB busses operates at SCLK frequency. Communication between the clock domain that is supplied externally from the RSI on the RSI_CLK signal and the internal RSI access to the PAB and DAB busses is accomplished using synchronizers in the RSI module. The RSI_CLK frequency is configured via the 8-bit CLKDIV field and the CLKDIV_BYPASS bit of the RSI_CLK_CONTROL register (see "RSI Clock Control Register (RSI_CLK_CONTROL)" on page 21-54).

If CLKDIV_BYPASS is set, the clock frequency driven on the RSI_CLK signal is derived directly from SCLK.

If CLKDIV_BYPASS is cleared, the clock divider logic provides an RSI_CLK frequency, where CLKDIV is an 8-bit value ranging between 0 and 255.

$$RSI\_CLK = \frac{SCLK}{2 \times (CLKDIV + 1)}$$

The RSI_CLK output is enabled or disabled via the CLK_EN bit in the RSI_CLK_CONTROL register and a power save feature is implemented via PWR_SV_EN that allows for the disabling of the RSI_CLK output when there are no transfers taking place on the RSI interface.

RSI Interface Configuration

The RSI supports multiple card types via the various protocols. Different card types may require slightly different interface configurations.

The command signal on MMC cards operates in two different modes depending on the cards operating mode. During the card identification mode, this signal operates in open-drain configuration, however upon the cards entry to data transfer mode, the signal is then configured to push-pull mode. The internal pull-up resistor of the RSI_CMD signal is only intended to keep the signal from floating. The internal pull-up resistor is not sufficient during the card identification phase when the MMC card RSI_CMD signal is operating in open-drain mode. If support for MMC devices is required, an external pull-up resistor should be added to the SD_CMD signal as detailed in the JEDEC standard. The bus width used for the data transfers is configurable to 1-bit, 4-bits, or 8-bits via the BUS_MODE field in the RSI_CLK_CONTROL register (see "RSI Clock Control Register (RSI_CLK_CONTROL)" on page 21-54).

In order to stop the signals from floating when no card is inserted or during times when all card drivers are in a high-impedance mode, various pull-up and pull-down resistor configurations can be enabled on the

RSI_DATAx signals. The RSI_CONFIG register (see "RSI Configuration Register (RSI_CONFIG)" on page 21-77) allows for the following options:

- Enable or disable a pull-down resistor on the RSI_DATA3 signal
- Enable or disable a pull-up resistor on the RSI_DATA3 signal
- Enable or disable pull-up resistors on the RSI_DATA7 through RSI_DATA4, RSI_DATA2 through RSI_DATA0 signals and the RSI_CMD signal

Card Detection

The RSI allows for software to detect when a card is inserted into its slot. There are a number of ways that this card detection can be performed.

SD and SDIO cards use an internal pull-up resistor on the RSI_DATA3 line as a card detect signal to indicate to the host that a card is present. In order to use the RSI_DATA3 signal for card detection, an external pull-down resistor should be added to the pin in order to pull the pin low during the time a card is not inserted. When a card is inserted into the slot, a rising edge is detected on RSI_DATA3 by the RSI and SD_CARD_DET is set within the RSI_ESTAT register. Once the card has been correctly identified, the SD_CARD_DET interrupt should be cleared and disabled then the pull-up resistor within the SD card should be disabled by issuing the SET_CLR_CARD_DETECT command.

When using the RSI_DATA3 signal for card detection with an external pull-down resistor, do not enable the internal pull-up resistor by setting PU_DAT3.

The recommended method of detecting the insertion of a card is to use the card detect feature that is made available through most sockets. Sockets supporting this feature can have the card detect pin de-bounced and connected to a GPIO pin in order to allow not only interrupt-driven card detection but also interrupt-driven card removal. This is the most

reliable and efficient method of detecting the insertion and removal of a card as some MMC devices may not implement the card detect pull-up resistor on the RSI_DATA3 signal. Once a card is detected, the GPIO pin can have the interrupt level inverted to then generate an interrupt on card removal.

The final approach to detecting the insertion and removal of a card is to simply use software polling. Software would poll the slot periodically using the card identification commands for the supported card types. Once a card is inserted, this will result in valid responses being sent back to the host; when the card is removed, command and data timeout errors will be observed.

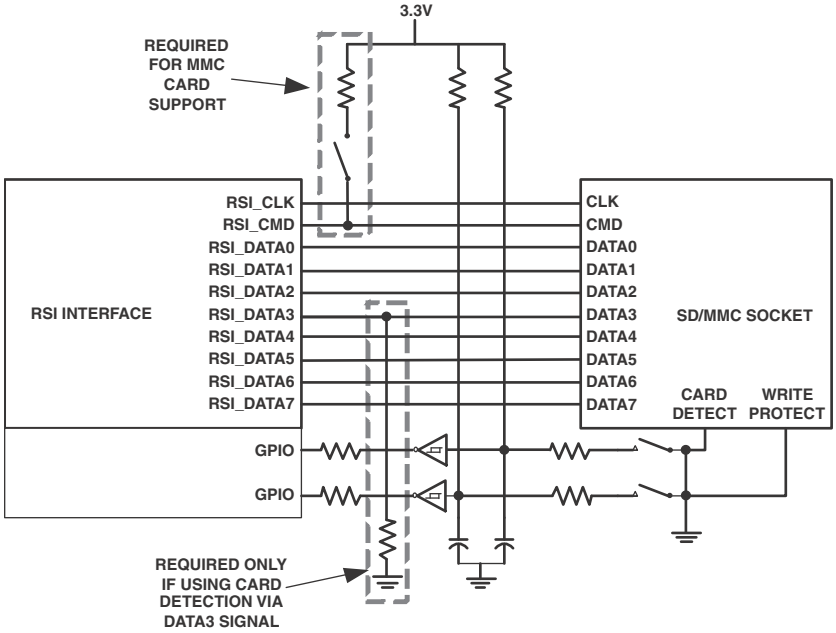


Figure 21-2. RSI Socket Interface

RSI Power Saving Configuration

The RSI requires two internal clock signals that are derived directly from SCLK. In order for the RSI to function, these clocks must be enabled via RSI_CLK_EN in the RSI_CONFIG register. Clearing RSI_CLK_EN disables the RSI regardless of the other RSI clock configurations. One of these clock signals is routed to the clock divider and generates the clock that is provided on the RSI_CLK signal. The RSI_CLK signal can be enabled or disabled via CLK_EN in the RSI_CLK_CONTROL register and a power save feature is implemented via PWR_SV_EN that allows for the disabling of the RSI_CLK output when there are no transfers taking place on the RSI interface, providing additional power saving options.

Table 21-3. RSI Power Saving Configurations

CLKS_EN	CLK_EN	PWR_SV_E	RSI State	RSI_CLK output
0	0	0	Disabled	No clock
0	0	1	Disabled	No clock
0	1	0	Disabled	No clock
0	1	1	Disabled	No clock
1	0	0	Enabled	No clock
1	0	1	Enabled	No clock
1	1	0	Enabled	Continuous clock[1]
1	1	1	Enabled	Clock only driven during transfers[1]

1 The PWR_ON field of the RSI_PWR_CTL register must be set to 0x3. If PWR_ON is 0x0, the clock will not be output.

RSI Commands and Responses

The RSI sends commands to and receives responses from the card via the RSI_CMD signal. The command to be sent to the card is issued by writing to the RSI_COMMAND register (see "RSI Command Register (RSI_COMMAND)" on page 21-56). This register contains a 6-bit

CMD_IDX field that contains the command index to be sent to the card providing support for a total of 64 commands, 0 (CMD0) to 63 (CMD63). Some commands require an argument to be sent along with the command, such as an address for a read transaction. An argument is always sent with the command and it is the responsibility of the card to either ignore or use the argument field based on the command that is received. The argument sent with the command is provided via the RSI_ARGUMENT register (see "RSI Argument Register (RSI_ARGUMENT)" on page 21-56).

All command transfers are protected by a 7-bit cyclic redundancy check (CRC) code, more commonly referred to as a CRC7 checksum. This allows for transmission errors to be detected and the command to be re-issued to the card in the event of an error. All commands sent to the card are composed of 48-bits as shown in Table 21-4.

Table 21-4. RSI Command Format

Bit Position	Width	Value	Description
47	1	0	Start bit
46	1	1	Transmitter bit
45:40	6	-	Command index
39:8	32	-	Argument
7:1	7	-	CRC7 checksum
0	1	1	End bit

The RSI_COMMAND register, as well as providing a means for issuing the required command, also provides configuration information on whether a response is to be expected back from the card and the type of response.

The RSI can be configured via the CMD_RESP and CMD_L_RESP fields of the RSI_COMMAND register to expect the following response types:

- No response
- Short response (see Table 21-5)
- Long response (see Table 21-6)

Table 21-5. RSI Short Response Format

Bit Position	Width	Value	Description
47	1	0	Start bit
46	1	0	Transmitter bit
45:40	6	-	Command index or check bits[1]
39:8	32	-	Card status, register contents or argument field
7:1	7	-	CRC7 checksum or check bits[2]
0	1	1	End bit

1 Responses that do not contain the command index have a check bits field that contains “111111”.
2 Responses that do not contain a CRC7 checksum have a check bits field that contains “1111111”.

Table 21-6. RSI Long Response Format

Bit Position	Width	Value	Description
135	1	0	Start bit
134	1	0	Transmitter bit
133:128	6	111111	Check bits
127:1	127	-	Register contents including internal CRC7
0	1	1	End bit

Like the commands, all responses are sent on the RSI_CMD signal.
A response always has a “0” start bit followed by a “0” transmission bit

to indicate the transfer is from card to host. Unlike the commands issued by the host, not all responses are protected by a CRC7 checksum. Refer to the appropriate specification for full details on the response formats and whether they are protected by a CRC7 checksum.

When a short response is received, the response is broken down by the RSI and the 32-bit field containing bits 39:8 of the 48-bit response is stored to RSI_RESPONSE0, where bit 39 of the response corresponds to bit 31 of RSI_RESPONSE0 and bit 8 of the response to bit 0 of RSI_RESPONSE0. Bits 45:40 of the response are stored to the RESP_CMD field of the RSI_RESP_CMD register.

For a long response, bits 127:1 of the response are stored in RSI_RESPONE0-3, where bit 31 of RSI_RESPONSE0 contains the most significant bit (bit 127) of the response and bit 0 of RSI_RESPONSE3 contains bit 1 of the response. Bit 31 of RSI_RESPONSE3 is always zero.

Figure 21-3 shows the command path state machine. In order for the state machine to be active, the RSI must be enabled via RSI_CLK_EN. Disabling the clocks to the RSI will result in the state machine returning to the IDLE state. The command path state machine is responsible for setting and clearing a number of status flags within the RSI_STATUS register (see "RSI Status Register (RSI_STATUS)" on page 21-64).

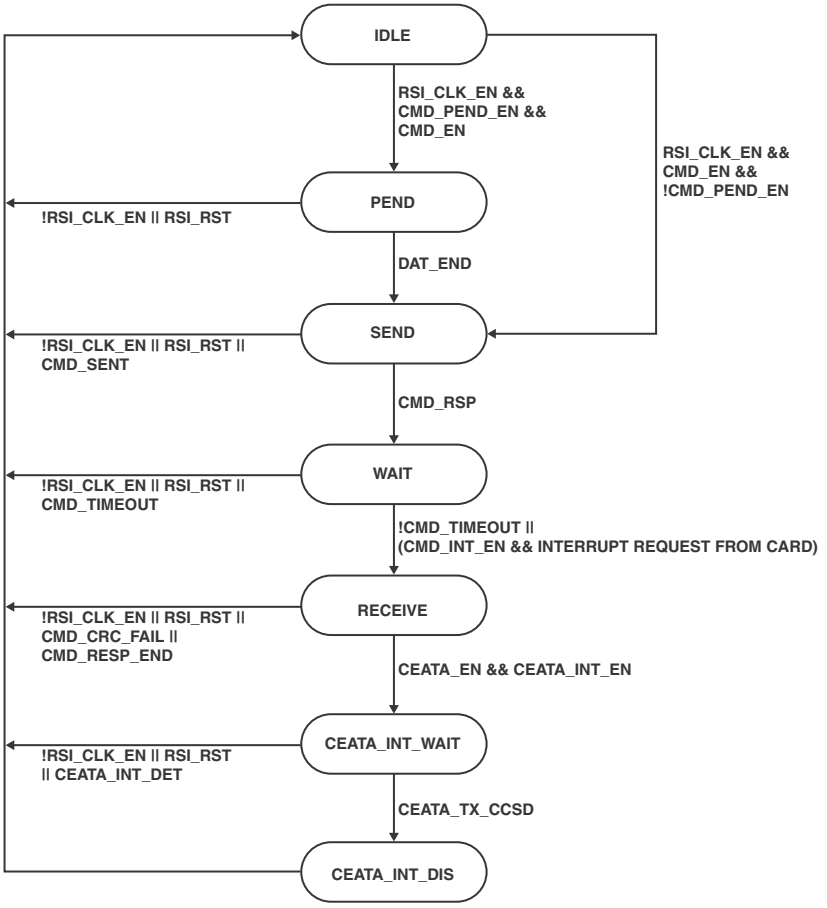


Figure 21-3. RSI Command Path State Machine

Table 21-7 lists the status flags and exception flags that are affected by the command path state machine.

Table 21-7. RSI Command Path Status Flags

RSI_STATUS Flag	Description	State Flag Set in
CMD_ACT	Command transfer is in progress	WAIT_S
CMD_SENT	Command without response sent successfully	SEND
CMD_TIMEOUT	Response timeout occurred (64 RSI_CLK cycles)	WAIT_S
CMD_CRC_FAIL	Response CRC failure	RECEIVE
CMD_RESP_END	Response CRC successful	RECEIVE
CEATA_INT_DET	CE-ATA command completion signal detected	CEATA_INT_WAIT

The command path operates in a half-duplex mode, so that commands and responses can either be sent or received. If the state machine is not in the SEND state, the RSI_CMD output is in high impedance state. Figure 21-4 describes a typical command and response transfer, the RSI_CMD signal is sampled by the card and the host on the rising edge of RSI_CLK.

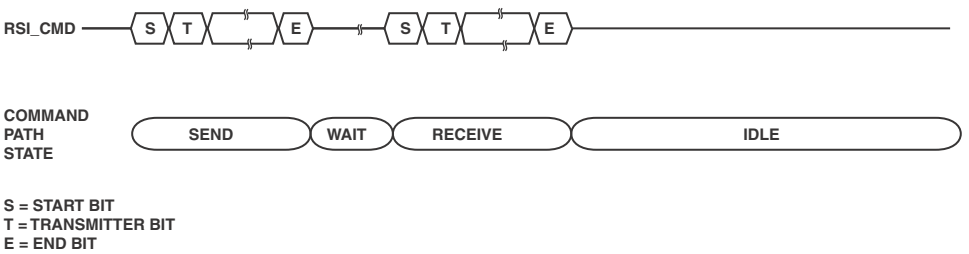


Figure 21-4. RSI Command Transfer

The following sections describes the RSI command path states.

IDLE State

The command path state machine remains in the IDLE state when not active. The command path state machine becomes enabled and leaves the IDLE state once the CMD_EN bit of the RSI_COMMAND register is set. The state will transition to the PEND state if the CMD_PEND_EN bit is set within RSI_COMMAND; otherwise it will enter the SEND state.

When the command path state machine returns to the IDLE state from one of the other states and the result of the return is not due to the RSI being disabled or reset, the state machine will remain in the IDLE state for at least eight RSI_CLK cycles. During this time, the RSI will continue to drive the RSI_CLK signal even if the PWR_SV_EN feature is enabled. This is required in order to allow the card to complete the current operation. Only after the eight RSI_CLK cycles have passed will the state machine leave the IDLE state if enabled again.

PEND State

The PEND state is entered if the CMD_PEND_EN bit is set within RSI_COMMAND. The state machine remains in the PEND state until it is notified by the data path sub block that a data transfer has completed. This is indicated by the DAT_END flag being set as a result of the RSI_DATA_CNT decrementing to zero. This mode is intended to enable the automatic transmission of a STOP_TRANSMISSION command after reading or writing the required amount of data for stream-based transactions.

The CMD_PEND_EN feature is not functional for block-based transfers and cannot be used to automatically issue the STOP_TRANSMISSION command for MULTIPLE_BLOCK_READ or MULTIPLE_BLOCK_WRITE operations.

SEND State

During the SEND state, the RSI sets the CMD_ACT flag in CMD_STATUS to indicate a transfer is in progress. The behavior of the state machine upon

completion of sending the command depends upon whether the command expects a response back from the card. If no response is expected, the RSI clears the CMD_ACT flag and sets the CMD_SENT flag to indicate that a command operation without a response has been completed and then the state transitions to the IDLE state. If a response is expected, the RSI enters the WAIT state.

WAIT State

Upon entering the WAIT state, the RSI awaits the response to be received on the RSI_CMD signal. Upon entering this state, an internal timer starts running. If the response is not received within 64 RSI_CLK cycles (max. NCR), the CMD_TIMEOUT flag is set and the CMD_ACT flag is cleared. The state machine then enters the IDLE state, awaiting the next action. If a response is detected as being sent back from the card as indicated by the "0" start bit on the RSI_CMD signal, the RSI transitions to the RECEIVE state to receive a 48- or 136-bit response.

The WAIT state is also capable of detecting card interrupts. This is an optional feature that applies only to MMC cards. This feature is enabled by setting the CMD_INT_EN bit within RSI_COMMAND. When CMD_INT_EN is set, the timeout timer that is normally started upon entry to the WAIT state is disabled. The RSI remains in this state until a card interrupt is detected. Cards that implement this interrupt feature may have functions that result in the response being delayed and triggered by some internal event in the card. Once the event is triggered the card then sends the response. The RSI then detects this start bit of the response then proceeds to the RECEIVE state.

RECEIVE State

The RSI reads in the response from the card on the RSI_CMD signal when in the RECEIVE state. Upon receiving either the short or long response, the CMD_ACT flag is cleared and the CMD_RESP_END flag is set if the response passed the CRC check. A CRC failure in the response results in the

CMD_CRC_FAIL flag being set. At this point the state machine then transitions to the IDLE state.

Some CE-ATA commands require additional functionality upon reaching this state. This additional functionality requires sending a command completion signal back to the host upon completion of a specific task. For commands that require this functionality, the CEATA_EN and CEATA_CCS_EN bits of the RSI_DATA_CONTROL register should be set prior to enabling the command path state machine. After receiving the response, the state machine then enters the CEATA_INT_WAIT state.

CEATA_INT_WAIT State

Upon entering this state, the RSI waits for the CE-ATA device to issue the command completion signal. This is indicated by the device sending a "0" on the RSI_CMD signal. Upon detection of the command completion signal, the CMD_ACT flag is cleared and the CEATA_INT_DET flag of the RSI_ESTAT register is set. Alternatively, the command completion signal of the CE-ATA device can be disabled by the RSI. This action is performed by setting the CEATA_TX_CCSD bit of the RSI_CEATA_CONTROL register, at which point the state machine enters the CEATA_INT_DIS state. The CEATA_TX_CCSD bit can be set prior to enabling the command path state machine. This will result in the CCSD sequence being issued after the response is received rather than having to wait for the response then setting this CEATA_TX_CCSD.

CEATA_INT_DIS State

Upon entering this state, the RSI issues the command completion signal disable sequence on the RSI_CMD signal before then transitioning to the IDLE state and clearing the CMD_ACT flag. The command completion signal disable sequence issued is the binary sequence "00001".

RSI Command Path CRC

The command CRC generator of the RSI calculates the 7-bit CRC checksum for all 40 bits preceding the CRC code for both 48-bit commands and 48-bit responses. This includes the start bit, transmitter bit, command index, and command argument (or card status). The 7-bit CRC checksum is calculated for the first 120 bits of the register contents field for the long response format. Note that the start bit, transmitter bit, and the six check bits are not used in the CRC calculation for the long response. The command and response CRC checksum is a 7-bit value that is calculated as follows:

$$CRC[6:0] = Remainder \qquad \frac{x_1 \cdot M(x)}{G(x)}$$

with:

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

and for a short response:

$$M(x) = x_{39} \cdot (start\ bit) + \ldots + x_0 \cdot (last\ bit\ before\ CRC)$$

or for a long response:

$$M(x) = x_{19} \cdot (start\ bit) + \ldots + x_0 \cdot (last\ bit\ before\ CRC)$$

RSI Data

Data transfers both to and from the RSI take place over the RSI data bus signals `RSI_DATA7-0`. The RSI data bus width is configured via the `BUS_MODE` field of the `RSI_CLK_CONTROL` register (see "RSI Clock Control

Register (RSI_CLK_CONTROL)" on page 21-54). The default configuration is for 1-bit bus mode, whereby the data is transferred over the RSI_DATA0 signal. Alternatively, 4-bit mode or 8-bit mode may be enabled after configuring the card for 4-bit or 8-bit mode of operation, respectively. The RSI has a data path state machine that operates at RSI_CLK frequency. The state machine becomes enabled and leaves the IDLE state when the DATA_EN field of RSI_DATA_CONTROL is set, enabling the data transfer. The state entered upon leaving the IDLE state is determined by DATA_DIR. The data path state machine is shown in Figure 21-5.

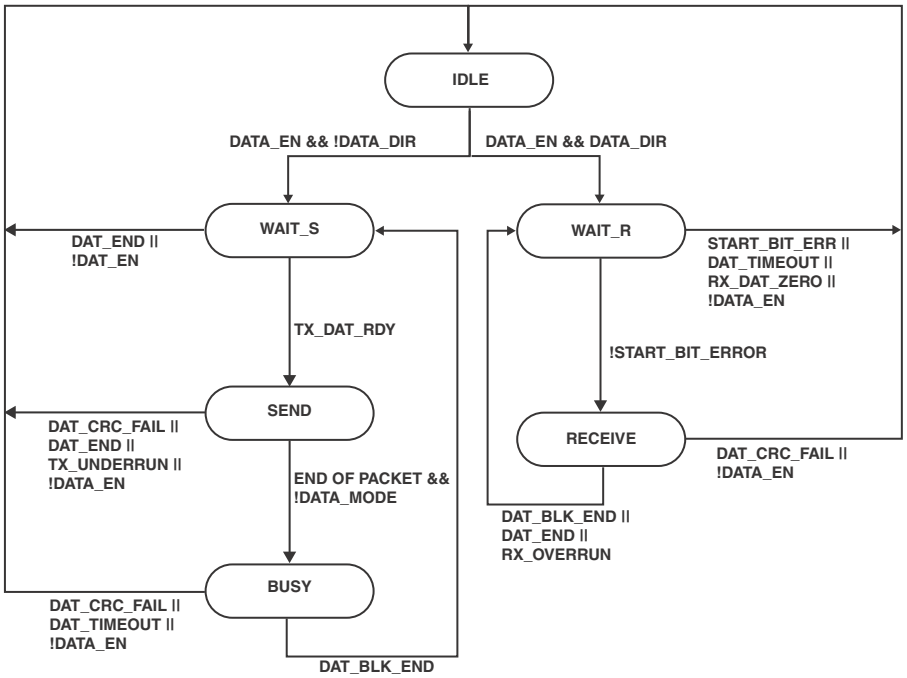


Figure 21-5. RSI Data Path State Machine

Table 21-8. RSI_STATUS Flags

RSI_STATUS Flag	Description	States Flag Set in
TX_ACT	Data transmit in progress	WAIT_S
RX_ACT	Data receive in progress	WAIT_R
DAT_BLK_END	Data block sent successfully and CRC pass token received	BUSY (block transfer mode only)
	Data block received correctly and CRC passed	RECEIVE (block transfer only)
DAT_CRC_FAIL	Data block CRC failed on transmit	SEND if transmitted data is not a multiple of DATA_BLK_LGTH. BUSY if CRC token indicates failure.
	Data block CRC failed on receive	RECEIVE
DAT_TIMEOUT	Transmit timeout occurred before card deasserted busy signal on RSI_DATA0	BUSY
	Receive timeout error occurred before start bit of data detected	WAIT_R
DAT_END	All data sent	SEND
	All data received	RECEIVE
START_BIT_ERR	Start bit not detected on all RSI_DATAx signals	WAIT_R
TX_FIFO_STAT	Transmit FIFO is half empty	SEND
TX_FIFO_FULL	Transmit FIFO is full	SEND
TX_FIFO_EMPTY	Transmit FIFO is empty	SEND
TX_UNDERRUN	Transmit FIFO under run error	SEND
TX_DAT_RDY	Valid data available in the transmit FIFO	SEND
RX_FIFO_STAT	Receive FIFO is half empty	RECEIVE
RX_FIFO_FULL	Receive FIFO is full	RECEIVE
RX_FIFO_EMPTY	Receive FIFO is empty	RECEIVE

Table 21-8. RSI_STATUS Flags (Cont'd)

RSI_STATUS Flag	Description	States Flag Set in
RX_ OVERRUN	Receive FIFO over run error	RECEIVE
RX_FIFO_RDY	Valid data is available in the receive FIFO	RECEIVE

RSI Data Transmit Path

The transmit path consists of the WAIT_S, SEND, and BUSY states. Before enabling the data path state machine via RSI_DATA_CONTROL, both RSI_DATA_LGTH and RSI_DATA_TIMER must be configured. Upon leaving the IDLE state and entering the WAIT_S state, the RSI sets the TX_ACTIVE flag and copies RSI_DATA_LGTH into RSI_DATA_CNT. The behavior of the SEND state is influenced by the transfer mode.

If the RSI is configured for stream-transfer mode, the RSI sends data to the card until RSI_DATA_CNT expires, at which time the DATA_END flag is set and the state machine returns to the IDLE state. Additionally, the transition of RSI_DATA_CNT to zero will result in the command path state machine being activated if currently in the PEND state. If at any point during the stream transfer the transmit FIFO becomes empty and data is not available in the FIFO by the time the next transfer is due to take place, the TX_UNDERRUN flag is set before returning to the IDLE state.

In block transfer mode, DATA_BLK_LGTH bytes are transmitted as specified during the write to RSI_DATA_CONTROL, each byte transferred also results in the decrementing of RSI_DATA_CNT. Upon completion of the block transfer, the RSI appends an internally generated 16-bit CRC code and an end bit before waiting for the card response on the RSI_DATA0 line to indicate whether the data was received correctly by the card. If the CRC response token sent by the card indicates that the data was received correctly, the DAT_BLK_END flag is set before moving onto the BUSY state; otherwise, the DAT_CRC_FAIL flag is set before returning to the IDLE state. The decrementing of RSI_DATA_CNT to zero results in the DAT_END flag being set. If

the total number of bytes transmitted for the current block results in the RSI_DATA_CNT decrementing to zero and the number of bytes transferred is not equal to DATA_BLK_LGTH, the transmission stops and the DAT_CRC_FAIL flag is set and the data path returns to the IDLE state. If at any point during the block transfer the transmit FIFO becomes empty and data is not available in the FIFO by the time the next transfer is due to take place, the TX_UNDERRUN flag is set before returning to the IDLE state.

Upon entry to the BUSY state, the RSI starts internally decrementing the timeout value as specified by RSI_DATA_TIMER. While in the BUSY state, the RSI continually samples the RSI_DATA0 signal, which at this time may be driven low by the card to indicate that the card is busy. Upon a logic high state being detected, indicating that the card is no longer busy, the state machine returns to the WAIT_S state where it then either returns to IDLE if all data has been sent or it moves back to the SEND state to start another block transfer. If the RSI timeout counter expires before the RSI_DATA0 signal is detected high, the RSI sets the DAT_TIMEOUT flag and returns to the IDLE state.

RSI Data Receive Path

The receive path consists of the WAIT_R and the RECEIVE states. Before enabling the data path state machine via RSI_DATA_CONTROL, RSI_DATA_LGTH and RSI_DATA_TIMER must be configured. Upon leaving the IDLE state and entering the WAIT_R state, the RSI sets the RX_ACTIVE flag and copies RSI_DATA_LGTH into RSI_DATA_CNT. The behavior of the RECEIVE state is influenced by the transfer mode.

Once the receive path has entered the WAIT_R state after being enabled for a receive transaction, the RSI starts decrementing the timeout value supplied via RSI_DATA_TIMER. If the RSI is configured for a 1-bit data bus, the DAT_TIMEOUT flag is set if a start bit is not detected on the RSI_DATA0 signal before the timeout counter reaches zero and the state machine returns to the IDLE state. If the RSI is configured for 4-bit bus mode and the start bit is not detected on all four RSI_DATAx signals prior to the

timeout counter expiring, the DAT_TIMEOUT flag is set and the state machine returns to the IDLE state. If the RSI is configured for 4-bit bus mode and a start bit is detected on some of the RSI_DATAx signals but not all of them on the same sampled clock cycle, the START_BIT_ERR flag is set and the state machine returns to the IDLE state. Upon correct detection of the start bit, the state machine moves into the RECEIVE state.

The behavior of the RECEIVE state differs for stream and block transfers. For stream transfers, received data is packed into bytes and written to the data FIFO. Data is continually received and written to the data FIFO until RSI_DATA_CNT decrements to zero. When the counter reaches zero, the remaining data in the shift register is written into the FIFO, the DAT_END flag is set and the state machine transitions to the WAIT_R state where upon detecting the emptying of the receive FIFO the RX_DAT_ZERO flag is set and the state transitions to the IDLE state. If at any time during the receive state the data FIFO becomes full and data has not been read from the FIFO prior to the next byte being written to the FIFO, the RX_OVERRRUN flag is set and the state transitions to the WAIT_R state then onto the IDLE state.

In block transfer mode, the received data is packed into bytes and written to the data FIFO. Once DATA_BLK_LGTH bytes have been received, the RSI reads the 16-bit CRC check bits. If the received CRC matches the internally calculated CRC, the DAT_BLK_END flag is set and the state transitions to the WAIT_R state. If the RSI_DATA_CNT counter expires in alignment with the end of a block as specified via DATA_BLK_LGTH, the DAT_END flag is set in addition to the DAT_BLK_END flag and the state then transitions to the WAIT_R state where upon detecting the emptying of the receive FIFO the RX_DAT_ZERO flag is set and the state transitions to the IDLE state. If RSI_DATA_CNT expires prior to the end of a block as specified via DATA_BLK_LGTH being received, the DAT_CRC_FAIL flag is set and the state transitions to the IDLE state.

RSI Data Path CRC

The data CRC generator of the RSI calculates the 16-bit CRC checksum for all bits sent or received for a given block transaction. The data path CRC generator is not enabled for stream-based data transfers. For a 1-bit bus configuration, the 16-bit CRC is calculated for all data sent on the RSI_DATA0 signal. For a 4-bit-wide data bus, the 16-bit CRC is calculated separately for each individual RSI_DATAx signal. The data path CRC checksum is a 16-bit value calculates as follows:

$$CRC[15:0] = Remainder \frac{x_{16} \cdot M(x)}{G(x)}$$

with:

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

where:

$$M(x) = x_{((8 \times DTX\_BLK\_LGTH) - 1)} \cdot (first\ data\ bit) + \ldots + x_0 \cdot (last\ data\ bit)$$

RSI Data FIFO

The data FIFO is a 32-bit wide, 16-word deep data buffer with transmit and receive logic. The FIFO is configuration depends on the state of the TX_ACT and RX_ACT flags. If TX_ACT is set, the FIFO operates as a transmit FIFO supplying data to the RSI for transfer to the card. The RX_ACT flag configures the FIFO as a receive FIFO whereby the RSI writes the data received from the card. If neither the TX_ACT nor RX_ACT flags are set, the FIFO is disabled.

When the transmit FIFO is disabled, all the transmit status flags are deasserted and the transmit read and write pointers are reset. The RSI asserts

the TX_ACT flag upon starting a data transmit operation. During the data transfer, the transmit logic maintains a number of transmit FIFO status flags as shown in Table 21-9.

Table 21-9. RSI Transmit FIFO Status Flags

RSI_STATUS Flag	Description
TX_FIFO_STAT	Transmit FIFO is half empty
TX_FIFO_FULL	Transmit FIFO is full
TX_FIFO_EMPTY	Transmit FIFO is empty
TX_UNDERRUN	Transmit FIFO under run error
TX_DAT_RDY	Valid data available in the transmit FIFO

When the receive FIFO is disabled, all receive status flags are deasserted and the receive read and write pointers are reset. The RSI asserts the RX_ACT flag upon starting a data read transaction. During the data transfer, the receive logic maintains the receive FIFO status flags shown in Table 21-10.

Table 21-10. RSI Receive FIFO Status Flags

RSI_STATUS Flag	Description
RX_FIFO_STAT	Receive FIFO is half empty
RX_FIFO_FULL	Receive FIFO is full
RX_FIFO_EMPTY	Receive FIFO is empty
RX_OVERRUN	Receive FIFO under run error
RX_DAT_RDY	Valid data available in the receive FIFO

SDIO Interrupt and Read Wait Support

In order for the RSI to accommodate SDIO functionality, two additional features are implemented:

- Hardware interrupt support over the RSI_DATA1 pin
- Read wait request over the RSI_DATA2 pin

SDIO devices may have multiple interrupt sources within the SDIO device that are mapped to a single interrupt line. The interrupt is level-sensitive, allowing multiple functions to generate an interrupt simultaneously. Thus, the interrupt request will continually be asserted until all sources generating an interrupt are determined and cleared by the RSI. The sources of the interrupts are found by interrogating the SDIO device and are cleared via function unique operations.

The SDIO device sends an interrupt request to the RSI by asserting the RSI_DATA1 signal low. The interrupt status is indicated by the SDIO_INT_DET bit of the RSI_ESTAT register. The status can be configured to generate an interrupt on the processor via the SDIO_INT_DET_MASK bit of the RSI_EMASK register.

When the RSI is configured for a 1-bit bus width, the interrupt may be generated by the SDIO with no timing constraints as the RSI_DATA1 signal acts as a dedicated IRQ signal. The RSI should be configured via RSI_CONFIG such that pull-up are enabled on all RSI_DATAx signals. Upon the RSI sampling RSI_DATA1 low, the RSI asserts the SDIO_INT_DET flag; this flag is asserted until the RSI_DATA1 signal is sampled high again.

When the RSI is configured for 4-bit bus widths, the RSI_DATA1 signal is shared between the IRQ signal and the RSI_DATA1 signal. In this configuration, the interrupt may only be recognized by the RSI within a specific interrupt period.

Programming Model

This section contains the following procedures:

- "Card Identification"
- "Single Block Write Operations" on page 21-34
- "Single Block Read Operation" on page 21-38
- "Multiple Block Write Operation" on page 21-42
- "Multiple Block Read Operation" on page 21-47

Card Identification

Before data transfers can take place between the RSI and the SD/MMC/SDIO device, the device type must first be identified. During this phase of the card identification procedure, the RSI_CLK frequency must be no greater than 400 kHz.

SD Card Identification Procedure

The SD card identification procedure is as follows:

1. Issue the IDLE command to the card via the RSI_COMMAND register
2. Issue the SEND_IF command to the card via the RSI_COMMAND register supplying the host supply voltage and a check pattern via the RSI_ARGUMENT register. The command expects an R7 response type. If a valid response with a compatible voltage range and matching check pattern is received, the card is likely an SD version 2.00 or later complaint card. If a response is received with an incompatible voltage range, the card cannot be used. If no response is received at all (as indicated by the CMD_RESP_TIMEOUT field of the RSI_STATUS register), continue from step 5.

3. Issue the RSI_SEND_OP_COND command via the RSI_COMMAND register, supplying the voltage window supported and whether the host supports high capacity cards via the RSI_ARGUMENT register. The RSI expects an R3 response to this command, at which time the card can be rejected if the voltage range is not compatible. If the card returns a response indicating that it is busy, resend the RSI_SEND_OP_CMD until the card indicates that it is ready. If the host does not support the high capacity mode (as indicated by setting the HCS bit of the argument to 0), a high capacity card will never clear the busy status bit. The card should be identified within 1 second. If in that timeframe the card is still busy or no valid responses have been received, the card should be rejected.

4. If the host supports high-capacity cards, verify whether the response in RSI_RESPONSE0 indicates the card capacity status (CCS) bit is set. If CCS is set, an SD Version 2.00 or later high-capacity SD memory card is present; proceed to step 6. If the CCS bit is cleared, the card is an SD Version 2.00 or later standard-capacity memory card; proceed to step 6.

5. Issue the RSI_SEND_OP_COND command via the RSI_COMMAND register, supplying the voltage window supported and with the high-capacity support (HCS) bit set to 0 via the RSI_ARGUMENT register. The RSI expects an R3 response to this command, at which time the card can be rejected if the voltage range is not compatible. If the card returns a response indicating that it is busy, resend the RSI_SEND_OP_CMD until the card indicates that it is ready. The card should be identified within 1 second. If in that timeframe the card is still busy or no valid responses have been received, the card should be rejected. Once the response indicates that the card is ready, the card type has been identified as an SD Version 1.x standard-capacity memory card.

6. Issue the ALL_SEND_CID command to which an R2 response type is expected. This will result in the card sending the 128-bit card identification (CID) register and transitioning from ready to identification mode.

7. Issue the SEND_RELATIVE_ADDR command to which an R6 response type is expected. This will result in the card issuing a new relative address which must be used in order to select the card in the future for data transfers. The card will then move into standby mode completing the identification procedure.

MMC Identification Procedure

The MMC identification procedure is as follows:

1. Issue the IDLE command to the card via the RSI_COMMAND register.

2. Issue the SEND_OP_COND command to the card via the RSI_COMMAND register, supplying the operating voltage window that the host is compatible with and the access mode that the host supports (byte or sector) via the RSI_ARGUMENT register. The RSI expects an R3 type response. This allows the host to reject the card if it is not compatible with the supply voltage or if the access mode is not supported by the host software. If the card returns an indication that it is busy, repeat this step until the card is either rejected or not busy.

3. Issue the ALL_SEND_CID command via the RSI_COMMAND register. The RSI expects an R2 response to this command. This will result in the card sending the 128-bit card identification (CID) register and transitioning from ready to identification mode.

4. Issue the SET_RELATIVE_ADDR command, providing a 16-bit relative card address (RCA) via the RSI_ARGUMENT register that will get assigned to the card. An R1 response type is expected for this command. This will result in the card being assigned with the provided

RCA, which must be used in order to select the card in the future for data transfers. The card will then move into standby mode, completing the identification procedure.

Single Block Write Operations

Block write operations typically consist of 512 bytes of data per block. If the card is found to support other block lengths or the default block length as specified in the CID register is not 512, the block length of the RSI must be configured accordingly. The block length of the card and the block length of the RSI must be configured for the same block size at all times. The block length of the RSI is configured via the DATA_BLK_LGTH field of the RSI_DATA_CONTROL register.

It is important to pay attention as to when the data path state machine is enabled and when data is written to the FIFO for transfer to the card. Write transactions require that data be written after the response has completed for the WRITE_BLOCK command. If the data path state machine is enabled prior to sending the WRITE_BLOCK command, data must not be written to the transmit FIFO via the DMA or core until after the response has been received as indicated by the CMD_RESP_END flag. Failure to adhere to this procedure can result in data being written to the card in violation to the block write timing parameters, resulting in a data CRC failure.

Using Core

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 512 bytes for a single block.
6. Write the appropriate timeout value for a write operation to the RSI_DATA_TIMER register.
7. Write the destination start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512 byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or sector-addressable MMC card.
8. Write the WRITE_BLOCK command to the RSI_COMMAND register, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

9. Wait for the CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register.
10. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN should also be set to enable the data path state machine. All other fields of the RSI_DATA_CONTROL register should be zero.
11. Write data to the RSI_FIFO register until the FIFO becomes full as indicated by the TX_FIFO_FULL flag of the RSI_STATUS register. Continue to write data to the FIFO as long as the FIFO is not full or write data in blocks of eight 32-bit words if polling on the TX_FIFO_STAT bit indicating the transmit FIFO is half empty. Continue until all 128 32-bit words (512 bytes) have been transferred.
12. Wait for the card to respond with the CRC token by waiting for the DAT_BLK_END flag to be set. DAT_END will also be set at this time if the RSI_DATA_LGTH register was set to 512 bytes in step 5.
13. Clear the DAT_BLK_END and DAT_END flags via the RSI_STATUSCL register.

Using DMA

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.

3. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Configure the DMA channel assigned to the RSI controller. Write DMAx_START_ADDR with the address of the first byte of data to be written to the card. The DMAx_X_COUNT register should be set to 128, and the DMAx_X_MODIFY register to 4. The DMAx_CONFIG register should be set for DMA enable (a word size of 32-bits).
6. Once the DMA channel has been configured and enabled, write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 512 bytes for a single block.
7. Write the appropriate timeout value for a write operation to the RSI_DATA_TIMER register.
8. Write the destination start address to the RSI_ARGUMENT register. The address supplied must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or sector-addressable MMC card.
9. Write the WRITE_BLOCK command to RSI_COMMAND, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.
10. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
11. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN and DATA_DMA_EN should also be set to

enable the data path state machine and to allow the DMA controller to access the transmit FIFO. All other fields of the RSI_DATA_CONTROL register should be zero.

12. Wait for the card to respond with the CRC token by waiting for the DAT_BLK_END flag to be set. DAT_END will also be set at this point if the RSI_DATA_LGTH register was set to 512 bytes in step 5.

13. Clear the DAT_BLK_END and DAT_END flags via the RSI_STATUSCL register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register, if applicable.

Single Block Read Operation

Block read operations typically consist of 512 bytes of data per block. If the card is found to support other block lengths or the default block length as specified in the CID register is not 512, the block length of the RSI must be configured accordingly. The block length of the card and the block length of the RSI must be configured for the same block size at all times. The block length of the RSI is configured via the DATA_BLK_LGTH field of the RSI_DATA_CONTROL register.

It is important to pay attention as to when the data path state machine is enabled and when data is read from the receive FIFO for data transfers from the card to the RSI. Read transactions can occur on the RSI_DATAx signals prior to the response of the command being received. It is therefore advisable to enable the data path state machine, and DMA controller if being used, either:

- Prior to issuing a command that involves a data read packet
- Immediately after the command has been issued but prior to pending on the CMD_RESP_END flag

If the core is being used to read the receive FIFO, it is advised not to pend on the CMD_RESP_END flag. It is possible for data to be driven on the RSI_DATAx signals two RSI_CLK cycles after the end bit of the command. At minimum, an additional 48 RSI_CLK cycles will pass before the response is received, during which time the receive buffer may potentially have received 24 bytes of data on a 4-bit bus and will be approaching the half full state. Software should ensure that the receive buffer does not become full prior to data being read from the receive FIFO.

Using Core

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 512 bytes for a single block.
6. Write the appropriate timeout value for a read operation to the RSI_DATA_TIMER register.

7. Write the destination start address to the RSI_ARGUMENT register. The address supplied must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or sector-addressable MMC card.

8. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN and DATA_DIR should also be set to enable the data path state machine and indicate the transfer direction is from card to controller. All other fields of the RSI_DATA_CONTROL register should be zero.

9. Write the READ_SINGLE_BLOCK command to the RSI_COMMAND register, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

10. In order to meet some timing restrictions related to block read operations, it is advisable to not wait for the CMD_RESP_END indication within the RSI_STATUS register but instead move immediately on the next step. This is due to the card being able to send data before a response can completed on the RSI_CMD signal, moving immediately onto step 11 will ensure a receive FIFO overflow does not occur.

11. Poll the RX_FIFO_RDY bit or the RX_DAT_ZERO bit of RSI_STATUS indicating the receive FIFO has data available, or the receive FIFO is empty. As long as the receive FIFO is not empty, read data from the RSI_FIFO register until all 512 bytes have been read

12. Once all bytes have been read, wait for the DAT_BLK_END flag to indicate that the data was received correctly and passed the CRC check. The DAT_END flag may also be set, depending on the value written to RSI_DATA_LGTH.

13. Clear the DAT_BLK_END and DAT_END flags via the RSI_STATUSCL register.

Using DMA

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Configure the DMA channel assigned to the RSI controller. Write DMAx_START_ADDR with the address of the first byte of where the received data is to be stored. The DMAx_X_COUNT register should be set to 128 and the DMAx_X_MODIFY register to 4. The DMAx_CONFIG register should be set for DMA enable (a word size of 32-bits and direction set to memory write).
6. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 512 bytes for a single block.
7. Write the appropriate timeout value for a read operation to the RSI_DATA_TIMER register.
8. Write the source start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512-byte boundary if mis-aligned accesses are not enabled and the card is not a high-capacity SD card or sector-addressable MMC card.

9. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN, DATA_DIR, and DATA_DMA_EN should also be set to enable the data path state machine, set the transfer direction from card to controller and allow the DMA controller access to the receive FIFO. All other fields of the RSI_DATA_CONTROL register should be zero.

10. Write the READ_SINGLE_BLOCK command to the RSI_COMMAND register, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

11. Unlike core accesses, it is safe to poll on CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register. The DMA controller enabled in step 5 will ensure any data sent to the receive FIFO prior to the CMD_RESP_END flag being set is received correctly.

12. Wait for the DAT_BLK_END flag to indicate that the data was received correctly and passed the CRC check. The DAT_END flag may also be set, depending on the value written to RSI_DATA_LGTH.

13. Clear the DAT_BLK_END and DAT_END flags via the RSI_STATUSCL register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register, if applicable.

Multiple Block Write Operation

Block write operations typically consist of 512 bytes of data per block. If the card is found to support other block lengths or the default block length as specified in the CID register is not 512, the block length of the RSI must be configured accordingly. The block length of the card and the block length of the RSI must be configured for the same block size at all times. The block length of the RSI is configured via the DATA_BLK_LGTH field of the RSI_DATA_CTL register.

Using Core

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. For example, write 4096 to write eight blocks of 512 bytes.
6. Write the appropriate timeout value for a write operation to the RSI_DATA_TIMER register.
7. Write the destination start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or a sector-addressable MMC card.
8. Write the WRITE_MULTIPLE_BLOCK command to RSI_COMMAND, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

9. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.

10. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN should also be set to enable the data path state machine. All other fields of the RSI_DATA_CONTROL register should be zero.

11. Write data to the RSI_FIFO register until the FIFO becomes full as indicated by the TX_FIFO_FULL flag of the RSI_STATUS register. Continue to write data to the FIFO as long as the FIFO is not full or write data in blocks of eight 32-bit words if polling on the TX_FIFO_STAT bit indicating the transmit FIFO is half empty. Continue until all 128 32-bit words (512 bytes) have been transferred.

12. Wait for the card to respond with the CRC token by waiting for the DAT_BLK_END flag to be set.

13. Clear the DAT_BLK_END flag.

14. Repeat steps 11 to 13 for the number of blocks to be transferred or until DAT_END flag is set.

15. Write the RSI_COMMAND register with the STOP_TRANSMISSION command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

16. Clear the DAT_END and CMD_RESP_END flags via the RSI_STATUSCL register.

Using DMA

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Configure the DMA channel assigned to the RSI controller. Write DMAx_START_ADDR with the address of the first byte of data to be written to the card. The DMAx_X_COUNT register should be set to the overall number of 32-bit words to be written; for example, write 1024 to transfer 4096 bytes. The DMAx_X_MODIFY register should be set to 4. The DMAx_CONFIG register should be set for DMA enable and a word size of 32-bits.
6. Once the DMA channel has been configured and enabled, write the number of bytes to be transferred to the RSI_DATA_LGTH register. For example, write 4096 to write eight blocks of 512 bytes.
7. Write the appropriate timeout value for a write operation to the RSI_DATA_TIMER register.

8. Write the destination start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or sector addressable MMC card.
9. Write the WRITE_MULTIPLE_BLOCK command to the RSI_COMMAND, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.
10. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
11. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN and DATA_DMA_EN should also be set to enable the data path state machine and to allow the DMA controller to access the transmit FIFO. All other fields of the RSI_DATA_CONTROL register should be zero.
12. Poll for the DAT_END flag or alternatively poll for each instance of the DAT_BLK_END flag that will be set on successful completion of each block transfer. For a 4096 byte transfer, DAT_BLK_END will be set eight times and should be cleared after it is detected via the RSI_STATUSCL register.
13. Write the RSI_COMMAND register with the STOP_TRANSMISSION command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.
14. Clear the DAT_END and CMD_RESP_END flag via the RSI_STATUSCL register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register, if applicable.

Multiple Block Read Operation

Block read operations typically consist of 512 bytes of data per block. If the card is found to support other block lengths or the default block length as specified in the CID register is not 512, the block length of the RSI must be configured accordingly. The block length of the card and the block length of the RSI must be configured for the same block size at all times. The block length of the RSI is configured via the DATA_BLK_LGTH field of the RSI_DATA_CONTROL register.

Using Core

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and there are no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 4096 for a transfer of eight 512 byte blocks.
6. Write the appropriate timeout value for a read operation to the RSI_DATA_TIMER register.

7. Write the destination start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or a sector-addressable MMC card.

8. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN and DATA_DIR should also be set to enable the data path state machine and indicate the transfer direction is from card to controller. All other fields of the RSI_DATA_CONTROL register should be zero.

9. Write the READ_MULTIPLE_BLOCK command to the RSI_COMMAND register, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

10. In order to meet some timing restrictions related to block read operations, it is advisable to not wait for the CMD_RESP_END indication within the RSI_STATUS register but instead move immediately on the next step. This is due to the card being able to send data before a response can completed on the RSI_CMD signal, moving immediately onto step 11 will ensure a receive FIFO overflow does not occur.

11. Poll the RX_FIFO_RDY bit or the RX_DAT_ZERO bit of RSI_STATUS indicating the receive FIFO has data available, or the receive FIFO is empty. As long as the receive FIFO is not empty, read data from the RSI_FIFO register until 512 bytes have been read.

12. Once the block has been read, wait for the DAT_BLK_END flag to indicate that the data was received correctly and passed the CRC check.

13. Clear the DAT_BLK_END flag via RSI_STATUSCL.

14. Repeat steps 11 to 13 until the required number of blocks have been read or until the DAT_END flag has been set.
15. Write the RSI_COMMAND register with the STOP_TRANSMISSION command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.
16. Clear the DAT_END and CMD_RESP_END flags via the RSI_STATUSCL register.

Using DMA

The procedure is as follows:

1. Write the RSI_ARGUMENT register with the cards RCA. The 16-bit RCA should be written to the upper 16-bits of the RSI_ARGUMENT register.
2. Write the RSI_COMMAND register with the SELECT/DESELECT_CARD command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1b.
3. Wait for the CMD_RESP_END indication within the RSI_STATUS register, and clear the status bit once detected via the RSI_STATUSCL register.
4. Ensure that the device is not busy and that no errors occurred by verifying the response contained in RSI_RESPONSE0.
5. Configure the DMA channel assigned to the RSI controller. Write DMAx_START_ADDR with the address of the first byte of where the received data is to be stored. The DMAx_X_COUNT register should be set to the number of 32-bit words to be read, which would be 1024

for a 4096 byte read transfer. The DMAx_X_MODIFY register should be set to 4. The DMAx_CONFIG register should be set for DMA enable (a word size of 32-bits and direction set to memory write).

6. Write the number of bytes to be transferred to the RSI_DATA_LGTH register. This will be 4096 for eight blocks of 512 bytes.
7. Write the appropriate timeout value for a read operation to the RSI_DATA_TIMER register.
8. Write the source start address to the RSI_ARGUMENT register. The supplied address must be aligned to a 512-byte boundary if misaligned accesses are not enabled and the card is not a high-capacity SD card or a sector-addressable MMC card.
9. Enable the data path state machine by writing to the RSI_DATA_CONTROL register with DATA_BLK_LGTH set to 9 for a 512-byte block. DATA_EN, DATA_DIR, and DATA_DMA_EN should also be set to enable the data path state machine. Set the transfer direction from card to controller and allow the DMA controller access to the receive FIFO. All other fields of the RSI_DATA_CONTROL register should be zero.
10. Write the READ_MULTIPLE_BLOCK command to the RSI_COMMAND register, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.
11. Unlike core accesses, it is safe to poll on CMD_RESP_END indication within the RSI_STATUS register and clear the status bit once detected via the RSI_STATUSCL register. The DMA controller, enabled in step 5 will ensure any data sent to the receive FIFO prior to the CMD_RESP_END flag being set is received correctly.

12. Poll for the DAT_END flag or alternatively poll for each instance of the DAT_BLK_END flag that will be set on successful completion of each block transfer. For a 4096-byte transfer, DAT_BLK_END will be set eight times and should be cleared after it is detected via the RSI_STATUSCL register.

13. Write the RSI_COMMAND register with the STOP_TRANSMISSION command, configuring the command path state machine to expect a short response by setting CMD_RESP and clearing CMD_L_RESP. The response type is R1.

14. Clear the DAT_END and CMD_RESP_END flags via the RSI_STATUSCL register. Also clear the DMA_DONE bit of the DMAx_IRQ_STATUS register, if applicable.

RSI Registers

Table 21-11 summarizes the RSI registers together with their function, memory-mapped address, type, and access.

Table 21-11. RSI Module Registers

Register Name	Function	Address	Type	Access
RSI_PWR_CONTROL	RSI power control register on page 21-53	0xFFC03800	R/W	16-bit
RSI_CLK_CONTROL	RSI clock control register on page 21-54	0xFFC03804	R/W	16-bit
RSI_ARGUMENT	RSI argument register on page 21-56	0xFFC03808	R/W	32-bit
RSI_COMMAND	RSI command register on page 21-56	0xFFC0380C	R/W	16-bit
RSI_RESP_CMD	RSI response command register on page 21-58	0xFFC03810	R	16-bit

Table 21-11. RSI Module Registers (Cont'd)

Register Name	Function	Address	Type	Access
RSI_RESPONSE0 RSI_RESPONSE1 RSI_RESPONSE2 RSI_RESPONSE3	RSI response registers on page 21-59	0xFFC03804 0xFFC03808 0xFFC0381C 0xFFC03820	R	32-bit
RSI_DATA_TIMER	RSI data timer register on page 21-60	0xFFC03824	R/W	32-bit
RSI_DATA_LGTH	RSI data length register on page 21-61	0xFFC03828	R/W	16-bit
RSI_DATA_CONTROL	RSI data control register on page 21-61	0xFFC0382C	R/W	16-bit
RSI_DATA_CNT	RSI data counter register on page 21-63	0xFFC03830	R	16-bit
RSI_STATUS	RSI status register on page 21-64	0xFFC03834	R	32-bit
RSI_STATUSCL	RSI status clear register on page 21-67	0xFFC03838	W1A	16-bit
RSI_MASK0 RSI_MASK1	RSI IRQ0 mask registers on page 21-69	0xFFC0383C 0xFFC03840	R/W	32-bit
RSI_FIFO_CNT	RSI FIFO counter register on page 21-72	0xFFC03848	R	16-bit
RSI_CEATA_CONTROL	RSI CE-ATA control register on page 21-73	0xFFC0384C	R/W1A/W	16-bit
RSI_FIFO	RSI data FIFO register on page 21-74	0xFFC03880	R/W	32-bit
RSI_ESTAT	RSI exception status register on page 21-74	0xFFC038C0	R/W1C	16-bit
RSI_EMASK	RSI exception mask register on page 21-76	0xFFC038C4	R/W	16-bit
RSI_CONFIG	RSI configuration register on page 21-77	0xFFC038C8	R/W	16-bit

Table 21-11. RSI Module Registers (Cont'd)

Register Name	Function	Address	Type	Access
RSI_RD_WAIT_EN	RSI read wait enable register on page 21-79	0xFFC038CC	R/W1A/W	16-bit
RSI_PID0 RSI_PID1 RSI_PID2 RSI_PID3	RSI peripheral identification registers on page 21-80	0xFFC038D0 0xFFC038D4 0xFFC038D8 0xFFC038DC	R	16-bit

RSI Power Control Register (RSI_PWR_CONTROL)

The RSI_PWR_CONTROL register contains bits that control the power to the RSI module as well as the open-drain configuration for the RSI_CMD signal. The PWR_ON field must be set to "11" in order for the RSI to be enabled. The RSI_CMD_OD bit, when set, results in the RSI driving the RSI_CMD signal in open-drain mode. The default mode of operation is push-pull. After a data write, data cannot be written to this register for a five SCLK cycles.

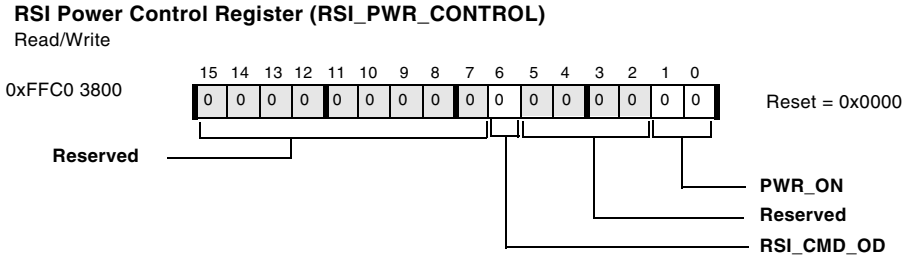


Figure 21-6. RSI Power Control Register

Table 21-12. RSI_PWR_CONTROL Register

Bit	Name	Function	Type	Default
1:0	PWR_ON	Power on 00 = RSI disabled 01 = Reserved 10 = Reserved 11 = RSI enabled	RO	0
5:2	Reserved	Reserved	RO	0
6	RSI_CMD_OD	RSI_CMD open drain 0 = Disabled (push-pull) 1 = Enabled	RO	0
15:7	Reserved	Reserved	RO	0

RSI Clock Control Register (RSI_CLK_CONTROL)

The RSI_CLK_CONTROL register provides control functionality for the RSI clock. RSI_CLK can be derived directly from the SCLK signal by enabling CLKDIV_BYPASS; otherwise, RSI_CLK frequency is determined from the current SCLK frequency and the CLKDIV field:

$$RSI\_CLK = \frac{SCLK}{2 \times (CLKDIV + 1)}$$

In order to conserve power, the RSI clock can be disabled without disabling the entire RSI interface via the CLK_EN bit; additionally the PWR_SV_EN bit, when set, results in the RSI_CLK signal only been driven when the RSI is performing a transfer either to or from the card. In additional to clock control functionality, the data bus width of the RSI interface is also controlled from this register.

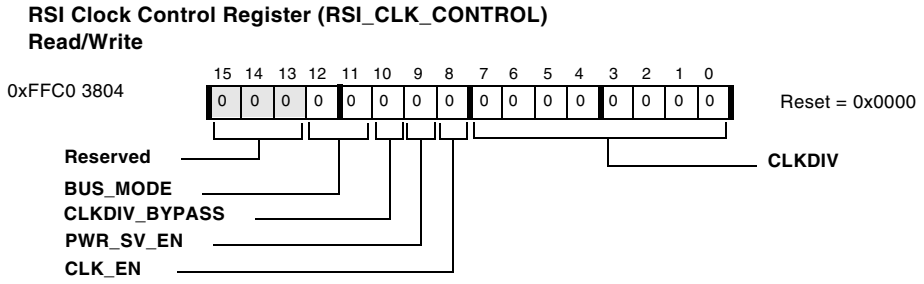


Figure 21-7. RSI Clock Control Register

Table 21-13. RSI_CLK_CONTROL Register

Bit	Name	Function	Type	Default
7:0	CLKDIV	Clock divisor 0x0 to 0xFF (see equation)	R/W	0
8	CLK_EN	RSI_CLOCK enable 0 = Disable RSI_CLK 1 = Enable RSI_CLK	R/W	0
9	PWR_SV_EN	Power save enable 0 = Disabled (RSI_CLK always driven) 1 = Enabled (RSI_CLK only enabled when bus is active)	R/W	0
10	CLKDIV_BYPASS	Bypass clock divisor 0 = Disabled (do not bypass clock divisor) 1 = Enabled (RSI_CLK derived directly from SCLK)	R/W	0
12:11	BUS_MODE	Data bus width 00 = 1-bit data bus 01 = 4-bit data bus 10 = 8-bit data bus 11 = Reserved	R/W	0
15:13	Reserved	Reserved	RO	0

RSI Argument Register (RSI_ARGUMENT)

The RSI_ARGUMENT register contains the 32-bit argument that is sent on the RSI_CMD signal as part of a command message. If a command requires an argument, the argument must first be loaded into the RSI_ARGUMENT register prior to writing and enabling the command in the RSI_COMMAND register.

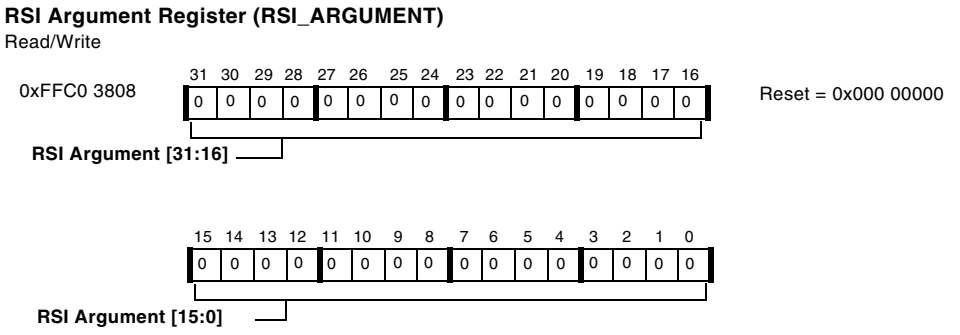


Figure 21-8. RSI Argument Register

RSI Command Register (RSI_COMMAND)

The RSI_COMMAND register is responsible for controlling the command path state machine. The CMD_IDX field contains the index of the command to be issued via the RSI as part of the command message. If the command requires a response, this is indicated via CMD_RSP_EN.

The length of the response (short or long) is controlled with CMD_LRSP_EN. The command path state machine becomes active once the CMD_EN bit is set and is disabled if this bit is cleared.

It is not required to manually clear the CMD_EN bit after the command sequence has completed. The command path state machine will automatically terminate and become IDLE once the operation has completed.

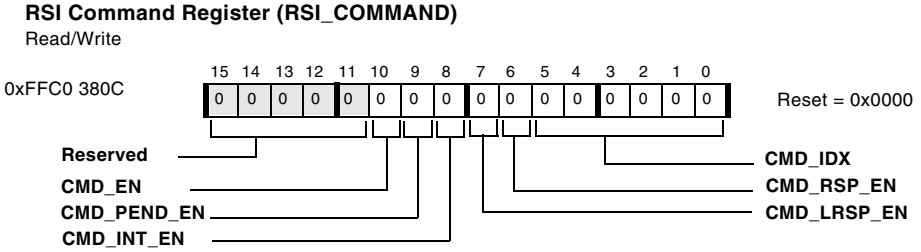


Figure 21-9. RSI Command Register

Table 21-14. RSI_COMMAND Register

Bit	Name	Function	Type	Default
5:0	CMD_IDX	Command index 0x3F - 0x00 (Command number to be issued)	R/W	0
6	CMD_RSP_EN	Wait for response 0 = Disabled 1 = Enabled	R/W	0
7	CMD_LRSP_EN	Long response enable 0 = Disabled (short response expected) 1 = Enabled (long response expected)	R/W	0
8	CMD_INT_EN	Command interrupt enable 0 = Disabled (timeout after 64 RSI_CLK cycles) 1 = Enabled (disable timeout counter and wait for interrupt)	R/W	0

Table 21-14. RSI_COMMAND Register (Cont'd)

Bit	Name	Function	Type	Default
9	CMD_PEND_EN	Pend enable 0 = Disabled (send command immediately) 1 = Enabled (wait for DAT_END before sending command)	R/W	0
10	CMD_EN	Command enable 0 = Disable command path state machine 1 = Enable command path state machine	R/W	0
15:11	Reserved	Reserved	RO	0

RSI Response Command Register (RSI_RESP_CMD)

The RSI_RESP_CMD register contains the command index field of the last response received. If the command response does not contain doe not contain a command index field (as is the case with a long response), the RESP_CMD field would typically be ignored. In this situation, it will likely contain "0x3F", which is the value of the reserved field of the response.

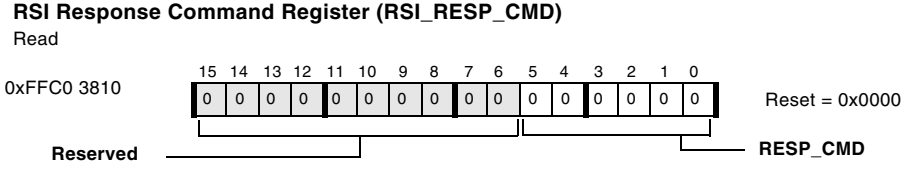


Figure 21-10. RSI Response Command Register

Table 21-15. RSI_RESP_CMD Register

Bit	Name	Function	Type	Default
5:0	RESP_CMD	Command index of last received response 0x3F - 0x00 (command index)	RO	0
15:6	Reserved	Reserved	RO	0

RSI Response Registers (RSI_RESPONSEx)

The four RSI_RESPONSEx registers (RSI_RESPONSE0, RSI_RESPONSE1, RSI_RESPONSE2, and RSI_RESPONSE3,) contain the response information received back from a card for a given command message. The received response may be 32 or 127 bits in length, depending on whether the response type is short or long. The most significant bit of the response is received first and is located in bit 31 of the RSI_RESPONSE0 register. Bit 0 of RSI_RESPONSE3 is always zero. Table 21-16 shows the RSI response registers contents for the two types of responses.

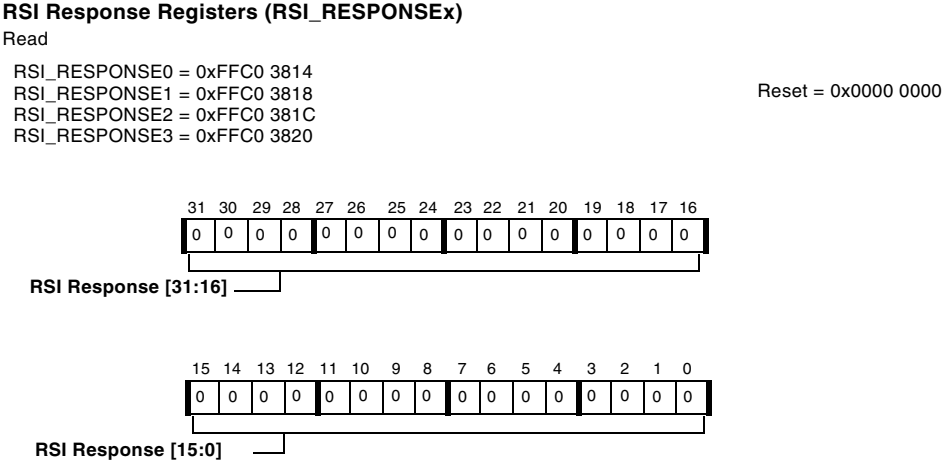


Figure 21-11. RSI Response Registers

Table 21-16. RSI Response Registers Content

Response Register	Short Response	Long Response
RSI_RESPONSE0	Response bits [31:0]	Response bits [127:96]
RSI_RESPONSE1	Not used	Response bits [95:64]
RSI_RESPONSE2	Not used	Response bits [63:32]
RSI_RESPONSE3	Not used	Response bits [31:1][1]

1 Bits 31:1 of the long response are stored in bits 30:0 of the RSI_RESPONSE3 register. Bit 31 of the RSI_RESPONSE3 register is not used and is always zero.

RSI Data Timer Register (RSI_DATA_TIMER)

The RSI_DATA_TIMER register contains a 32-bit value for the data timeout period (RSI_CLK cycles). An internal counter loads the value from this register, and starts to decrement when the data path state machine enters the WAIT_R or the BUSY states. If the timer decrements to zero while the data path state machine is still in either of these two states, the DAT_TIMEOUT flag of the RSI_STATUS register is set. The RSI_DATA_TIMER and the RSI_DATA_LGTH registers must both be written to prior to starting a data transfer via the RSI_DATA_CONTROL register.

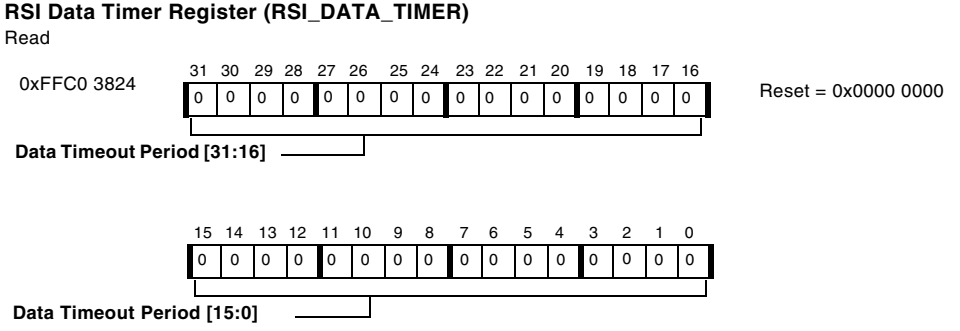


Figure 21-12. RSI Data Timer Register

RSI Data Length Register (RSI_DATA_LGTH)

The RSI_DATA_LGTH register contains a 16-bit value for the number of data bytes to be transferred before setting the DAT_END flag of the RSI_STATUS register. The value loaded to this register is copied into the RSI_DATA_CNT register when the data path state machine is enabled and starts the transfer.

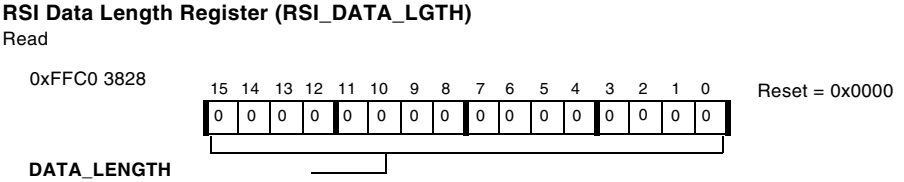


Figure 21-13. RSI Data Length Register

Table 21-17. RSI_DATA_LGTH Register

Bit	Name	Function	Type	Default
15:0	DATA_LENGTH	Number of bytes to be transferred	R/W	0

RSI Data Control Register (RSI_DATA_CONTROL)

The RSI_DATA_CONTROL register largely controls the data path state machine. The state machine becomes enabled once the DATA_EN bit is set. The direction of the transfer is determined by DATA_DIR. If the DMA channel is to be used for the data transfer, the DATA_DMA_EN bit must be set; otherwise, the RSI FIFO is only accessible via the core. For block transfers, the block length must be specified via DATA_BLK_LGTH, where the block length is $2^{DATA\_BLK\_LGTH}$. Two bits (CEATA_CCS_EN and CEATA_EN) in this register configure the behavior of the command path state machine for communication with CE-ATA devices. After a data write, data cannot be written to this register for five SCLK cycles.

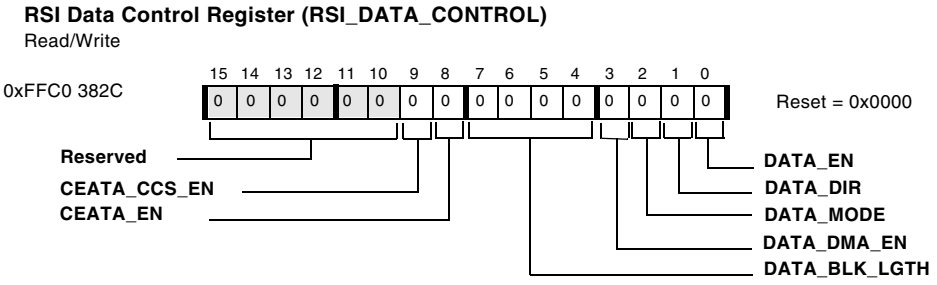


Figure 21-14. RSI Data Control Register

Table 21-18. RSI_DATA_CONTROL Register

Bit	Name	Function	Type	Default
0	DATA_EN	Data enable 0 = Disabled (disables data path state machine) 1 = Enabled (enables data path state machine)	R/W	0
1	DATA_DIR	Data transfer direction 0 = From RSI to card 1 = From card to RSI	R/W	0
2	DATA_MODE	Data transfer mode 0 = Block transfer 1 = Stream transfer	R/W	0
3	DATA_DMA_EN	Data DMA enable 0 = Disabled (use core to read/write RSI_FIFO) 1 = Enabled (use DMA controller to read/write RSI_FIFO)	R/W	0
7:4	DATA_BLK_LGTH	Data block length 0x0 - 0xC data block length (2^0 to 2^{12})	R/W	0
8	CEATA_EN	CE-ATA mode enable 0 = Disabled 1 = Enabled	R/W	0

Table 21-18. RSI_DATA_CONTROL Register (Cont'd)

Bit	Name	Function	Type	Default
9	CEATA_CCS_EN	Command completion signal enable 0 = Disabled 1 = Enabled (wait for command completion signal)	R/W	0
15:10	Reserved	Reserved	R/W	0

RSI Data Counter Register (RSI_DATA_CNT)

The RSI_DATA_CNT register is loaded from the RSI_DATA_LGTH register when the data path state machine becomes enabled and moves from the IDLE state to the WAIT_S or WAIT_R states. As the data is transferred, the counter decrements; upon decrementing to zero, the state machine then moves back to the IDLE state and the DAT_END flag of the RSI_STATUS register is set.

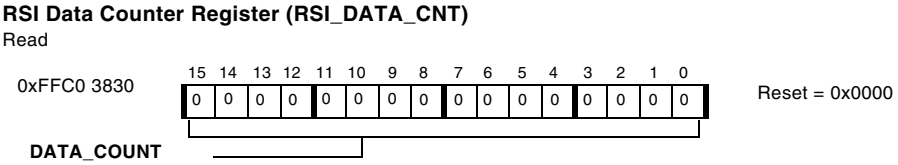


Figure 21-15. RSI Data Counter Register

Table 21-19. RSI_DATA_CNT Register

Bit	Name	Function	Type	Default
15:0	DATA_COUNT	Number of bytes still to be transferred	RO	0

RSI Status Register (RSI_STATUS)

The RSI_STATUS register contains both static and dynamic flags that indicate the status of the RSI. The static flags (bits [10:0]) remain asserted and are required to be cleared by writing to the RSI_STATUSCL register. The dynamic flags (bits [21:11]) change state, depending on the state of the underlying logic. The transmit and receive FIFO logic controls bits [21:12], which will vary depending on the state of the FIFO and whether the FIFO is currently enabled for a transmit or receive operation.

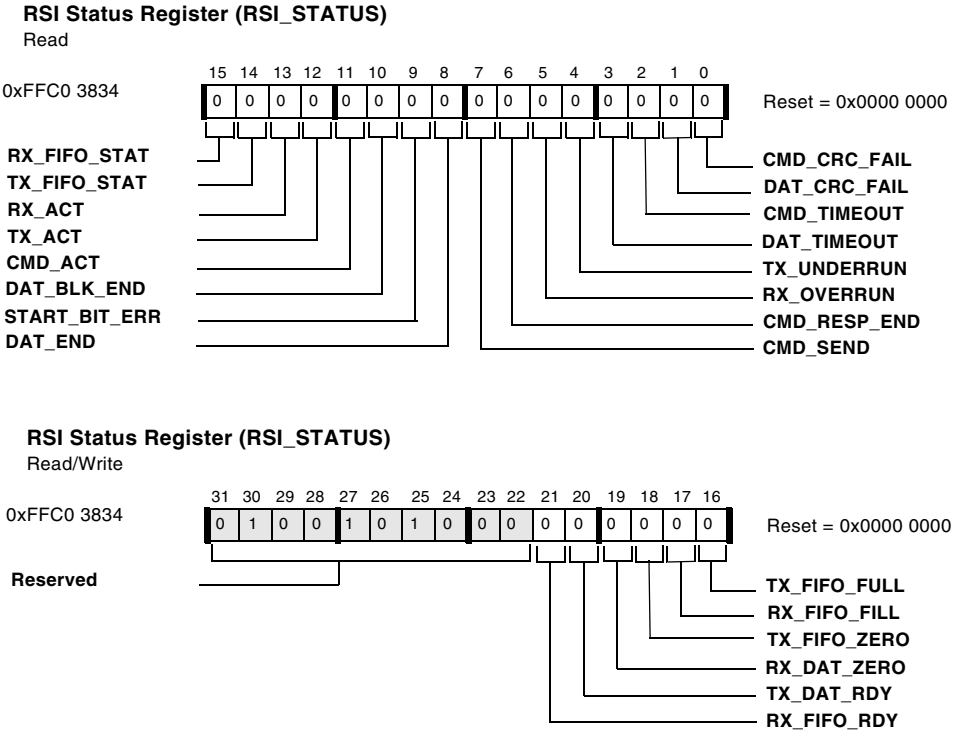


Figure 21-16. RSI Status Register

Table 21-20. RSI_STATUS Register

Bit	Name	Function	Type	Default
0	CMD_CRC_FAIL	Command response CRC fail 0 = No CRC received 1 = CRC failed on command response	RO	0
1	DAT_CRC_FAIL	Data CRC failure 0 = No CRC received on data block 1 = CRC failed on data block	RO	0
2	CMD_TIMEOUT	Command timeout 0 = Command response not timed out 1 = Command response timed out	RO	0
3	DAT_TIMEOUT	Data timeout 0 = Data not timed out 1 = Data timed out	RO	0
4	TX_UNDERRUN	Transmit FIFO underrun error 0 = No error 1 = Underrun error	RO	0
5	RX_OVERRUN	Receive FIFO overrun error 0 = No error 1 = Overrun error	RO	0
6	CMD_RESP_END	Command response received 0 = No response received 1 = Response received and CRC passed	RO	0
7	CMD_SENT	Command sent 0 = No command sent 1 = Command sent (no response required)	RO	0
8	DAT_END	End of data 0 = Not end of data 1 = End of data	RO	0

Table 21-20. RSI_STATUS Register (Cont'd)

Bit	Name	Function	Type	Default
9	START_BIT_ERR	Start bit error 0 = No start bit error 1 = Start bit error (start bit not detected on all enabled data signals)	RO	0
10	DAT_BLK_END	Data block end 0 = No data block end 1 = End of data block and CRC passed	RO	0
11	CMD_ACT	Command active 0 = No command active 1 = Command transfer in progress	RO	0
12	TX_ACT	Data transmit active 0 = No data transmit in progress 1 = Data transmit in progress	RO	0
13	RX_ACT	Data receive active 0 = No data receive in progress 1 = Data receive in progress	RO	0
14	TX_FIFO_STAT	Transmit FIFO watermark 0 = No FIFO watermark detected 1 = Transmit FIFO half empty	RO	0
15	RX_FIFO_STAT	Receive FIFO watermark 0 = No FIFO watermark detected 1 = Receive FIFO half full	RO	0
16	TX_FIFO_FULL	Transmit FIFO full 0 = Not full 1 = Transmit FIFO full	RO	0
17	RX_FIFO_FULL	Receive FIFO full 0 = Not full 1 = Receive FIFO full	RO	0
18	TX_FIFO_ZERO	Transmit FIFO empty 0 = Not empty 1 = Transmit FIFO empty	RO	0

Table 21-20. RSI_STATUS Register (Cont'd)

Bit	Name	Function	Type	Default
19	RX_DAT_ZERO	Receive FIFO empty 0 = Not empty 1 = Receive FIFO empty	RO	0
20	TX_DAT_RDY	Transmit data available 0 = No data 1 = Data available in transmit FIFO	RO	0
21	RX_FIFO_RDY	Receive data available 0 = No data 1 = Data available in receive FIFO	RO	0
31:22	Reserved	Reserved	RO	0

RSI Status Clear Register (RSI_STATUSCL)

The RSI_STATUSCL register is used to clear the static flags of the RSI_STATUS register. Write a "1" to any of the bits to clear the corresponding flag in the RSI_STATUS register.

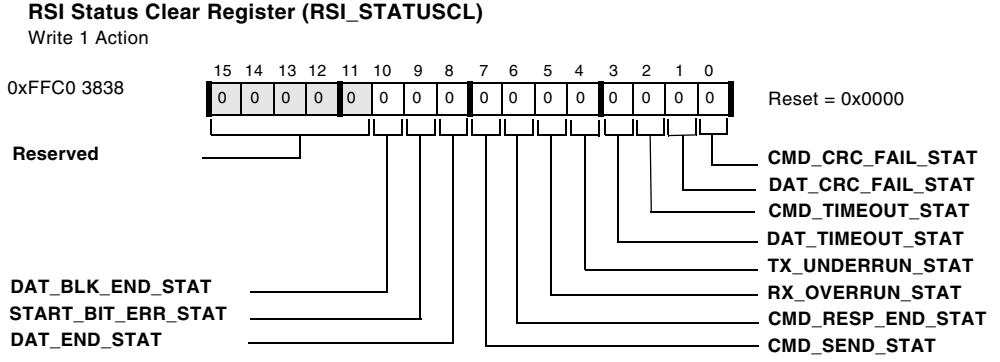


Figure 21-17. RSI Status Clear Register

Table 21-21. RSI_STATUSCL Register

Bit	Name	Function	Type	Default
0	CMD_CRC_FAIL_STAT	Clear command response CRC fail 0 = No effect 1 = Clear CMD_CRC_FAIL	W1A	0
1	DAT_CRC_FAIL_STAT	Clear data CRC failure 0 = No effect 1 = Clear DAT_CRC_FAIL	W1A	0
2	CMD_TIMEOUT_STAT	Clear command timeout 0 = No effect 1 = Clear CMD_TIMEOUT	W1A	0
3	DAT_TIMEOUT_STAT	Clear data timeout 0 = No effect 1 = Clear DAT_TIMEOUT	W1A	0
4	TX_UNDERRUN_STAT	Clear transmit FIFO underrun error 0 = No effect 1 = Clear TX_UNDERRUN	W1A	0
5	RX_OVERRUN_STAT	Clear receive FIFO overrun error 0 = No effect 1 = Clear RX_OVERRUN	W1A	0
6	CMD_RESP_END_STAT	Clear command response received 0 = No effect 1 = Clear CMD_RSEP_END	W1A	0
7	CMD_SENT_STAT	Clear command sent 0 = No effect 1 = Clear CMD_SENT	W1A	0
8	DAT_END_STAT	Clear end of data 0 = No effect 1 = Clear DAT_END	W1A	0
9	START_BIT_ERR_STAT	Clear start bit error 0 = No effect 1 = Clear START_BIT_ERR	W1A	0

Table 21-21. RSI_STATUSCL Register (Cont'd)

Bit	Name	Function	Type	Default
10	DAT_BLK_END_STAT	Clear data block end 0 = No effect 1 = Clear DAT_BLK_END	W1A	0
15:11	Reserved	Reserved	W1A	0

RSI Interrupt Mask Registers (RSI_MASKx)

The RSI_MASKx registers (RSI_MASK0 and RSI_MASK1) determine which of the static and dynamic flags of the RSI_STATUS register generate an interrupt request to the SIC via one of the two available RSI interrupts. An interrupt is enabled by setting the corresponding bit in the RSI_MASKx register to 1. Interrupts enabled in the RSI_MASK0 register will result in an IRQ being sent via the IRQ0 signal of the RSI, and interrupts enabled in the RSI_MASK1 register generate an IRQ on the IRQ0 signal of the RSI.

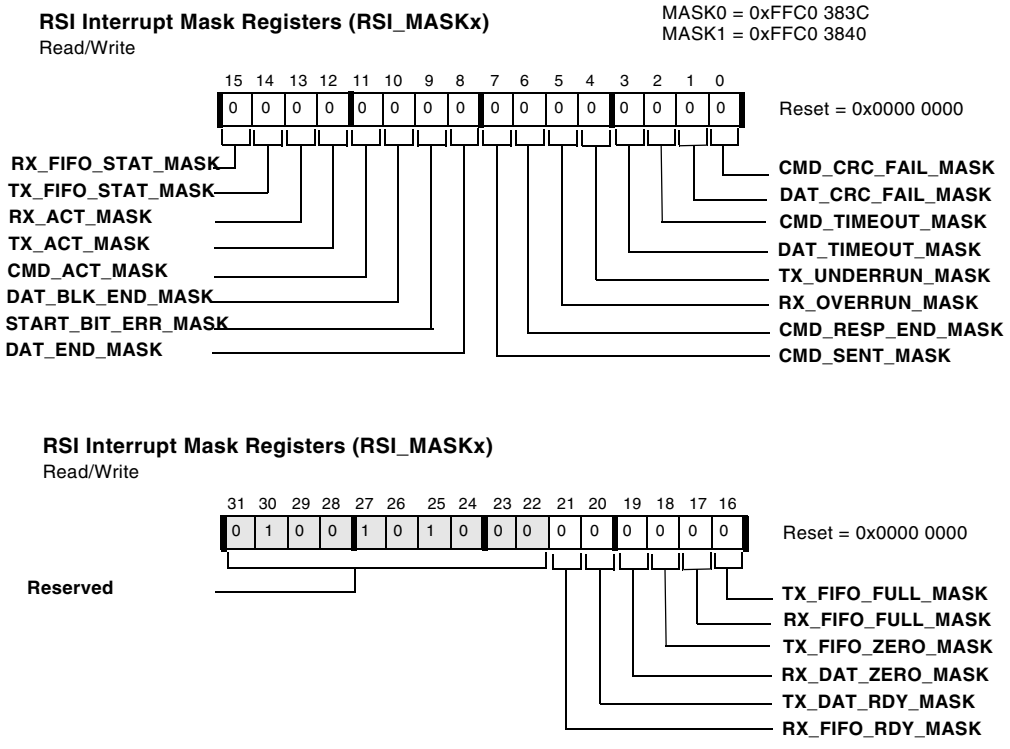


Figure 21-18. RSI Interrupt Mask Registers

Table 21-22. RSI_MASKx Registers

Bit	Name	Function	Type	Default
0	CMD_CRC_FAIL_MASK	Command response CRC fail 0 = Disable interrupt 1 = Enable interrupt	R/W	0
1	DAT_CRC_FAIL_MASK	Data CRC failure 0 = Disable interrupt 1 = Enable interrupt	R/W	0
2	CMD_TIMEOUT_MASK	Command timeout 0 = Disable interrupt 1 = Enable interrupt	R/W	0

Table 21-22. RSI_MASKx Registers (Cont'd)

Bit	Name	Function	Type	Default
3	DAT_TIMEOUT_MASK	Data time out 0 = Disable interrupt 1 = Enable interrupt	R/W	0
4	TX_UNDERRUN_MASK	Transmit FIFO underrun error 0 = Disable interrupt 1 = Enable interrupt	R/W	0
5	RX_OVERRUN_MASK	Receive FIFO overrun error 0 = Disable interrupt 1 = Enable interrupt	R/W	0
6	CMD_RESP_END_MASK	Command response received 0 = Disable interrupt 1 = Enable interrupt	R/W	0
7	CMD_SENT_MASK	Command sent 0 = Disable interrupt 1 = Enable interrupt	R/W	0
8	DAT_END_MASK	End of data 0 = Disable interrupt 1 = Enable interrupt	R/W	0
9	START_BIT_ERR_MASK	Start bit error 0 = Disable interrupt 1 = Enable interrupt	R/W	0
10	DAT_BLK_END_MASK	Data block end 0 = Disable interrupt 1 = Enable interrupt	R/W	0
11	CMD_ACT_MASK	Command active 0 = Disable interrupt 1 = Enable interrupt	R/W	0
12	TX_ACT_MASK	Data transmit active 0 = Disable interrupt 1 = Enable interrupt	R/W	0
13	RX_ACT_MASK	Data receive active 0 = Disable interrupt 1 = Enable interrupt	R/W	0

Table 21-22. RSI_MASKx Registers (Cont'd)

Bit	Name	Function	Type	Default
14	TX_FIFO_STAT_MASK	Transmit FIFO watermark 0 = Disable interrupt 1 = Enable interrupt	R/W	0
15	RX_FIFO_STAT_MASK	Receive FIFO watermark 0 = Disable interrupt 1 = Enable interrupt	R/W	0
16	TX_FIFO_FULL_MASK	Transmit FIFO full 0 = Disable interrupt 1 = Enable interrupt	R/W	0
17	RX_FIFO_FULL_MASK	Receive FIFO full 0 = Disable interrupt 1 = Enable interrupt	R/W	0
18	TX_FIFO_ZER/W_MASK	Transmit FIFO empty 0 = Disable interrupt 1 = Enable interrupt	R/W	0
19	RX_DAT_ZER/W_MASK	Receive FIFO empty 0 = Disable interrupt 1 = Enable interrupt	R/W	0
20	TX_DAT_RDY_MASK	Transmit data available 0 = Disable interrupt 1 = Enable interrupt	R/W	0
21	RX_FIFO_RDY_MASK	Receive data available 0 = Disable interrupt 1 = Enable interrupt	R/W	0
31:22	Reserved	Reserved	R/W	0

RSI FIFO Counter Register (RSI_FIFO_CNT)

The RSI_FIFO_CNT register contains a value indicating the number of 32-bit words still to be read from or written to the FIFO. RSI_FIFO_CNT is loaded from the RSI_DATA_LGTH register when the DATA_EN bit of the RSI_DATA_CONTROL register is set. If the data length is not word-aligned (multiple of 4), the remaining 1 to 3 bytes are regarded as a word.

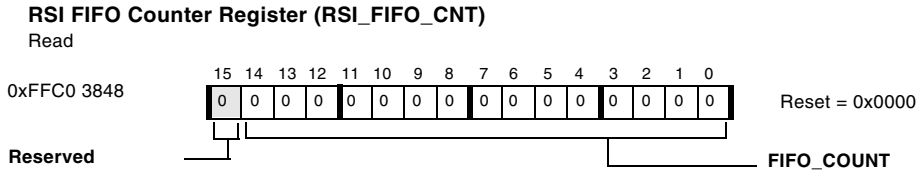


Figure 21-19. RSI FIFO Counter Register

Table 21-23. RSI_FIFO Register

Bit	Name	Function	Type	Default
14:0	FIFO_COUNT	Number of 32-bit words remaining	RO	0
15	Reserved	Reserved	RO	0

RSI CE-ATA Control Register (RSI_CEATA_CONTROL)

The RSI_CEATA_CONTROL register contains bits applicable to CE-ATA mode of operation. CEATA_TX_CCSD, when set, results in the RSI sending the command completion signal disable sequence to the CE-ATA device to notify the device not to send back the command completion signal. The CEATA_TX_CCSD bit is a write-1-action bit and remains set until actively cleared. If the bit is set prior to enabling the command path state machine, the CCSD signal will automatically be sent after the response is received from the CE-ATA device and the command path state machine will return to the IDLE state.

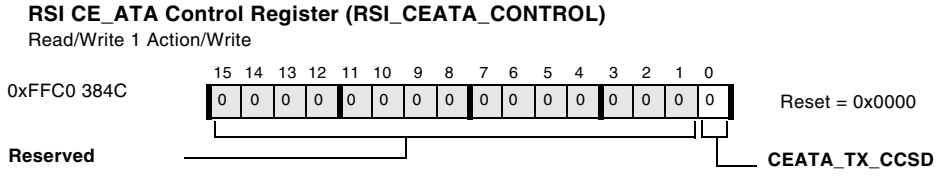


Figure 21-20. RSI CE_ATA Control Register

Table 21-24. RSI_CEATA_CONTROL Register

Bit	Name	Function	Type	Default
0	CEATA_TX_CCSD	Transmit command completion signal disable 0 = No action 1 = Send command completion signal disable sequence	R/W1A/W	0
15	Reserved	Reserved	-	0

RSI Data FIFO Register (RSI_FIFO)

The RSI_FIFO register provides access to the 16-entry transmit and receive FIFO. The register is accessed as a 32-bit word.

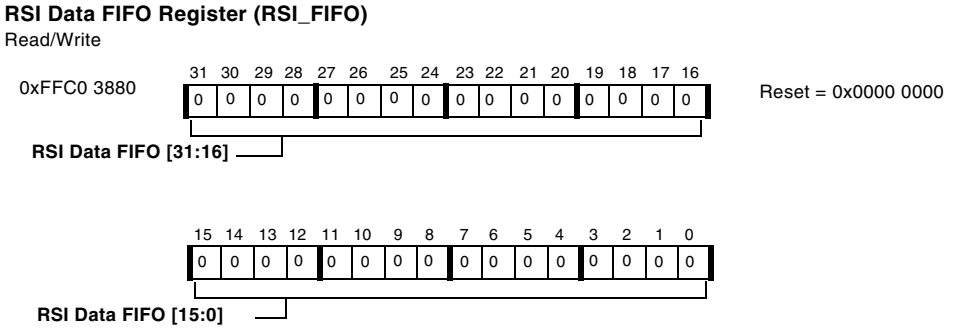


Figure 21-21. RSI Data FIFO Register

RSI Exception Status Register (RSI_ESTAT)

The RSI_ESTAT register contains exception status bits for SDIO cards, CE-ATA devices, and the card detection logic. These status bits can be uses to generate an interrupt request via the IRQ0 signal by enabling the interrupt in the RSI_EMASK register. All bits in this register are write-1-to-clear bits. The SDIO interrupt is an interrupt generated by SDIO cards on the RSI_DATA1 signal. The SD_CARD_DET bit is set when a

rising edge is detected on the RSI_DATA3 signal and is intended for use with devices that support card detection using this signal. CEATA_INT_DET indicates whether the command completion response has been received from the attached CE-ATA device, indicating that the ATA operation has completed successfully or that ATA command termination has occurred as the result of an error condition.

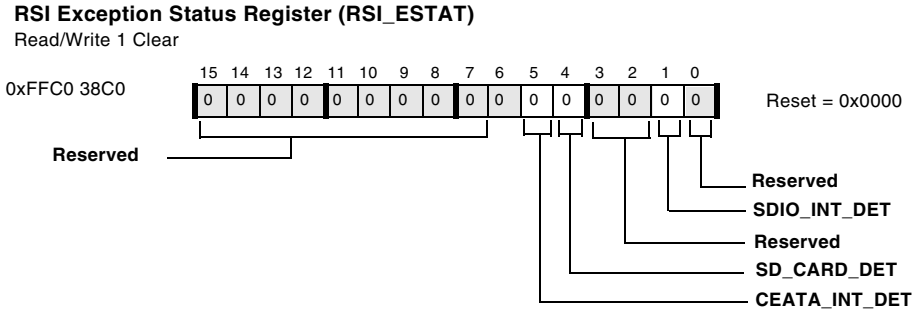


Figure 21-22. RSI Exception Status Register

Table 21-25. RSI_ESTAT Register

Bit	Name	Function	Type	Default
0	Reserved	Reserved	RO	0
1	SDIO_INT_DET	SDIO interrupt detect 0 = No interrupt detected 1 = Interrupt detected	R/W1C	0
3:2	Reserved	Reserved	RO	0
4	SD_CARD_DET	Card detect interrupt 0 = No interrupt detected 1 = Interrupt detected	R/W1C	0
5	CEATA_INT_DET	Command completion signal detect 0 = No CCS detected 1 = CCS detected	R/W1C	0
15:6	Reserved	Reserved	RO	0

RSI Exception Mask Register (RSI_EMASK)

The RSI_EMASK register contains mask bits for the RSI_ESTAT status bits. Writing a "1" to the RSI_EMASK bit enables the interrupt for the corresponding bit in the RSI_ESTAT register.

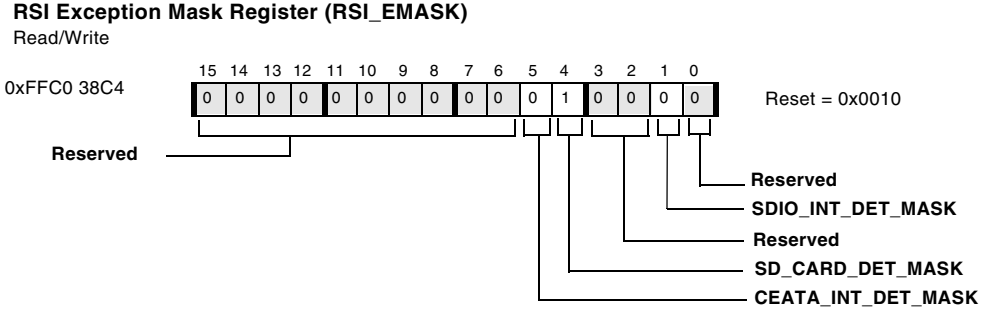


Figure 21-23. RSI Exception Mask Register

Table 21-26. RSI_EMASK Register

Bit	Name	Function	Type	Default
0	Reserved	Reserved	R/W	0
1	SDIO_INT_DET_MASK	SDIO interrupt enable 0 = Interrupt disabled 1 = Interrupt enabled	R/W	0
3:2	Reserved	Reserved	R/W	0
4	SD_CARD_DET_MASK	Card detect interrupt enable 0 = Interrupt disabled 1 = Interrupt enabled	R/W	0
5	CEATA_INT_DET_MASK	Command completion signal detect enable 0 = Interrupt disabled 1 = Interrupt enabled	R/W	0
15:6	Reserved	Reserved	RO	0

RSI Configuration Register (RSI_CONFIG)

The RSI_CONFIG register controls bits that enable and disable portions of the RSI module. The RSI_CLK_EN bit must be set in order to enable the RSI for operation. After reset, PD_DAT3 is set. This bit is mutually exclusive with PU_DAT3, therefore if the pull-up resistor is to be enabled on RSI_DATA3, then PD_DAT3 should be cleared. If an external pull-down resistor is used for implementing card detection on the RSI_DATA3 signal, then PU_DAT3 should not be set. The pull-up and pull-down resistors on the RSI_DATAx signals become active only when the corresponding GPIO pins are configured for RSI functionality via the pin multiplexing. For example, if only the 4-bit data bus is enabled in the pin multiplexing, setting PU_DAT will only enable the pull-up resistors on the signals that are configured for RSI use. The RSI_CONFIG register also provides additional functionality for SDIO support. To enable SDIO 4-bit mode, in addition to setting the bus width to 4-bit via the BUS_MODE field of the RSI_CLK_CONTROL register, SDIO4_EN should be set. The MW_EN bit, when set, allows for SDIO interrupts to be detected outside the specified one-cycle window and is set when interrupt support is required during multiple block read transactions from SDIO. The RSI can also be reset with the RSI_RST bit. Writing this bit resets the RSI module and returns all registers to their default values.

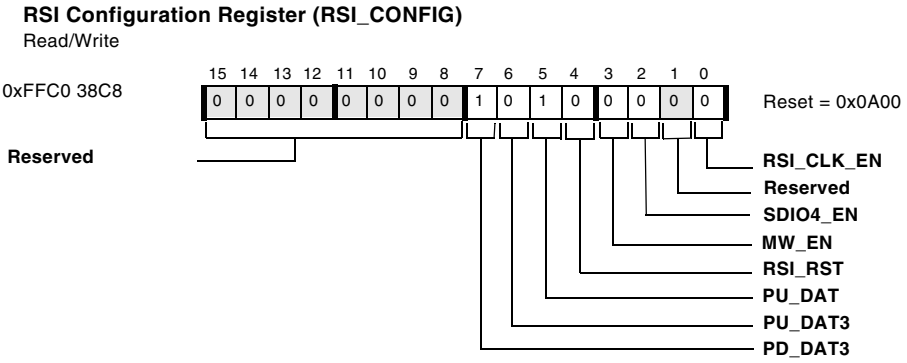


Figure 21-24. RSI Configuration Register

Table 21-27. RSI_CONFIG Register

Bit	Name	Function	Type	Default
0	RSI_CLK_EN	RSI clocks enable 0 = Disable internal RSI clocks 1 = Enable internal RSI clocks	R/W	0
1	Reserved	Reserved	R/W	0
2	SDIO4_EN	SDIO 4-bit enable 0 = Disable SDIO 4-bit mode 1 = Enable SDIO 4-bit mode	R/W	0
3	MW_EN	SDIO interrupt moving window enable 0 = Disabled 1 = Enabled (required when using SDIO multiple block read operations)	R/W	0
4	RSI_RST	RSI reset 0 = No action 1 = Reset the RSI	R/W	0
5	PU_DAT	Pull-up enable 0 = Disable pull-up resistor on RSI_DATA7-4 and RSI_DATA2-0 1 = Enable pull-up resistor on RSI_DATA7-4 and RSI_DATA2-0	R/W	0

Table 21-27. RSI_CONFIG Register (Cont'd)

Bit	Name	Function	Type	Default
6	PU_DAT3	RSI_DATA3 pull-up enable 0 = Disable pull-up resistor on RSI_DATA3 1 = Enable pull-up resistor on RSI_DATA3	R/W	0
7	PD_DAT3	RSI_DATA3 pull-down enable 0 = Disable pull-down resistor on RSI_DATA3 1 = Enable pull-down resistor on RSI_DATA3 For more system flexibility, no internal pull-down resistor is present. An external pull-down resistor is required for card detection capability on the RSI_DATA3 signal.	R/W	0
15:8	Reserved	Reserved	RO	0

RSI Read Wait Enable Register (RSI_RD_WAIT_EN)

The RSI_RD_WAIT_EN register contains the SDIO_RWR bit that, when set, issues a read wait request to an SDIO card. Once software is ready to resume the data transfer, this bit must be cleared. The functionality applies to both 1-bit and 4-bit SDIO modes.

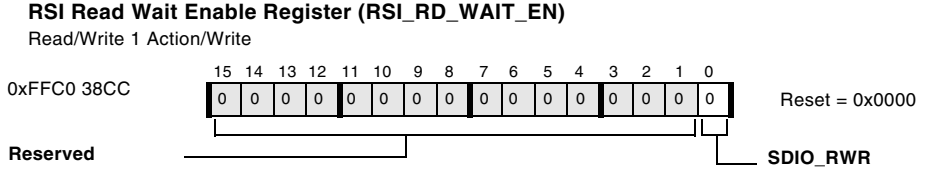


Figure 21-25. RSI Read Wait Enable Register

Table 21-28. RSI_RD_WAIT_EN Register

Bit	Name	Function	Type	Default
0	SDIO_RWR	RSI read wait request enable 0 = Normal operation 1 = Issue read wait request to SDIO device	R/W1A/W	0
15:1	Reserved	Reserved	RO	0

RSI Peripheral ID Registers (RSI_PIDx)

The RSI_PIDx registers (RSI_PID0, RSI_PID1, RSI_PID2, RSI_PID3, RSI_PID4, RSI_PID5, RSI_PID6, and RSI_PID7) contain a fixed value at reset and are used to identify the peripheral revision. There are a total of four 16-bit identification registers of which the lower 8-bits are valid. The contents of these four registers are listed in Table 21-30.

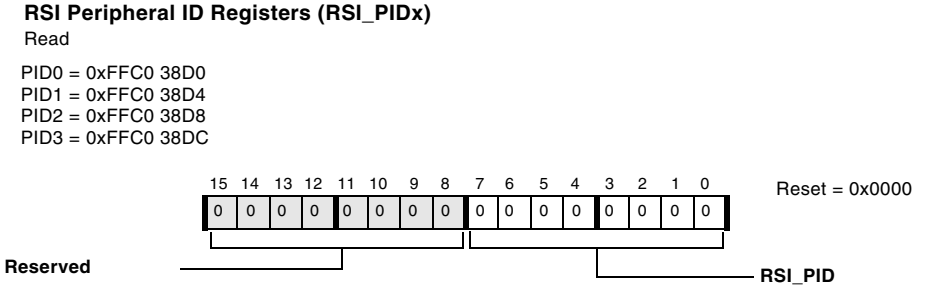


Figure 21-26. RSI Peripheral ID Registers

Table 21-29. RSI_PIDx Registers

Bit	Name	Function	Type	Default
7:0	RSI_PID	Peripheral ID	RO	0
15:8	Reserved	Reserved	RO	0

Table 21-30. Peripheral IDs

RSI Peripheral ID Register	RSI_PID Value
RSI_PID0	0x80
RSI_PID1	0x11
RSI_PID2	0x04
RSI_PID3	0x00

22 ADC CONTROL MODULE (ACM)

The ADC control module (ACM) on the ADSP-BF50x processor provides an interface that synchronizes the controls between the processor and an analog-to-digital converter (ADC). The analog-to-digital conversions are initiated by the processor, based on either external or internal events.

Traditionally, ADC sampling uses processor interrupts (initiated by the events) and the interrupt service routine programming of the appropriate peripheral (usually SPORT or SPI) for initiating the ADC conversion process. This traditional approach has some limiting factors:

- ADC sampling instances are not precisely controlled due to interrupt latencies (which can vary) or due to variable instruction execution cycles
- Consumption of processor MIPS can be prohibitive, especially for high frequency of conversion-related events.

The ADC control module (ACM) answers the limitations of the traditional approach to sampling by providing dedicated hardware, which samples the events and provides sampling signals to the ADC real-time. The ACM approach both saves processor MIPS and provides precise controllability for ADC sampling time. The ACM synchronizes the ADC conversion process (generating the ADC controls, ADC conversion start signal, and related controls), but the actual data acquisition from the ADC is accomplished by other peripherals.

On the ADSP-BF50x processors, the ADC module is used either with SPORT0/1. The ADSP-BF50x processors do not support ACM operation with SPI.

Figure 22-1 and Figure 22-2 show how an external or internal ADC is connected to the ACM and the SPORT.

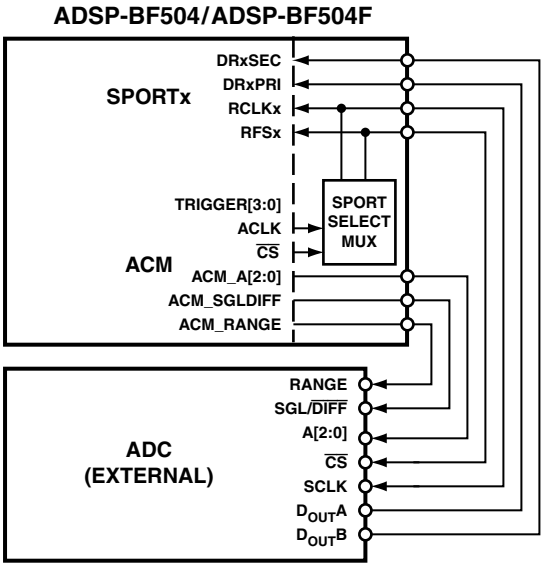


Figure 22-1. ADSP-BF504F – ACM Connections (for External ADC)

Comparing Figure 22-1 and Figure 22-2, note that the connections differ between ADSP-BF50x processors that include an internal ADC versus the processors that do not include an internal ADC.

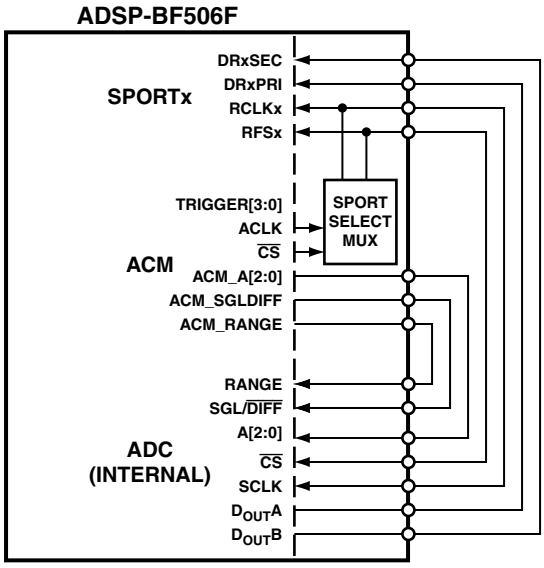


Figure 22-2. ADSP-BF506F – ACM Connections (for Internal ADC)

Interface Overview

The ACM consists of two independent 32-bit timers, 16 event register pairs, 16 event comparators, and a timing generation unit. The ACM can accept four external trigger inputs, with the timers set to start counting on receiving the external triggers (based on the mode of the ACM). The triggers can be independently selected for each timer. Each timer can be independently enabled. The two sets of 8 event register pairs (total of 16 event register pairs) determine the ADC controls and timing for each ADC sampling interval.

Each event register pair consists of an event control register (ACM_ERx) and an event time register (ACM_ETx). The ACM_ERx register enables a particular event and determines settings for the ADC controls (A[2:0], RANGE, SGL-DIFF, and others) for that particular ADC conversion. The ACM_ETx

register determines the time offset from the external trigger for each event sampling at the ADC.

Table 22-1. ACM Interface Pins

Pin	I/O	Description
ACM_A[2:0]	O	ADC Channel Select. 3-bit ADC Channel select signal
ACM_RANGE	O	Range Selector
ACM_SGLDIFF	O	Mode. Single Ended/Differential Mode selector
TRIGGER[3:0]	I	Trigger Inputs. Generated from external trigger events
$\overline{CS}$	O	Start of Conversion. Chip select for ADC, and connected as Frame Sync for SPORT
ACLK	O	ADC Clock. Clock output for ADC and SPORT

Figure 22-3 shows the ACM Block diagram.

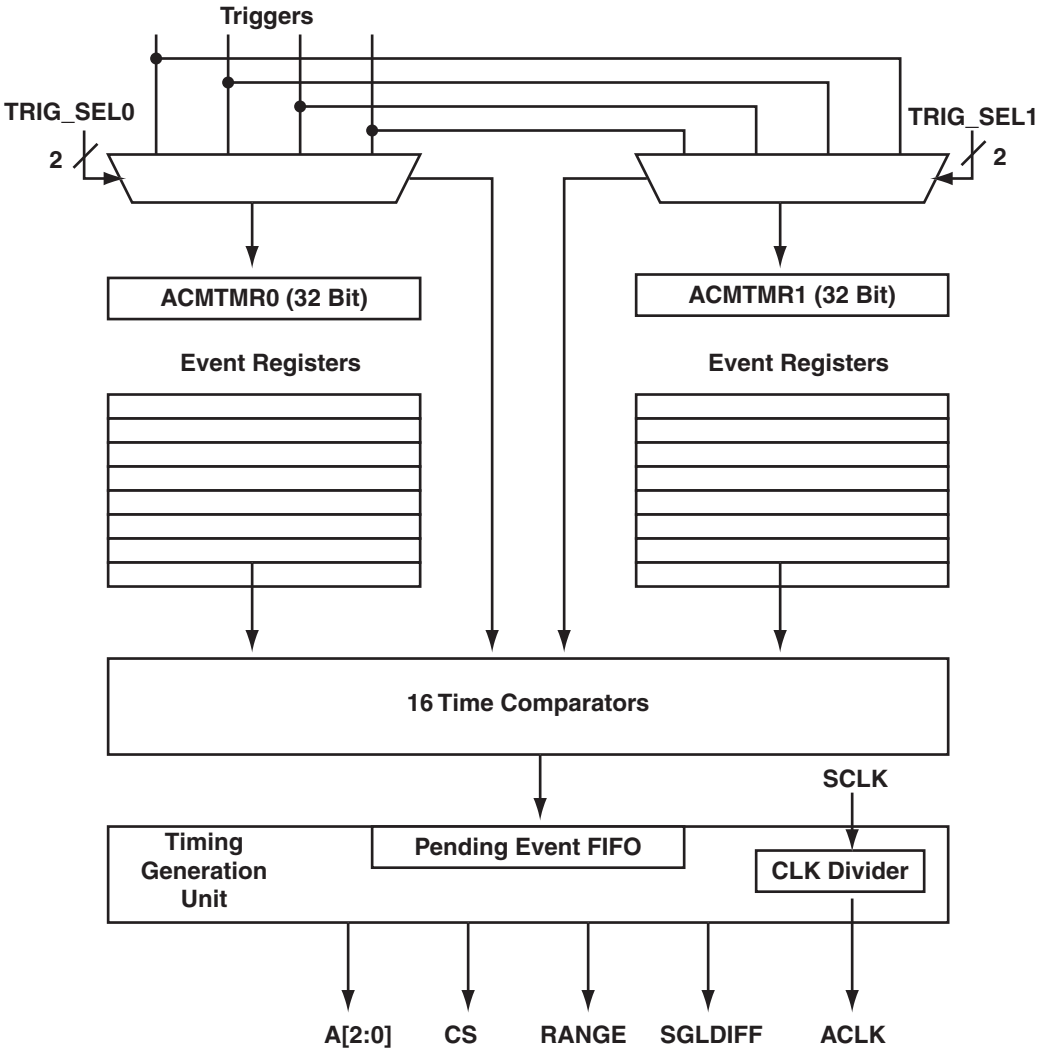


Figure 22-3. ACM Block Diagram

Events

An event, for the ACM, is a point in time where ADC sampling has to happen on a particular channel of the ADC with the specified control settings of the ADC.The parameters for the ADC sampling are specified in the ACM_ERx register, and the time-offset from the external trigger for the sampling of the ADC for that event (when the event has to occur) is determined by the ACM_ETx register.

Timers

The ACM has two independent 32-bit timers (ACMTMR0, ACMTMR1) that count based on system clock (SCLK). The timers can be independently enabled using the timer enable bits (ACMTMR0EN, ACMTMR1EN) in the ACM_CTL register. Each timer (by default) has 8 event register pairs associated with it. If both timers are enabled at the same time, the ACM_ER0-7 and ACM_ET0-7 event register pairs are associated with ACMTMR0, and the ACM_ER8-15 and ACM_ET8-15 event register pairs are associated with ACMTMR1. If only one timer is enabled, all of the event registers are associated with that particular timer. For example, if only ACMTMR1 is enabled, (if ACMTMR0EN=0 and ACMTMR1EN=1), the ACM_ER0-15 and ACM_ET0-15 event register pairs are associated with ACMTMR1. The timers start counting when an external trigger occurs that is selected for that particular timer. If an external trigger occurs while the timer is counting, the time resets and starts counting again.

After a trigger, the timer only stops counting under one of the following conditions:

1. A timer rollover occurs
2. All the events associated with the trigger have completed

Note that a timer rollover can never happen *unless* the ACM_ET register is programmed at some point after the trigger occurs; this is a practice that is contrary to ACM programming guidelines.

When an ACM timer is disabled or the ACM itself is disabled, the timer resets to zero.

External Triggers

Each ACM timer ACMTMRx may be triggered by one of the following trigger signals:

- ACM trigger input 0 (TRIGGER0) – PWM_SYNC0: The PWM_SYNC0 trigger may be either internally-generated by the PWM unit or externally-generated, depending on the configuration specified in the PWM0 module.
- ACM trigger input 1 (TRIGGER1) – PWM_SYNC1: The PWM_SYNC1 trigger may be either internally-generated by the PWM unit or externally-generated, depending on the configuration specified in the PWM1 module.
- ACM trigger input 2 (TRIGGER2) – Port F GPIO at PF10 or TMR2. When the Port F PF10 pin is configured in function mode (non-GPIO mode), then the ACM trigger input 2 is sourced from the output of TMR2. When the Port F PF10 pin is configured in GPIO mode, then the ACM trigger input 2 is sourced from the GPIO signal at PF10. The source of the GPIO signal may be either internal or external depending on the GPIO direction configuration programmed in PORTFIO_DIR.
- ACM trigger input 3 (TRIGGER3) – Port G GPIO at PG5 or TMR7. When the Port G PG5 pin is configured in function mode (non-GPIO mode), then the ACM trigger input 3 is sourced from the output of TMR7. When the Port G PG5 pin is configured in GPIO mode, then the ACM trigger input 3 is sourced from the GPIO signal at PG5. The source of the GPIO signal may be either internal or external depending on the GPIO direction configuration programmed in PORTGIO_DIR.

For all trigger signals, The active edge of the triggers is programmable in the ACM_CTL register as either rising edge or falling edge. Figure 22-4 shows the detailed ACM trigger generation logic.

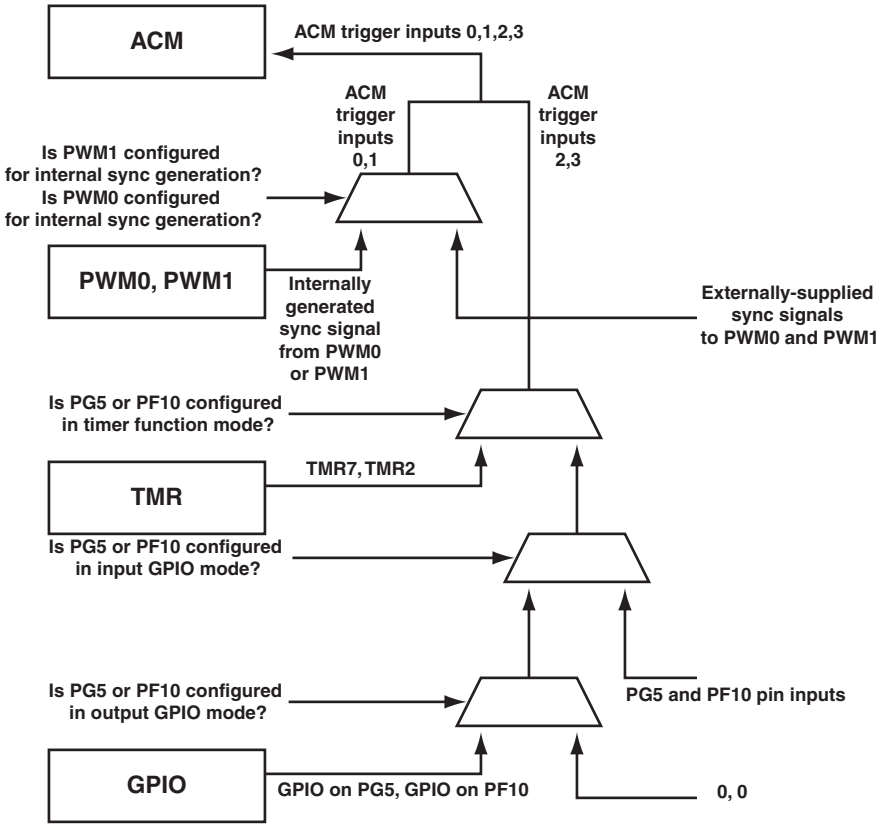


Figure 22-4. ACM Trigger Logic

When trigger sources external to the processor are used for triggering the ACM, (for example, external signals at the GPIO, timer, or PWM sync pins), the minimum pulse width for such trigger sources needs to be 1 SCLK period + 1 ns.

A latency of no more than 4 SCLK cycles exists between external trigger and ACMTMRx count start. Please refer to "ADC Sampling Latency" on page 22-18 for further details.

Event Register Pairs

The ACM has 16 event register pairs. Each pair consists of an ACM_ERx register and an ACM_ETx register. The ACM_ERx register enables the particular event and determines the ADC control settings for the particular event. The ACM_ETx register determines when the ADC sampling happens corresponding to the event. Assignment of the 16 event register pairs: *either* 8 can be assigned to each of the timers (if both timers are enabled) *or* all 16 can be assigned to one particular timer (if only one timer is enabled).

Event Comparators

There are 16 event time comparators to determine when an enabled event should happen. The comparators compare the event time with the corresponding timer count. If the time value matches, the comparators signal an active event signal to the timing generation unit. If more than one event is active during the same SCLK cycle, only the highest priority event is processed, and all other events are missed (even if there was space in the pending event FIFO). The priority of events is fixed, with event 0 having highest priority and event 15 having lowest priority.

Timing Generation Unit

The timing generation unit generates the ADC control signals based on the ACM_ERx register setting. The timings of external signals (ACLK, $\overline{\text{CS}}$, A[2:0], RANGE, and others) are determined by the ACM_TCx registers. If an event happens when another event is ongoing, the occurred event is stored in the pending event FIFO. After the current event completes, the pending event is serviced (for example, the ACM starts an ADC conversion for the event that occurred). If an event occurs when the pending event FIFO

is full, that event is missed. If an event is missed, the EMISS bit is set in the ACM_STAT register, and the corresponding bit in the ACM_MS register also is set. An optional missed event interrupt is generated if the missed event interrupt enable for that event is unmasked in the ACM_EMSK register.

The pending event FIFO has a depth of 4, so it can hold up to 4 pending events, after which the events are missed.

Interrupts

The ACM can generate two interrupts for each event: event completed or event missed. These can be selectively enabled by using the ACM_IMSK and ACM_EMSK registers. The event completion interrupt is generated only after the entire event completes externally (for example, when $\overline{CS}$ goes inactive, T_H period and T_Z periods are completed for that particular event).

The ACM_ES register provides the status of each event indicating which event created the interrupt and the event completion interrupt can be cleared by writing to the relevant W1C bit in the ACM_ES register.

The ACM_MS register provides the status of each missed event indicating which event miss created the interrupt and the missed event interrupt can be cleared by writing the relevant W1C bit in the ACM_MS register.

A Status bit set either in ACM_MS or ACM_ES creates an interrupt only if the corresponding bit in the ACM_IMSK or ACM_EMSK is enabled.

Description of Operation

This section describes the usage modes of the ACM, including how to use the ACM for implementing power down mode for the ADC and how to implement various sequencing modes.

ADC Power Down

The internal ADC available on ADSP-BF506F devices may be transitioned into a power-down mode by asserting and deasserting the $\overline{CS}$ signal for a number of clock cycles, as described in the "ADC — Modes of Operation" section of the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*. This ADC power-down mode of the internal ADC—and external ADCs that support a similar power-down mechanism—may be achieved by issuing a "dummy" ADC event (or events) after appropriately programming the T_{CSW} field in the ACM_TC register.

Single-Shot Sequencing Mode Emulation

In single-shot sequencing mode, all enabled events are sequentially issued one after the other on the occurrence of an ACM trigger. The sequence of events is fixed, starting with event 0 and ending with event 15.

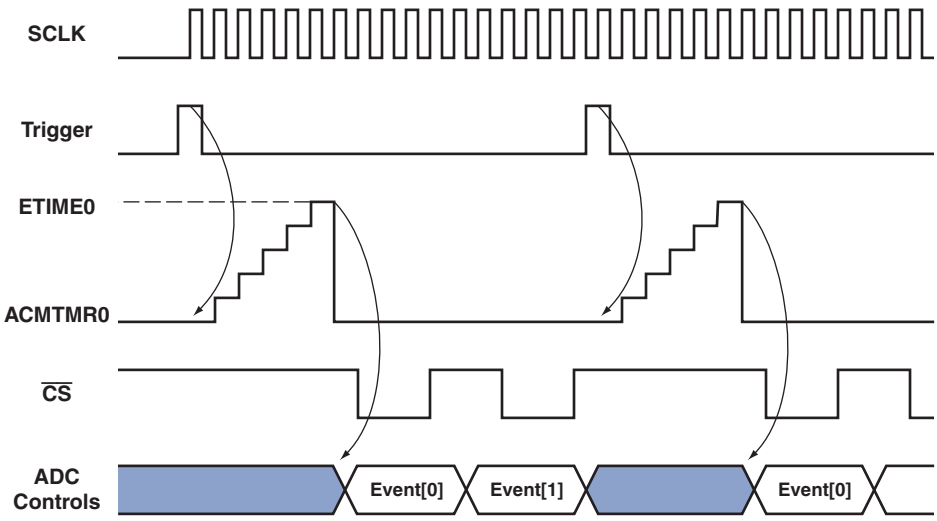


Figure 22-5. Single-Shot Sequencing Mode Requirement

Figure 22-5 shows an example of single-shot sequencing mode where only event 0 and event 1 are enabled. ETIME0 is the value written into the ACM_ET0 register, and ACMTMR0 is enabled in this mode.

To emulate this mode of operation using the ACM:

- Configure the appropriate trigger source for initiating ACM activity. Please refer to "Interface Overview" on page 22-3 for information on signals that can trigger the ACM counters.
- Enable only one ACM timer (ACMTMR0)
- Enable events and program the event time values as:

  Event 0 time = X

  Event 1 time = X+Y

  Event 2 time = X+2Y

  where X = ETIME0, the initial time offset from trigger (if needed)

  $Y = T_H+T_{CSW}+T_S+T_Z$

  Where T_H is the hold time, T_Z is the zero t time, and T_S is the setup time; for more information, see "ACM External Pin Timing" on page 22-20 and Figure 22-9 on page 22-20

  (Y has to be slightly less than the above value to ensure that the next event happens before the first event completes, so that the next event is in the pending FIFO and enables the transitions between events without a break)

Continuous Sequencing Mode Emulation

Continuous sequencing mode is similar to single-shot sequencing mode, except in continuous sequencing the event sequencing is continuously

repeated. As in single-shot mode, the time offset is programmable in continuous mode. The trigger in continuous mode is relevant only for the first time. Therefore, any subsequent triggers after the first active edge of the trigger are neglected.

Figure 22-6 shows an example of continuous sequencing mode with only two events – event 0, event 1 enabled.

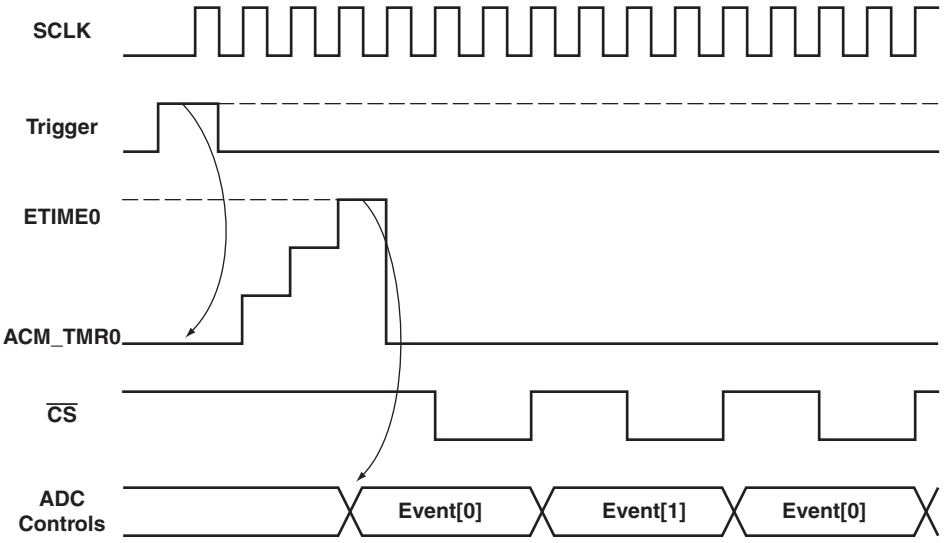


Figure 22-6. Continuous Sequencing Mode Requirement

Figure 22-6 does not exactly reflect exact internal ACM operation. It shows the user requirement to sample the ADC based on events in a particular sequence.

To emulate continuous sequencing mode using ACM:

- Enable only one timer (Timer0)
- Enable events and program the event time values as:

  Event 0 time = X

  Event 1 time = X+Y

  Event 2 time = X+2Y

  where X,Y value are as described in the single-shot case (Y can be slightly less than $T_H+T_{CSW}+T_S+T_Z$ to avoid any break between Events)

- Enable one of the two general purpose timers (TMR2 and TMR7) to generate an ACM trigger with the following time period:

  Timer time period = N*Y

  where N = number of enabled events

  Y = $T_H+T_{CSW}+T_S+T_Z$ (Y should be exactly equal here)

  For more information on ACM triggers, refer to the "Interface Overview" on page 22-3.

Continuous sequencing may also be implemented without the use of the general purpose timers TMR2 or TMR7, but through the use of general purpose I/O (GPIO) pin. This may be achieved by using a GPIO as a trigger and by enabling the event completion interrupt for an event that precede the last event. In the interrupt service routine, the GPIO is programmed to generate a new trigger in order to repeat the sequencing of events again.

The interrupt should be triggered early enough to account for any offset value (for example, `ETIME0` value)

Functional Description

The ACM uses the ACM timers and the event time register to create events. The user has to enable one of the timers (or both timers) for the ACM operation. Appropriate event control register and event time register values also have to be programmed. After receiving an external trigger, the timer starts counting. If at anytime the timer count matches the enabled event time (ACM_ETx) for an event associated with the timer, the comparators generate an event signal to the timing generation unit to start the ADC access. The counter continues counting, and (for each matching and enabled event) the ACM gives an event signal to the timing generation unit.

Figure 22-7 shows the ACM operation where only two events (event 0 and event 3) are enabled.

In Figure 22-7, the line labeled “ADC Controls” depicts the timing of the ADC control signals: A[2:0], RANGE, and SGLDIFF. Note that Figure 22-7 depicts a usage case in which ACM_ET3 is programmed with a count value that is greater than that programmed in ACM_ET0. So, event 3 occurs after event 0.

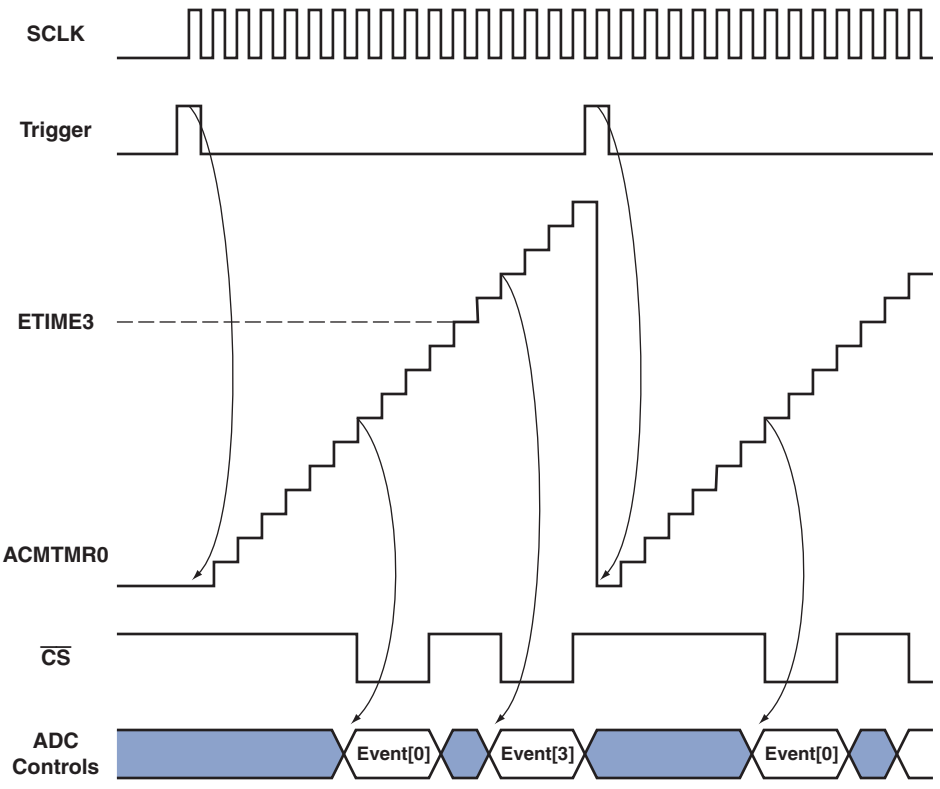


Figure 22-7. Mode Of Operation

There are, however, no restrictions on the order of the different events. ACM_ET0 can be greater than ACM_ET3, ACM_ET0 can be less than ACM_ET3, and ACM_ET0 can be equal to ACM_ET3. When the value in ACM_ET0 is greater than the value in ACM_ET3, event 0 occurs after event 3. When multiple events have the same event time values, for example the value in ACM_ET0 is equal to the value in ACM_ET3, the events are processed according to their priorities. Event 0 has the highest priority. So, when event 0 and event 3 share the same event time value, the timing generation unit processes event 0 while event 3 is missed. The event 3 missed (EM3) status bit is set in the missed event status register (ACM_MS), and the EMISS bit is set in the ACM_STAT register, indicating that an event has been missed. In

case more than two simultaneous events occur, only the highest priority event is processed, and all the lower priority events are missed (even if the pending event FIFO had space). In this case, the appropriate bits are set in the ACM_MS, and the EMISS bit is set in ACM_STAT.

If event times are not sufficiently spaced apart, an event could occur while a previous event is underway (while the $\overline{\text{CS}}$ of the previous event is asserted). In such a situation, the second event is queued in the pending event FIFO. If the pending event FIFO is full, the event will not get queued. Instead, the appropriate event miss indicators will be set in the ACM_MS and ACM_STAT registers. It is the programmers responsibility to ensure that the values in the event time registers do not lead to event misses. On disabling the ACM, all pending events in the FIFO are flushed.

When events that are triggered by both timers (ACMTMR0 and ACMTMR1) occurs simultaneously, the events triggered by ACMTMR0 are given higher priority. For example, when an ACMTMR0 event (one of events 0 through 7) and ACMTMR1 event (one of events 8 through 15) occur simultaneously, the ACMTMR0 event is processed by the timing generation unit or is queued in the pending event FIFO before the processing or the queuing of the ACMTMR1 event.

When ACMTMR0 and ACMTMR1 are triggered by sources that are not synchronized together, it is possible for the events controlled by the two timers to overlap. It is therefore important to consider the possibility of events occurring *either* simultaneously *or* being missed when enabling events on two asynchronously-triggered timers.

When all the events enabled for a given ACM timer (ACMTMRx) are processed, the ACM timer stops incrementing. (Note that this timer action is not reflected in Figure 22-7.)

The ACM can be used to generate various sequences of ADC sampling events through appropriate programming of event time registers, event control registers, and triggers. For more information, see the usage cases

described in "Single-Shot Sequencing Mode Emulation" on page 22-11 and "Continuous Sequencing Mode Emulation " on page 22-12.

ADC Sampling Latency

The ACM ensures a predictable latency between the internal occurrence of an event (event time value matching the timer count value) and the assertion of a sampling event by the timing generation unit (the assertion of $\overline{CS}$ and other ACM signals as appropriate).

Latency of Event to $\overline{CS}$ active = (T_S + T_{ED}) SCLK cycles

Where:

- T_S = ADC control setup cycles programmed in `ACM_TC` register
- T_{ED} = 1 SCLK

This predictable latency is applies only when the timing generation unit is idle. If the timing generation unit was processing a prior sampling event, the new event will be held in the pending event FIFO, and the latency will increase by the duration that the new event is held in the pending event FIFO.

The latency between the occurrence of an external trigger to the start of count of an ACM timer is three to four SCLK cycles.

The one SCLK cycle variability is due to delays in latching asynchronous external triggers. When the external trigger is synchronous to SCLK, the one SCLK cycle variability is eliminated and the latency from external trigger to start of count of an ACM timer becomes fixed at three SCLK cycles.

As a result, the total latency between an external trigger and between the assertion of an ADC sampling event, assuming that the sampling event does is not queued in the pending event FIFO, is:

Total Latency = T_{TRIG} + T_{ED} + T_{PD} + T_S

Where:

T_{PD} is the delay programmed in the Event Time (ACM_ETx) register. (See Figure 22-8.)

Figure 22-8 shows latency details from occurrence of external triggers to assertion to ADC sampling events.

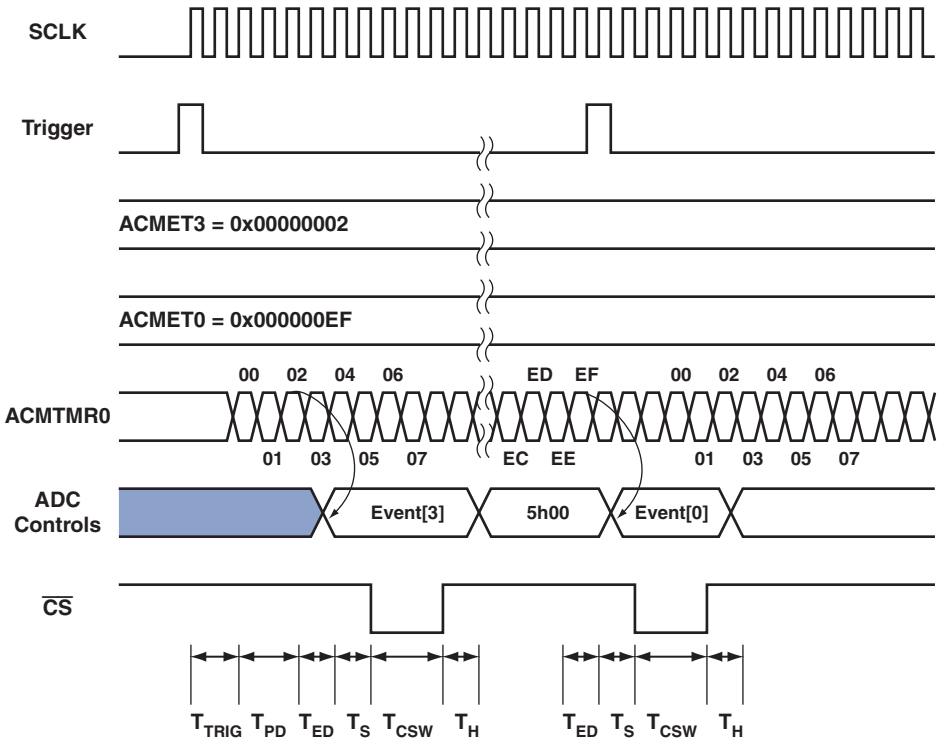


Figure 22-8. Trigger-to-Event Latency

In Figure 22-8, observe the following timing definitions:

- T_{TRIG} = trigger to timer start delay (3 to 4 SCLK)
- T_{ED} = internal event delay (1 SCLK)
- T_S - set up time
- T_{CSW} = $\overline{CS}$ width
- T_H = hold time

ACM External Pin Timing

All ADC controls (ACM_A[2:0], ACM_SGLDIFF, and ACM_RANGE) and $\overline{CS}$ are driven on the rising edge of SCLK. As a result, these signals are not synchronous to ACLK. The setup, hold, and other timing parameters of the ADC controls, $\overline{CS}$, and the frequency of ACLK can be configured in the ACM timing configuration registers (ACM_TCx). The polarity of $\overline{CS}$ and ACLK can be configured in the ACM control register (ACM_CTL). The timing parameters of the ADC controls (ACM_A[2:0], ACM_RANGE, and ACM_SGLDIFF) cannot be individually specified.

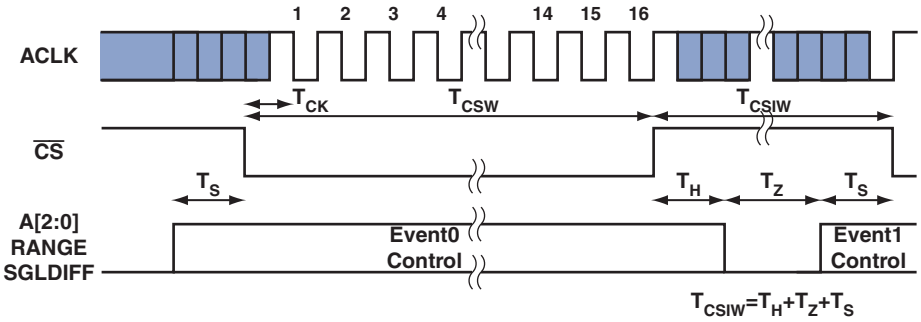


Figure 22-9. External Pin Timings

The inactive period of $\overline{CS}$ (T_{CSIW} as shown in the Figure 22-9) is the sum of the three timing parameters – Setup Time (T_S), Zero Time (T_Z) and the Hold Time (T_H):

$T_{CSIW} = T_S + T_Z + T_H$.

Appropriate specification of the values of those three parameters can yield the desired inactive period of $\overline{CS}$.

In order to provide a predictable latency from the occurrence of an internal event to the assertion of an external ADC sampling event, the ADC controls and $\overline{CS}$ must be driven on the rising edge of SCLK. Therefore, the Setup Time (T_S) of these signals is specified in terms of SCLK.

To achieve accurate timing relationship between $\overline{CS}$ and ACLK (which is a free running clock), the ACLK signal is re-aligned with the active edge of $\overline{CS}$. This realignment of ACLK ensures that the setup time of the first active edge of ACLK, with respect to the active edge of $\overline{CS}$, is at least 1 ACLK cycle.

The figures in the following sections:

- "Case 1—Chip Select Asserted During the High Phase of ACLK" on page 22-22
- "Case 2—Chip Select Asserted During the Low Phase of ACLK" on page 22-23
- "Case 3—Chip Select Asserted Right Before the Falling Edge of ACLK" on page 22-24
- "Case 4—Chip Select Asserted Right Before the Rising Edge of ACLK" on page 22-25
- "Case 5—ACLK Polarity Set to 1 (CLKPOL=1)" on page 22-26

show various scenarios of ACLK re-alignment. All of these figures assume an ACLK:SCLK ratio of 1:4.

The figures show both the ACM-generated $\overline{CS}$ signal, which is output externally onto the appropriate SPORT Receive Frame Sync (RFSx) pin, and the SPORTx_RFS signal, which is an internal signal that is routed to the receive frame sync input of the appropriate SPORT.

Case 1—Chip Select Asserted During the High Phase of ACLK

Figure 22-10 shows the realignment of ACLK when $\overline{CS}$ is asserted during the high phase of ACLK. The first edge of ACLK after the assertion of $\overline{CS}$ is the falling edge.

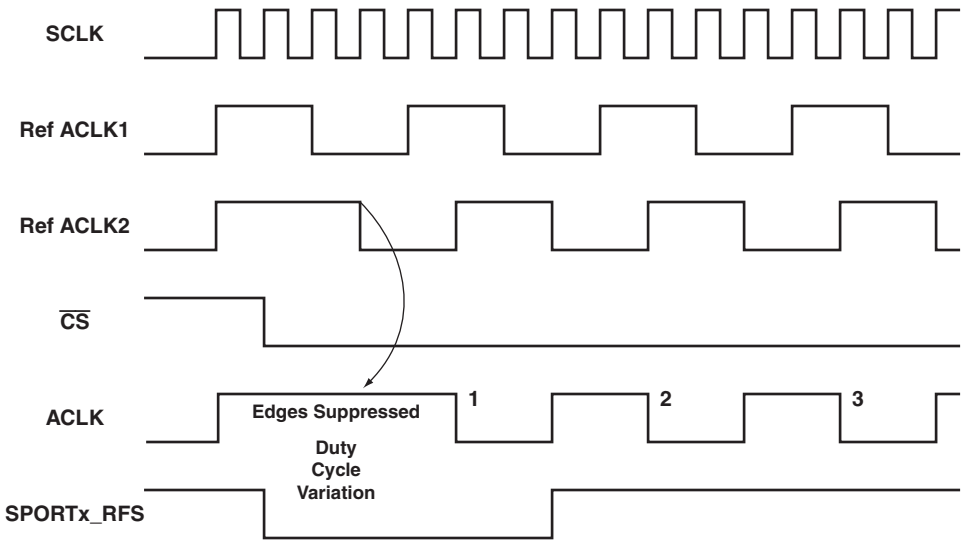


Figure 22-10. ACLK Adjustment for the Case of $\overline{CS}$ Assertion During the High Phase of ACLK

The two reference clock signals (Ref ACLK1 and Ref ACLK2) are shown to illustrate how the ACLK signal can be generated from a free running clock (Ref ACLK1) in order to meet the timing requirements between ACLK and $\overline{CS}$. Ref ACLK2 is based on the free running clock Ref ACLK1, but is adjusted such that its period is immediately reset upon the assertion

of $\overline{CS}$. The resulting ACLK signal, shown in Figure 22-10, is such that the time from the active edge of $\overline{CS}$ to the falling edge of ACLK is constant at a period of 1 ACLK cycle.

Case 2—Chip Select Asserted During the Low Phase of ACLK

When $\overline{CS}$ is asserted during the low phase of ACLK, as shown in Figure 22-11, ACLK is immediately pulled high causing a duty cycle variation. It is important to ensure that systems interfacing with the ACM can tolerate such duty cycle variation. In this case, similar to case 1, the time from the active edge of $\overline{CS}$ to the falling edge of ACLK is 1 ACLK period.

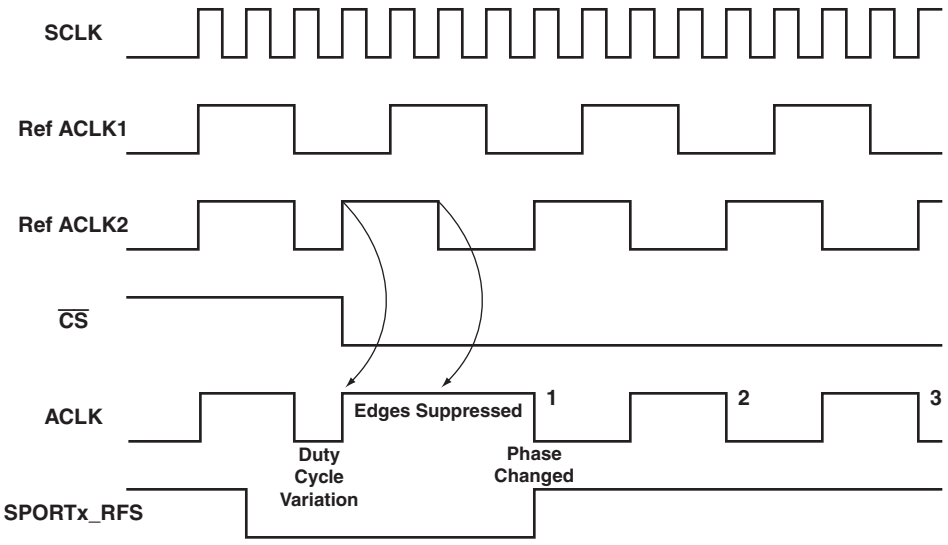


Figure 22-11. ACLK Adjustment for the Case of $\overline{CS}$ Assertion During the Low Phase of ACLK

Case 3—Chip Select Asserted Right Before the Falling Edge of ACLK

When $\overline{CS}$ is asserted right before the falling edge of ACLK, the falling edge of ACLK is suppressed, as shown in Figure 22-12. This ensures that the time from the active edge of $\overline{CS}$ to the falling edge of ACLK is constant at a period of 1 ACLK cycle. Notice that this suppression of ACLK falling edge leads to duty cycle variation. It is important to ensure that systems interfacing with the ACM can tolerate such duty cycle variation.

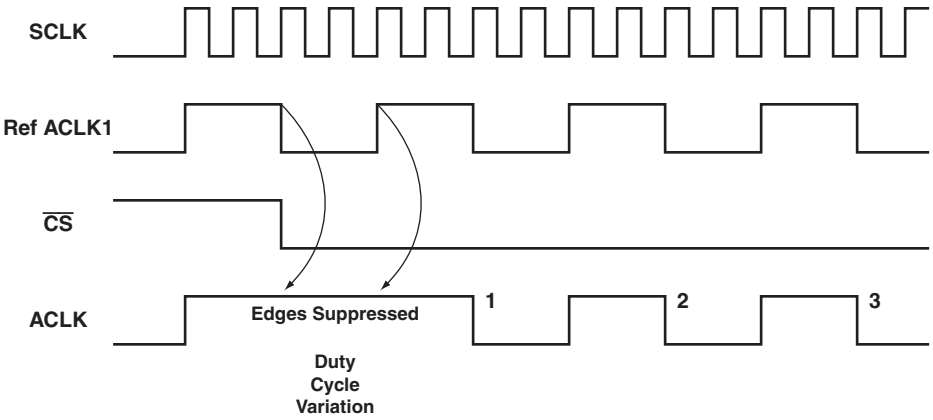


Figure 22-12. ACLK Adjustment for the Case of $\overline{CS}$ Assertion Right Before the Falling Edge of ACLK (CLKPOL =0)

Case 4—Chip Select Asserted Right Before the Rising Edge of ACLK

When $\overline{CS}$ is asserted right before the rising edge of ACLK, the high phase of ACLK is extended, as shown in Figure 22-13.

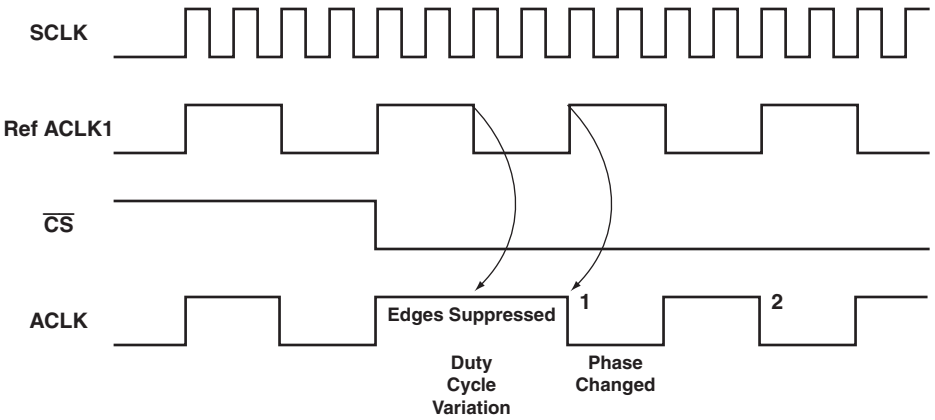


Figure 22-13. ACLK Adjustment for the Case of $\overline{CS}$ Assertion Right Before the Rising Edge of ACLK (CLKPOL =0)

This extension ensures that the time from the active edge of $\overline{CS}$ to the falling edge of ACLK is constant at a period of 1 ACLK cycle. Notice that this leads to duty cycle variation. It is important to ensure that systems interfacing with the ACM can tolerate such duty cycle variation.

Case 5—ACLK Polarity Set to 1 (CLKPOL=1)

When the ACLK polarity is set to 1 (bit CLKPOL is set to 1 in the ACM_CTL register), the first ACLK edge after the assertion of $\overline{CS}$ is the rising edge.

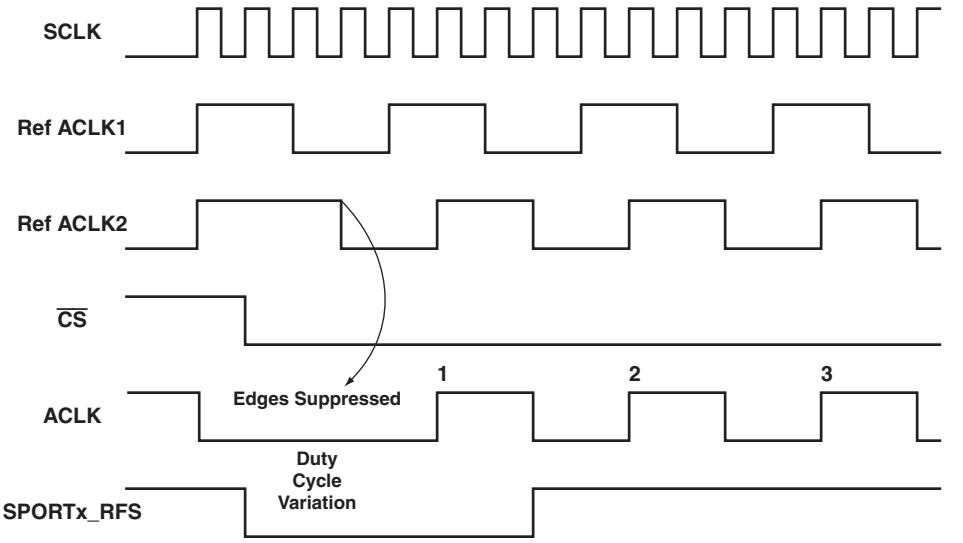


Figure 22-14. Showing ACLK With CLKPOL =1

The ACM ensures that the time from the active edge of $\overline{CS}$ to the rising edge of ACLK has a constant duration of 1 ACLK cycle. Figure 22-14 shows an example diagram of the case where CLKPOL=1.

ACM Timing Specifications

The AC timings of ACM signals are specified in the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*. When trigger sources external to the processor are used for triggering the ACM, (for example, external signals on the GPIO, timer, or PWM sync pins), the minimum pulsewidth for such trigger sources needs to be 1 SCLK period + 1 ns in order for the ACM trigger logic to detect the trigger.

When the ACM is used in conjunction with the SPORT, the setup and hold timing requirements for the SPORT data signals with respect to ACLK are different from those requirements with respect to internally-generated or externally-supplied SPORT clock. Consult the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for information on these timing requirements.

Programming Model

When the ACM is used in conjunction with SPORT0 or SPORT1, the ACM should be enabled before enabling the SPORT and should be disabled after disabling the SPORT.

Before enabling the ACM by writing 1 into the ACMEN bit in the ACM_CTL register, all other ACM_CTL bits should be programmed.

Modifying the ACMTMRxEN bits in the ACM_CTL register is not recommended while the ACM is enabled. Doing so can cause events to change dependency from one timer to the other and can cause the values in the ACM status registers (ACM_ES and ACM_STAT) to be inaccurate.

This means that both timers (ACMTMR0 and ACMTMR1) should be enabled (or disabled) together by programming the corresponding ACMTMRxEN bits while the ACM is disabled. If one of the timers is already running and the user wishes to enable the other timer, the ACM should be disabled first before enabling the timers and re-enabling the ACM.

Because the Event Completed bit (ECOM) in the ACM_STAT register indicates that all the events associated with a trigger are completed, this bit may not reflect the completion of all ACM events in the case where both timers ACMTMR0 and ACMTMR1 are enabled and each is triggered by a different trigger source. For example, if both ACMTMR0 and ACMTMR1 are enabled, if the trigger input associated with ACMTMR0 occurs first, and if all the events associated with ACMTMR0 are completed before the trigger input associated with ACMTMR1 is received, the ECOM bit will be set even when none of

the events associated with ACMTMR1 were processed. When the trigger associated with ACMTMR1 occurs, ECOM is cleared. When all the events associated with ACMTMR1 are processed, ECOM is set to 1 again. Therefore, if indication of the completion of all ACM events—in the case where both timers ACMTMR0 and ACMTMR1 are enabled and each is triggered by a different trigger source—is needed, the Event Status (ACM_ES) register may be used instead of, or in conjunction with, the ECOM bit in ACM_STAT.

When the ACM is used in conjunction with SPORT0 or SPORT1, ACLK supplies the SPORT receive clock signal. Because ACLK is an external clock relative to the SPORT peripheral, any SPORT requirements around a minimum number of stable external clock cycles before assertion of the first SPORT frame sync need to observed.

The SPORT requires a minimum of 3 clock cycles before it is able to recognize a frame sync. Therefore, when the ACM is used in conjunction with the SPORT, 3ACLK cycles should elapse before first assertion of $\overline{\text{CS}}$. This can be guaranteed by any of the following methods:

- Ensuring that ACM triggers are generated at least 3 ACLK cycles after the ACM is enabled.
- Ensuring that the event time value (ACM_ETx) of the first active event is such that 3 ACLK cycles would elapse before the event is processed.

When the minimum number of ACLK cycles before the assertion of $\overline{\text{CS}}$ is not observed, the SPORT may miss the data of the first ADC sampling event.

ACM Registers

The ADC controller module has a number of memory-mapped registers (MMRs) that regulate its operation. These registers are ACM control register (ACM_CTL), ACM timing configuration registers (ACM_TCx), ACM

status register (ACM_STAT), ACM event status register (ACM_ES), ACM interrupt mask register (ACM_IMSK), ACM missed event status register (ACM_MS), ACM event missed interrupt mask register (ACM_EMSK), ACM event control registers (ACM_ERx), and ACM event time registers (ACM_ETx).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

ACM Control (ACM_CTL) Register

The ACM_CTL register enables and selects the various modes of operation of the ACM.

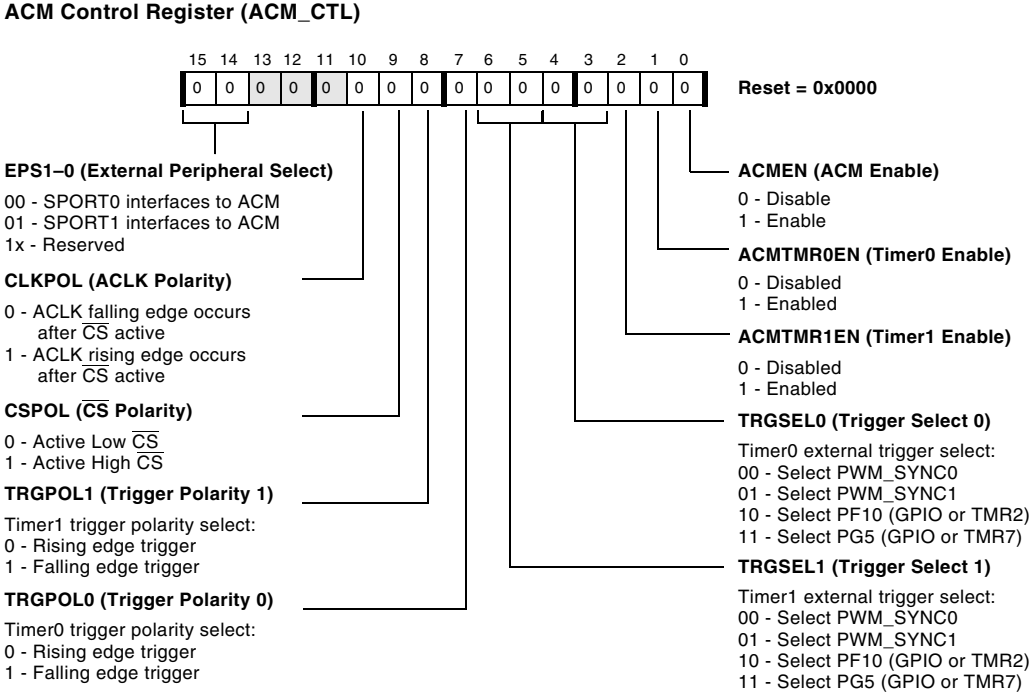


Figure 22-15. ACM Control (ACM_CTL) Register

ACM Status (ACM_STAT) Register

The ACM_STAT register indicates which event is currently being serviced, any pending events, any missed events, and any missed triggers.

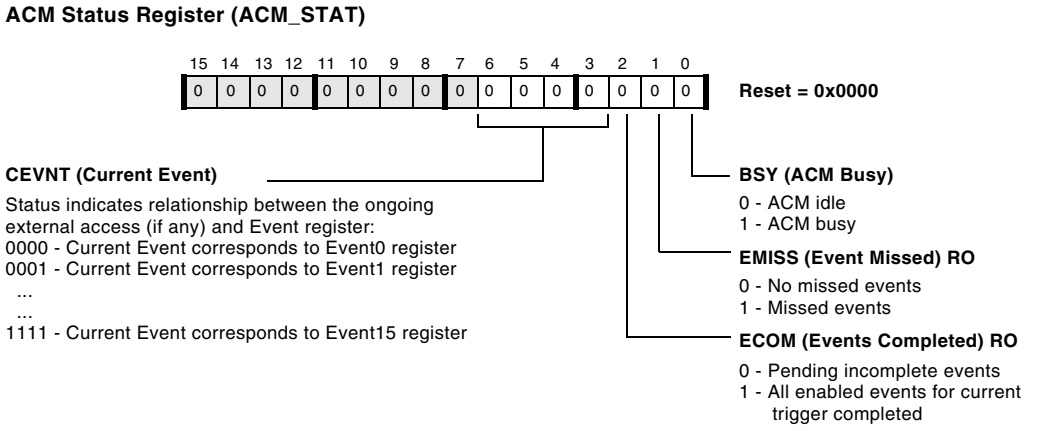


Figure 22-16. ACM Status (ACM_STAT) Register

ACM Busy (BSY=1)

ACM Busy status indicates that an external sampling event is in progress (for example, $\overline{CS}$ is active or about to go active)

Event(s) Missed (EMISS=1)

Event(s) Missed status indicates that one or more bits in the ACM_MS register are set. The EMISS bit has to be cleared by writing into the ACM_MS register.

Events Completed (ECOM=1)

Events Completed status indicates that all enabled events are completed for the current trigger. The ECOM bit gets cleared with each trigger. If both the timers are enabled, ECOM is set only after completion of all events for both.

ACM Event Status (ACM_ES) Register

The ACM Event Status register identifies which enabled event has occurred for a particular trigger cycle. When an ESx bit is cleared (=0), this status indicates that the ACM has not begun or completed conversion for Event x (conversion not done). When an ESx bit is set (=1), this status indicates that the ACM has completed conversion for Event x (conversion done).

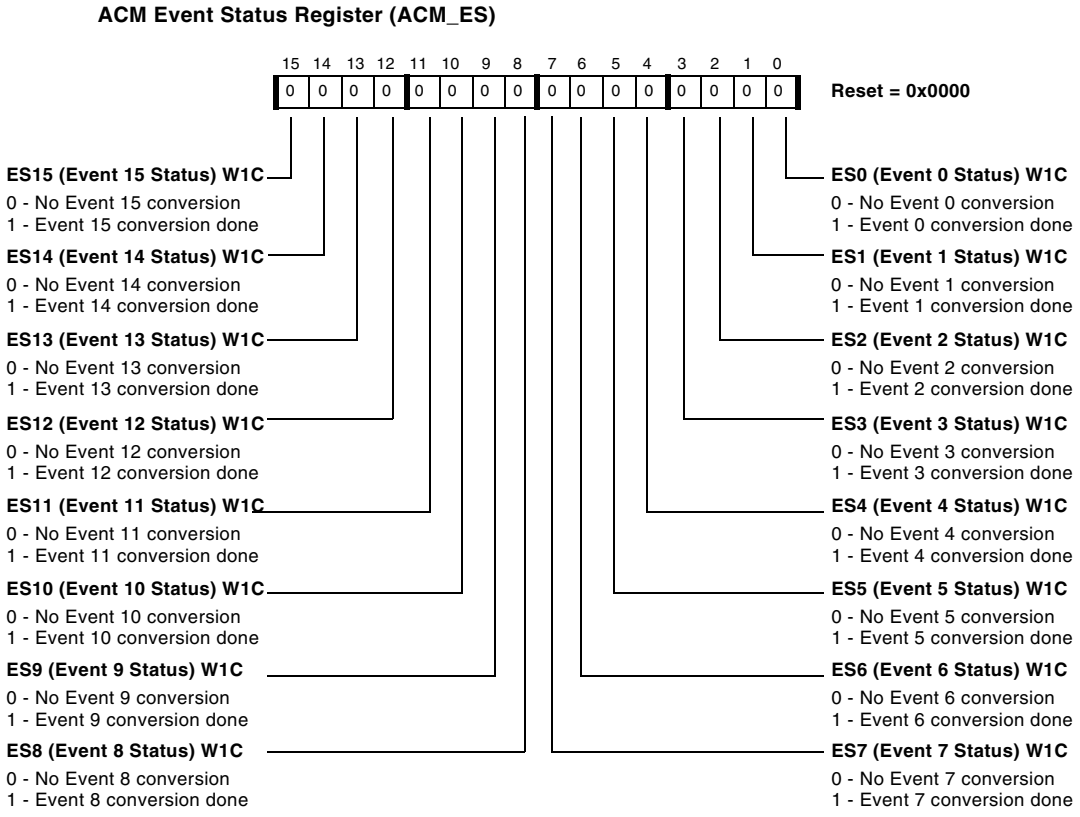


Figure 22-17. ACM Event Status (ACM_ES) Register

ACM Event Interrupt Mask (ACM_IMSK) Register

The ACM Interrupt Mask register selectively enables the interrupts associated with an event completion. When an IEx bit is set (=1), an interrupt is generated whenever Event x status (ESx in the ACM_ES register) gets set (for example, interrupt on Event x completion).

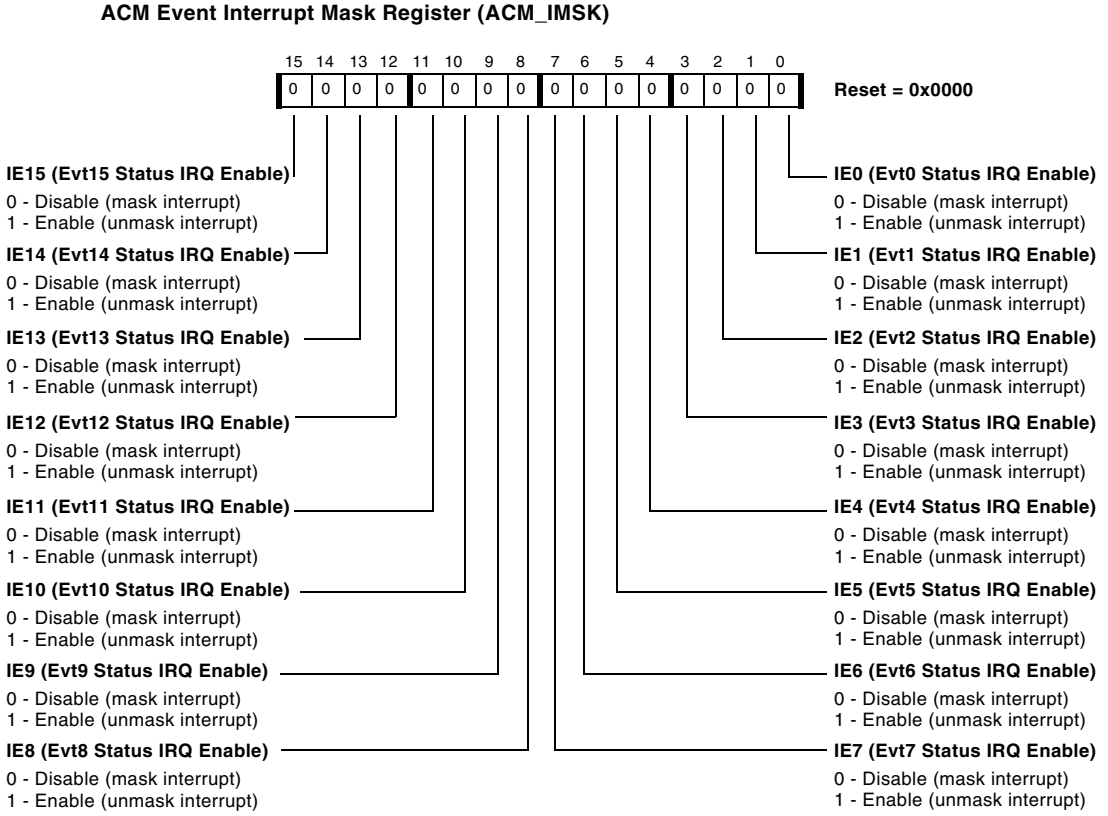


Figure 22-18. ACM Event Interrupt Mask (ACM_IMSK) Register

ACM Missed Event Status (ACM_MS) Register

The ACM Missed Event Status register indicates which enabled event has been missed for a particular trigger cycle. When an EMx bit is set (=1), this status indicates that event x was missed. This status generates an interrupt if the corresponding bit in the ACM_EMSK register is set.

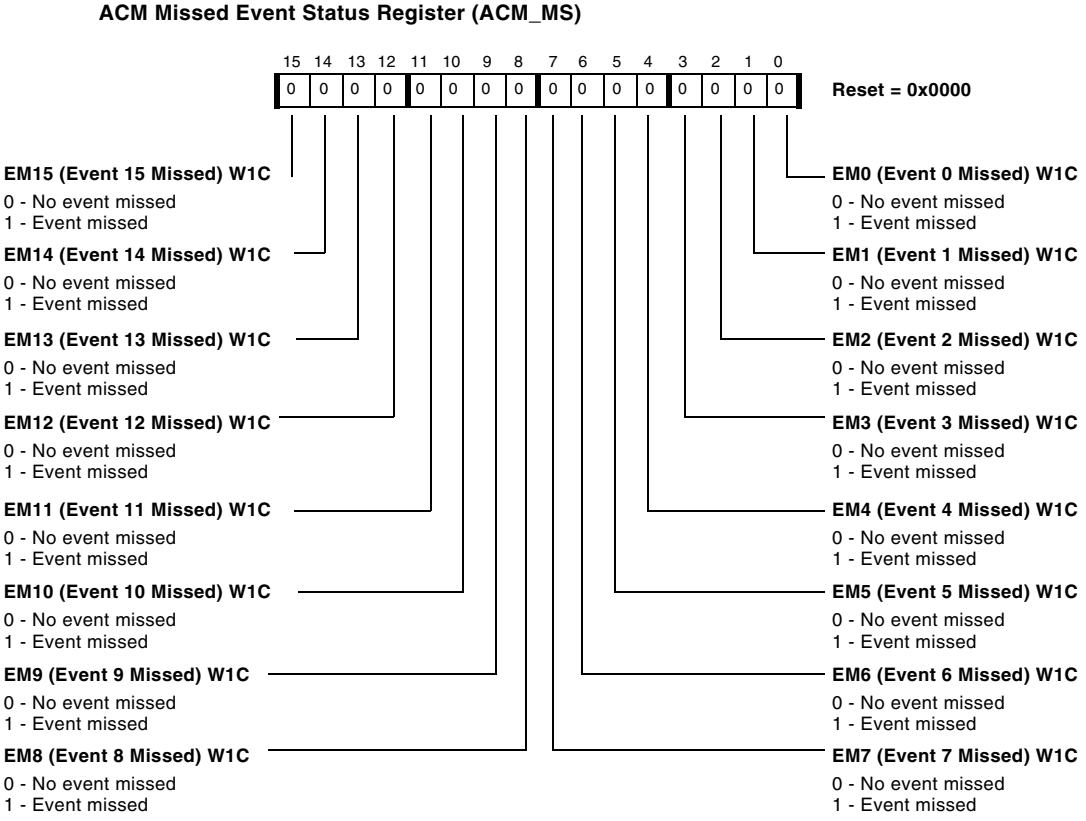


Figure 22-19. ACM Missed Event Status (ACM_MS) Register

ACM Event Missed Interrupt Mask (ACM_EMSK) Register

The ACM Event Missed Interrupt Mask register selectively enables the interrupts associated with an event being missed. When an MIEx bit is set (=1), an interrupt is generated whenever Event x is missed (EMx in the ACM_MS register is set); for example, interrupt on Event x miss.

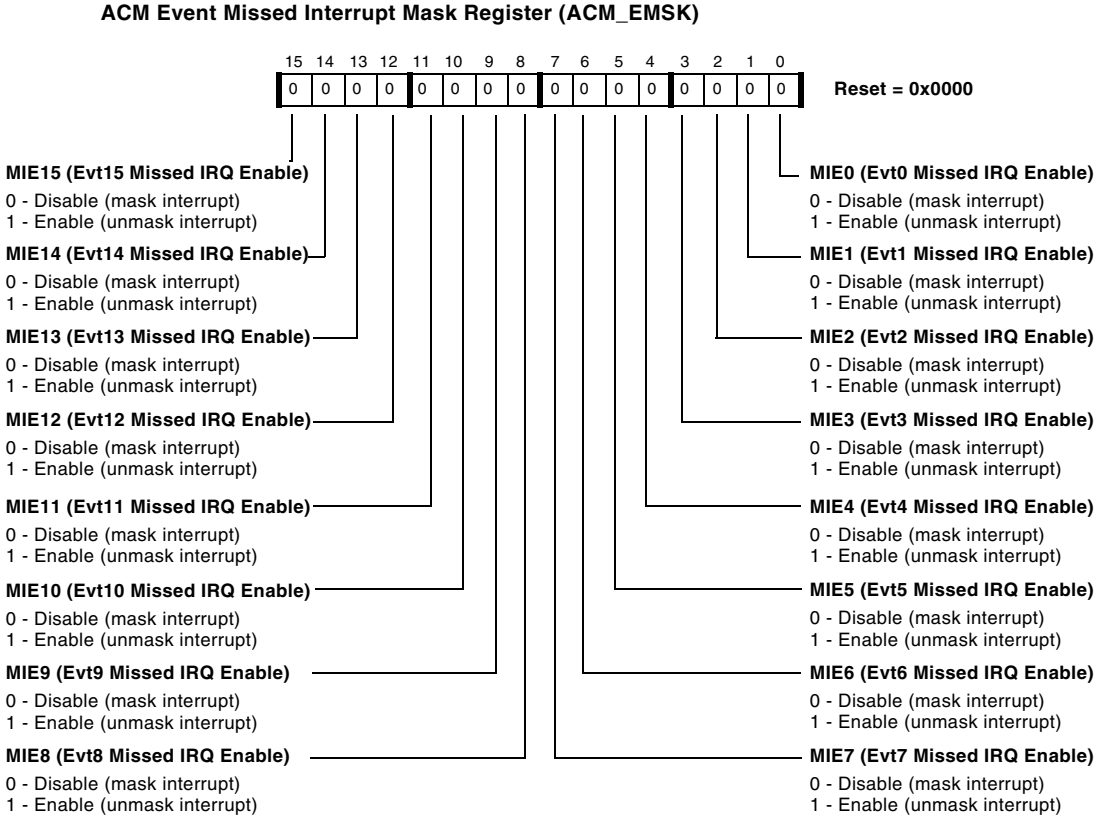


Figure 22-20. ACM Event Missed Interrupt Mask (ACM_EMSK) Register

ACM Event Control (ACM_ERx) Registers

The ACM Event Control registers hold the ADC Control value corresponding to the event. They also have enable bits to selectively enable a particular event.

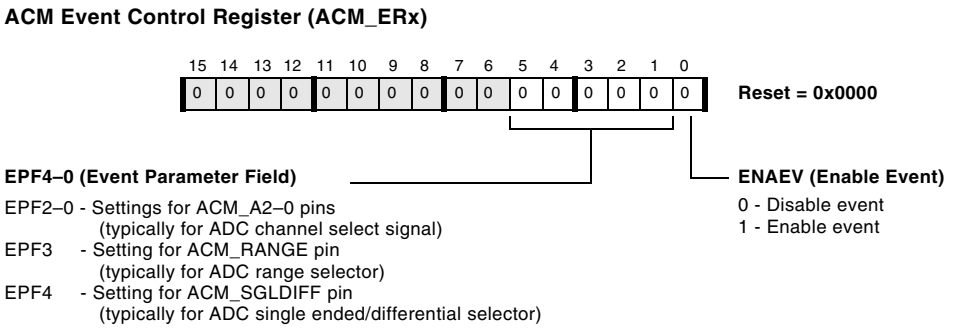


Figure 22-21. ACM Event Control (ACM_ERx) Registers

Enable Event (ENAEV=1)

When the ENEAV bit is set (=1), an event (time comparison match or other external trigger) causes a sampling event to occur to the ADC with the ADC controls selected by the EPF4–0 bit field

Event Parameter Field (EPF4–0)

The EPF4-0 bit field selects values for the ADC control pins, which are output when the enabled event occurs. Selection of EPFx values are based on the type of ADC, usage mode, and other items.

To prevent incorrect results, the ACM_ER register should not be programmed when an event is active. Program the ACM_ER before providing a trigger, and re-program it after all the events complete (ECOM bit in the ACM_STAT register is set).

ACM Event Time (ACM_ETx) Registers

The ACM_ETx registers hold the 32-bit time value (ETIME bits) for each event. There are 16 event time registers: 8 are assigned to each ACM timer, if both timers are enabled. If only one timer is enabled, all 16 ACM_ETx registers are assigned to the enabled timer.

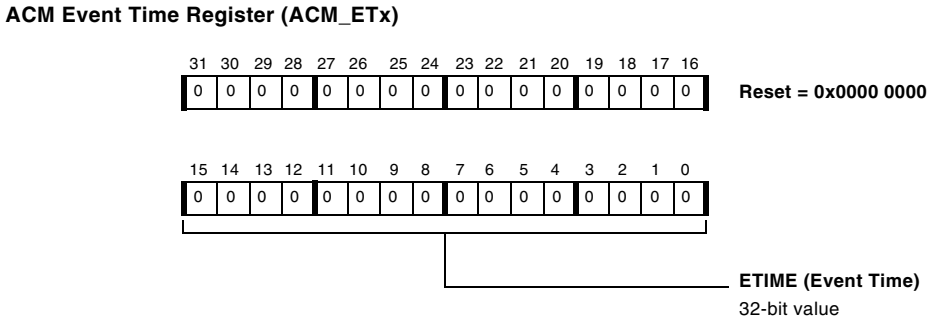


Figure 22-22. ACM Event Control (ACM_ERx) Registers

To prevent incorrect results, ACM_ET should not be programmed when an event is active. Program the ACM_ET before providing a trigger and re-program it after all the events are complete (ECOM bit in the ACM_STAT register is set).

ACM Timing Configuration (ACM_TCx) Registers

The ACM has two External Timing Configuration registers - Timing Configuration 0 (ACM_TC0) and Timing Configuration 1 (ACM_TC1). For information relating to signal timing and operation of the ACM_TCx registers, see "ACM External Pin Timing" on page 22-20. For timing specifications, see the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

ACM Timing Configuration 0 (ACM_TC0) Register

The ACM_TC0 register determines the frequency of ACLK (using the CKDIV field) and the setup cycles (using the SC field) for the ADC controls. Note that the setup cycles are specified in terms of SCLK.

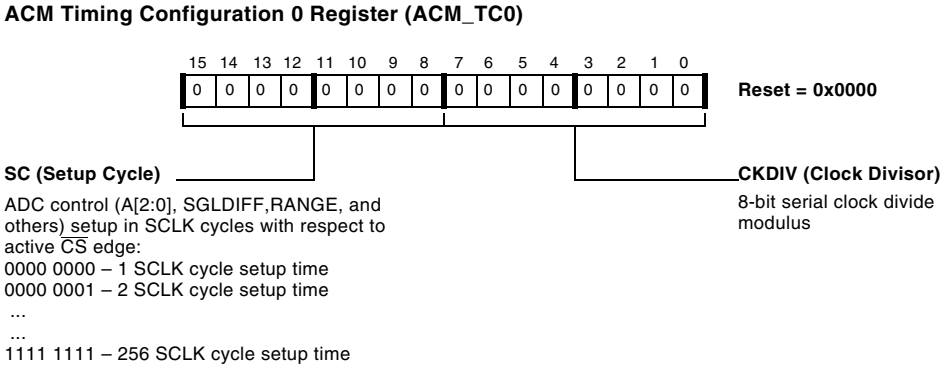


Figure 22-23. ACM Timing Configuration 0 (ACM_TC0) Register

The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the CKDIV field as follows:

ACLK frequency = (SCLK frequency)/(2 x (CKDIV + 1))

The maximum ACLK frequency is SCLK/2, and the minimum ACLK frequency is SCLK/512. So, for a 100 MHz SCLK, the ACLK range is from 195 KHz to 50 MHz.

ACM Timing Configuration 1 (ACM_TC1) Register

The ACM_TC1 register provides programmability for the active duration of Chip Select (T_{CSW}), Hold Cycles (T_H), and Zero Cycles (T_Z) for ADC controls.

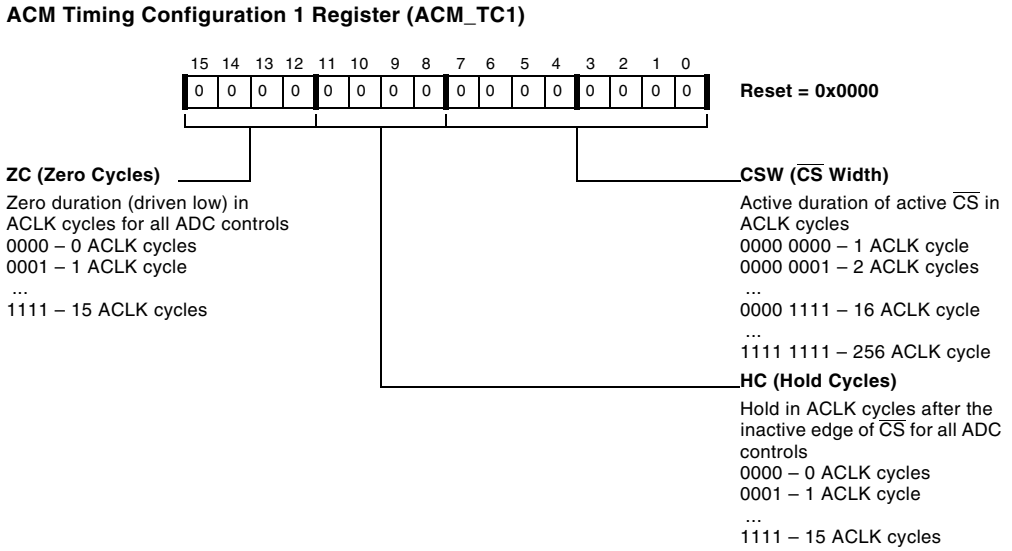


Figure 22-24. ACM Timing Configuration 1 (ACM_TC1) Register

Note that all timings specified in ACM_TC1 register are in terms of ACLK. For more information, see the T_{CSW}, T_H, and T_Z parameters shown in Figure 22-9.

Programming Examples

Listing 22-1 shows how to use the ACM in conjunction with a SPORT.

Listing 22-1. Using the ACM in Conjunction with a SPORT

```
/***********************************************
 ** Example pseudo code showing the process for **
 ** using the ACM in conjunction with a SPORT   **
 ***********************************************/

/* SPORT1 Selected
   ACLK falling edge occurs after CS
   Active low CS
   Trigger 1 polarity set to rising edge
   Trigger 0 polarity set to rising edge
   Trigger 1 select set to PWM_SYNC1
   Trigger 0 select set to PWM_SYNC0
   ACMTMR1 enabled
   ACMTMR0 enabled
   ACM disabled */
write(ACM_CTL, 0x4026, 16bit);

/* Timing configurations 1 register
   CS width (Tcsw) = 'd10
   Hold cycles (Th) = 'd15
   Zero cycles (Tz) = 'd2 */
write(ACM_TC1, 0x2f0a, 16bit);

/* Timing configurations 0 register
   Clock divisor (CKDIV) = 'd1
   Setup cycles (Ts) = 'd0 */
write(ACM_TC0, 0x0001, 16bit);

/* Event 2 ETIME = 'h00000001 */
write(ACM_ET2, 0x00000001, 32bit);

/* ACM signal settings for event 2:
```

```
   ACM_A[2:0] = 'h6
   ACM_RANGE = 1
   ACM_SGLDIFFS = 0
   Event 2 enabled */
write(ACM_ER2, 0x1d, 16bit);

/* Event 5 ETIME = 'h00000022 */
write(ACM_ET5, 0x00000022, 32bit);

/* ACM signal settings for event 5:
   ACM_A[2:0] = 'h7
   ACM_RANGE = 1
   ACM_SGLDIFFS = 1
   Event 5 enabled */
write(ACM_ER5, 0x3f, 16bit);

/* Event 14 ETIME = 'h00000001 */
write(ACM_ET14, 0x00000001, 32bit);

/* ACM signal settings for event 14:
   ACM_A[2:0] = 'h5
   ACM_RANGE = 1
   ACM_SGLDIFFS = 1
   Event 14 enabled */
write(ACM_ER14, 0x3b, 16bit);

/* All ACM event misses generate an interrupt */
write(ACM_EMSK, 0xffff, 16bit);

/* All ACM events generate interrupts */
write(ACM_IMSK, 0xffff, 16bit);

/* Enable the ACM before enabling the SPORT */
write(ACM_CTL, 0x4027, 16bit);
```

```
/* Configure and enable SPORT1 */
configure_enable_sport1();

/* Now setup a trigger to initiate sampling */
/* ensure that any SPORT requirements around a
   minimum number of stable external clock cycles
   before assertion of the first SPORT frame sync
   are observed (refer to "Programming Model" */
set_trig();

/* Wait for all events to complete */
wait();

/* Disable the SPORT */
disable_sport1();

/* Finally, disable the ACM */
write(ACM_CTL, 0x0000, 16bit);
```

23 ANALOG/DIGITAL CONVERTER (ADC)

The internal analog-to-digital converter (ADC) module on the ADSP-BF50x processor may be managed using the ADC control module (ACM). For more information on the ACM interface (which synchronizes the controls between the processor and the ADC), see "ADC Control Module (ACM)" on page 22-1.

This chapter provides information derived from the theory and operation sections of the AD7266 data sheet. For ADC specification *and* system design-for-performance information, see the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet*.

ADC Architecture

The ADSP-BF506F processor includes an ADC. All internal ADC signals are connected out to package pins to enable maximum flexibility in mixed signal applications.

The internal ADC is a dual, 12-bit, high speed, low power, successive approximation ADC that operates from a single 2.7 V to 5.25 V power supply and features throughput rates up to 2 MSPS. The device contains two ADCs, each preceded by a 3-channel multiplexer, and a low noise, wide bandwidth track-and-hold amplifier that can handle input frequencies in excess of 30 MHz.

Figure 23-1 shows the functional block diagram of the internal ADC.

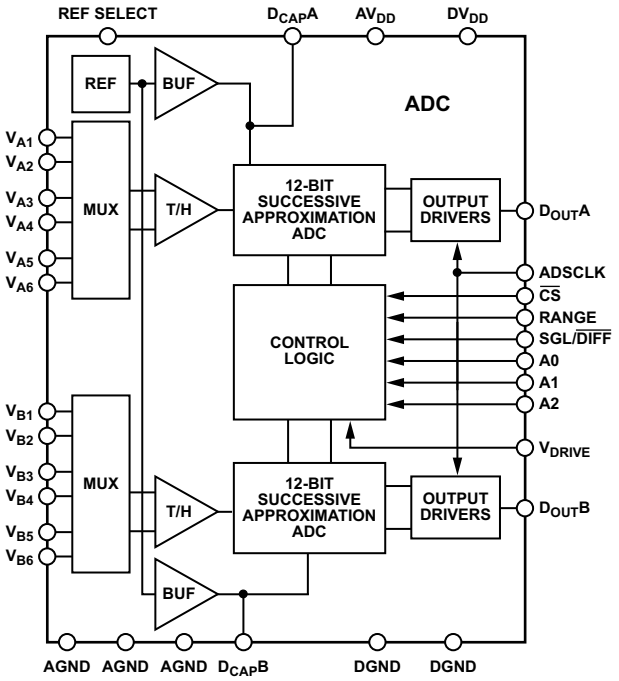


Figure 23-1. ADC (Internal) Functional Block Diagram

The ADC features include:

- Dual 12-bit, 3-channel ADC
- Throughput rate: up to 2 MSPS
- Specified for V_{DD} of 2.7 V to 5.25 V

- Pin-configurable analog inputs
    - 12-channel single-ended inputs
    - 6-channel fully differential inputs
    - 6-channel pseudo differential inputs
- Accurate on-chip reference: 2.5 V
- Dual conversion with read 437.5 ns, 32 MHz ADSCLK
- High speed serial interface
    - SPI®-, QSPI™-, MICROWIRE™-, and DSP-compatible
- Low power shutdown mode

The conversion process and data acquisition use standard control inputs allowing easy interfacing to microprocessors or DSPs. The input signal is sampled on the falling edge of $\overline{CS}$; conversion is also initiated at this point. The conversion time is determined by the ADSCLK frequency. There are no pipelined delays associated with the part.

The internal ADC uses advanced design techniques to achieve very low power dissipation at high throughput rates. The part also offers flexible power/ throughput rate management when operating in normal mode as the quiescent current consumption is so low.

The analog input range for the part can be selected to be a 0 V to V_{REF} (or $2 \times V_{REF}$) range, with either straight binary or twos complement output coding. The internal ADC has an on-chip 2.5 V reference that can be overdriven when an external reference is preferred.

Additional highlights of the internal ADC include:

- Two Complete ADC Functions Allow Simultaneous Sampling and Conversion of Two Channels — Each ADC has three fully/pseudo differential pairs, or six single-ended channels, as programmed.

The conversion result of both channels is simultaneously available on separate data lines, or in succession on one data line if only one serial port pin is available.

- High Throughput with Low Power Consumption
- The internal ADC offers both a standard 0 V to V_{REF} input range and a 2 × V_{REF} input range.
- No Pipeline Delay — The part features two standard successive approximation ADCs with accurate control of the sampling instant via a $\overline{CS}$ input and once off conversion control.

Maximum ADC Sampling Rate

When the ADC is connected to the serial port of the processor, the maximum sampling rate achievable depends on the timing specifications of both the ADC and the processor peripherals involved in the connectivity. The following sections describe two commonly used interface options that can support maximum sampling rates.

Interfacing the ADC With the ACM and the SPORT

As shown in Figure 22-1 on page 22-2 (“ADSP-BF504F – ACM Connections (for External ADC)”) and in Figure 22-2 on page 22-3 (“ADSP-BF506F – ACM Connections (for Internal ADC)”), the ACM generates the clock to drive the ADC and the SPORT. The following timing specifications apply:

- DRxPRI/DRxSEC minimum setup before clock = T_{SDR}
- Data access time after ADSCLK falling edge = T_4

Ignoring any board delays, a data bit transfer takes:

- Transfer Time per bit (T_B) = T_{SDR} + T_4

The transfer time per sample takes:

- Transfer Time per sample (T_S) = ($T_B \times N_B$) + T_{QUIET}

Where:

- N_B is the minimum number of bits necessary to tranfer a sample
- T_{QUIET} is the minimum time between two consecutive samples

Therefore, the maximum theoretic sampling rate:

$$F_{MTSR} = \frac{1}{T_S}$$

See the *ADSP-BF504, ADSP-BF504F, ADSP-BF506F Embedded Processor Data Sheet* for the actual value of the parameters necessary for the above calculations.

In practice, various factors (such as board delays and maximum frequency ratings) can reduce the maximum achievable sampling rate. Fore example, assuming the following values:

- T_{SDR} = 7 ns
- T_4 = 27 ns
- T_{QUIET} = 30 ns
- N_B = 14 bits

and board delays of 2 ns, the maximum achievable sampling rate may be caculated as follows:

- T_S = (34 × 14) + 30

$$F_{MTSR} = \frac{1}{T_S}$$

Interfacing the ADC With the SPORT and With TMR Pins

As shown in Figure 23-2, the processor timer generates the clock to drive the ADC and the SPORT.

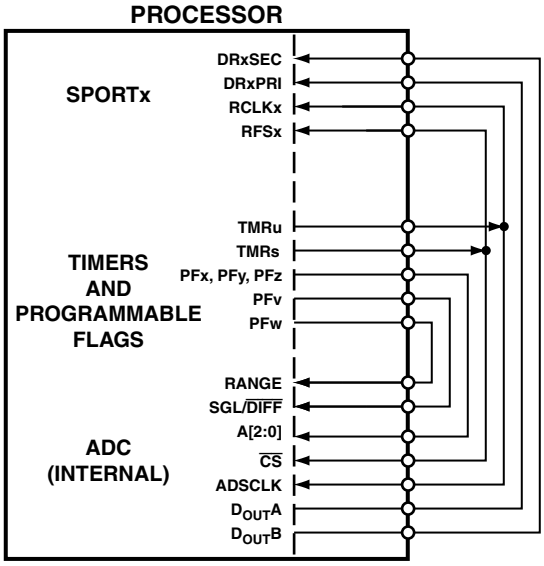


Figure 23-2. ADC, TMR, and SPORT Connections

For this system design, the following timing specifications apply:

- DRxPRI/DRxSEC minimum setup before external clock = T_{SDRE}
- Data access time after ADSCLK falling edge (V_{DD} = 5 V) = T_4

Assuming board delays of no more than 3 ns, the maximum sampling rate of 2 MSPS can be supported for V_{DD} = 5 V and ADSCLK = 31.25 MHz.

24 SYSTEM RESET AND BOOTING

This document contains material that is subject to change without notice. The content of the boot ROM as well as hardware behavior may change across silicon revisions. See the anomaly list for differences between silicon revisions. This document describes functionality of silicon revision 0.0 of the ADSP-BF50x processors.

Overview

When the $\overline{\texttt{RESET}}$ input signal releases, the processor starts fetching and executing instructions from the on-chip boot ROM at address 0xEF00 0000.

The internal boot ROM includes a small boot kernel that loads application data from an external memory or host device. The application data is expected to be available in a well-defined format called the boot stream. A boot stream consists of multiple blocks of data and special commands that instruct the boot kernel how to initialize on-chip L1 memories as well as off-chip volatile memories.

The boot kernel processes the boot stream block-by-block until it is instructed by a special command to terminate the procedure and jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

The processor features three dedicated input pins `BMODE[2:0]` that select the booting mode. The boot kernel evaluates the `BMODE` pins and performs

booting from respective sources. Table 24-1 describes the modes of the BMODE pins.

Table 24-1. Booting Modes

BMODE2-0	Boot Source	Description
000	No boot – idle	The processor does not boot. Rather, the boot kernel executes an IDLE instruction.
001	Boot from internal parallel flash[1] in asynchronous mode	In this mode, conservative timing parameters are used to communicate with the flash device. The boot kernel communicates with the flash device asynchronously.
010	Boot from internal parallel flash[1] in synchronous burst mode	In this mode, fast timing parameters are used to communicate with the flash device. The boot kernel configures the flash device for synchronous burst communication and boots from the flash synchronously.
011	Boot from external serial SPI memory	After an initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit or 32-bit addressable SPI flash or EEPROM memory that connects to $\overline{\text{SPI0\_SSEL1}}$.
100	Boot from SPI host	In this slave mode, the kernel expects the boot stream to be applied to SPI0 by an external host device.
101	Boot from PPI host	In this boot mode, the kernel expects data to be received over the 16-bit PPI port. Data transfers are controlled with the incoming PPI_FS1 signal. The processor uses the PPI_FS2 signal to indicate when it is ready to receive data and how much data is expected.
110	Reserved	Reserved
111	Boot from UART0 host	In this slave mode, the kernel expects the boot stream to be applied to UART0 by an external host device. Prior to providing the boot stream, the host device is expected to send a 0x40 (ASCII '@') character that is examined by the kernel to adjust the bit rate.

1 This mode is only available for products containing flash memory.

Reset and Power-up

Table 24-2 describes the six types of resets.

Note that each type resets the core except for the System Software reset.

Table 24-2. Resets

Reset	Source	Result
Hardware reset	The RESET pin causes a hardware reset.	Resets both the core and the peripherals, including the dynamic power management controller (DPMC). Resets bits [15:4] of the SYSCR register. For more information, see "System Reset Configuration (SYSCR) Register" on page 24-60.
Wakeup from hibernate state	Wake-up event as enabled in the VR_CTL register and reported by the PLL_STAT register.	Behaves as hardware reset except the WURESET bit in the SYSCR register is set. Booting can be performed conditionally on this event.
System software reset	Calling the bfrom_SysControl() routine with the SYSCTRL_SYSRESET option triggers a system reset.	Resets only the peripherals, excluding the RTC (real time clock) block and most of the DPMC. The system software reset clears bits [15:13] and bits [11:4] of the SYSCR register, but not the WURESET bit. The core is not reset and a boot sequence is not triggered. Sequencing continues at the instruction after bfrom_SysControl() returns.
Watchdog timer reset	Programming the watchdog timer causes a watchdog timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. (Because of the partial reset to the DPMC, the watchdog timer reset is not functional when the processor is in Sleep or Deep Sleep modes.) The SWRST or the SYSCR register can be read to determine whether the reset source was the watchdog timer.

Table 24-2. Resets (Cont'd)

Reset	Source	Result
Core double-fault reset	A core double fault occurs when an exception happens while the exception handler is executing. If the core enters a double-fault state, and the Core Double Fault Reset Enable bit (DOUBLE_FAULT) is set in the SWRST register, then a software reset will occur.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST or SYSCR registers can be read to determine whether the reset source was a core double-fault.
Software reset	This reset is caused by executing a RAISE 1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) through emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Program executions vector to the 0xEF00 0000 address. The boot code executes an immediate system reset to ensure system consistency.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted after a specified asserted hold time to perform a hardware reset. For more information, see the product data sheet.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the $\overline{\text{RESET}}$ pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the boot mode sequence configured by the state of the BMODE pins.

The BMODE pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DDEXT} or GND. The pins and the corresponding bits in the SYSCR register configure the boot mode that is employed after

hardware reset or system software reset. See the Blackfin Processor Programming Reference for further information.

Software Resets

A software reset may be initiated in three ways.

- By the watchdog timer, if appropriately configured
- Calling the `bfrom_SysControl()` API function residing in the on-chip ROM. For further information, see Chapter 8, "Dynamic Power Management".
- By the `RAISE 1` instruction

The watchdog timer resets both the core and the peripherals, as long as the processor is in Active or Full-On mode. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

In order to perform a system reset, the `bfrom_SysControl()` routine must be called while executing from L1 memory (either as cache or as SRAM). When L1 instruction memory is configured as cache, make sure the system reset sequence is read into the cache.

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

For a reset generated by formatting the watchdog timer, the processor transitions into the boot mode sequence. The boot mode is configured by the state of the `BMODE` bit field in the `SYSCR` register.

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

A software reset only affects the state of the core. The boot kernel immediately issues a system reset to keep consistency with the system domain.

Reset Vector

When reset releases, the processor starts fetching and executing instructions from address 0xEF00 0000. This is the address where the on-chip boot ROM resides.

On a hardware reset, the boot kernel initializes the EVT1 register to 0xFFA0 0000. When the booting process completes, the boot kernel jumps to the location provided by the EVT1 vector register. The content of the EVT1 register is overwritten by the TARGET ADDRESS field of the first block of the applied boot stream. If the BCODE field of the SYSCR register is set to 3 (no boot option), the EVT1 register is not modified by the boot kernel on software resets. Therefore, programs can control the reset vector for software resets through the EVT1 register. This process is illustrated by the flow chart in Figure 24-1.

The content of the EVT1 register may be undefined in emulator sessions.

Servicing Reset Interrupts

The processor services a reset event like other interrupts. The reset interrupt has top priority. Only emulation events have higher priority. When coming out of reset, the processor is in supervisor mode and has full access to all system resources. The boot kernel can be seen as part of the reset service routine. It runs at the top interrupt priority level.

Even when the boot process has finished and the boot kernel passes control to the user application, the processor is still in the reset interrupt. To enter user mode, the reset service routine must initialize the RETI register and terminate with an RTI instruction.

For a programming example, see “Example System Reset” on page 24-81.

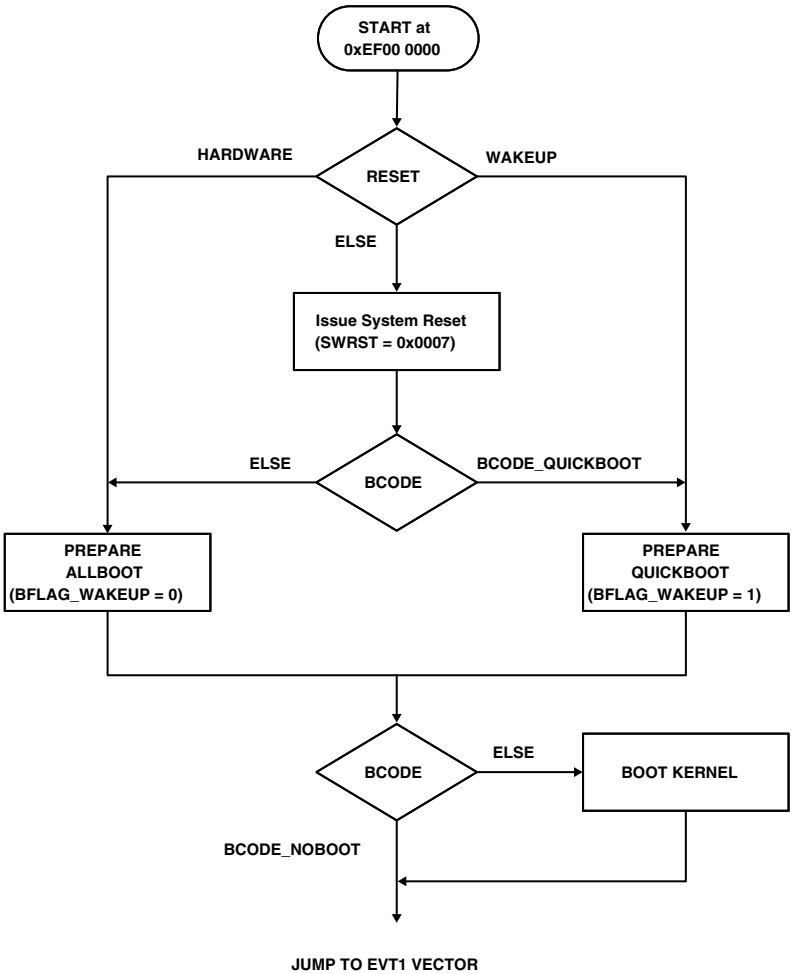


Figure 24-1. Global Boot Flow

Listing 24-1 and Listing 24-2 on page 24-82 show code examples that handle the reset event. See the *Blackfin Processor Programming Reference* for details on user and supervisor modes.

Systems that do not work in an operating system environment may not enter user mode. Typically, the interrupt level needs to be degraded down

to IVG15. Listing 24-3 and Listing 24-4 on page 24-83 show how this is accomplished.

Since the boot kernel is running at reset interrupt priority, NMI events, hardware errors and exceptions are not serviced at boot time. As soon as the reset service routine returns, the processor can service the events that occurred during the boot sequence. It is recommended that programs install NMI, hardware error, and exception handlers before leaving the reset service routine. This includes proper initialization of the respective event vector registers `EVTx`.

Basic Booting Process

After evaluating the `BMODE` pins, the boot kernel residing in the on-chip boot ROM starts processing the boot stream. The boot stream is either read from memory or received from a host processor. A boot stream represents the application data and is formatted in a special manner. The application data is segmented into multiple blocks of data. Each block begins with a block header. The header contains control words such as the destination address and data length information.

As Figure 24-2 illustrates, the VisualDSP++ tools suite features a loader utility (`elfloader.exe`). The loader utility parses the input executable file (`.DXE`), segments the application data into multiple blocks, and creates the header information for each block. The output is stored in a loader file (`.LDR`). The loader file contains the boot stream and is made available to hardware by programming or burning it into non-volatile external memory. Refer to the *VisualDSP++ Loader Manual* for information on switches for loader files.

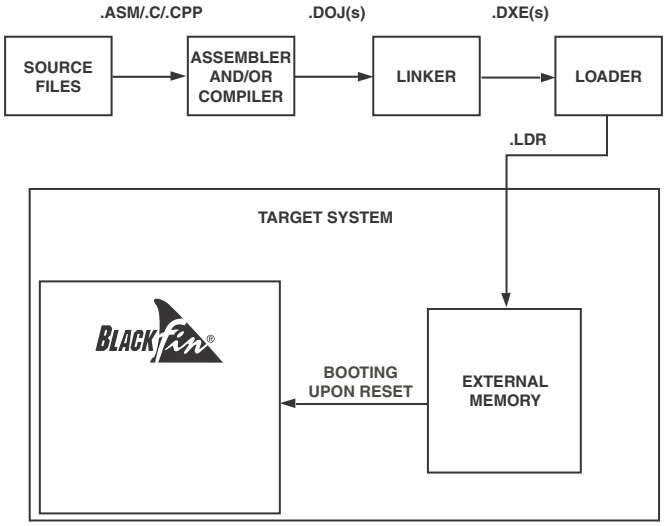


Figure 24-2. Project Flow for a Standalone System

Figure 24-3 shows the parallel or serial boot stream contained in a flash memory device. In host boot scenarios, the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin processor. After reset, the headers are read and parsed by the on-chip boot ROM, and processed block-by-block. Payload data is copied to destination addresses in on-chip L1 memory.

Booting into scratchpad memory (0xFFB0 0000–0xFFB0 0FFF) is not supported. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM. Similarly, booting into the upper 16 bytes of L1 data bank A (0xFF80 7FF0–0xFF80 7FFF by default) is not supported. These memory locations are used by the boot kernel for intermediate storage of block header information. These memory regions cannot be initialized at boot time. After booting, they can be used by the application during runtime.

When the BFLAG_INDIRECT flag for any block is set, the boot kernel uses another memory block in L1 data bank A (by default, 0xFF80 7F00–0xFF80 7FEF) for intermediate data storage. To avoid conflicts, the VisualDSP++ elfloader utility ensures this region is booted last.

The entire source code of the boot ROM is shipped with the VisualDSP++ tools installation. Refer to the source code for any additional questions not covered in this manual. Note that minor maintenance work may be done to the content of the boot ROM when silicon is updated.

Block Headers

A boot stream consists of multiple boot blocks, as shown in Figure 24-3. Every block is headed by a 16-byte block header. However, every block does not necessarily have a payload, as shown in Figure 24-4.

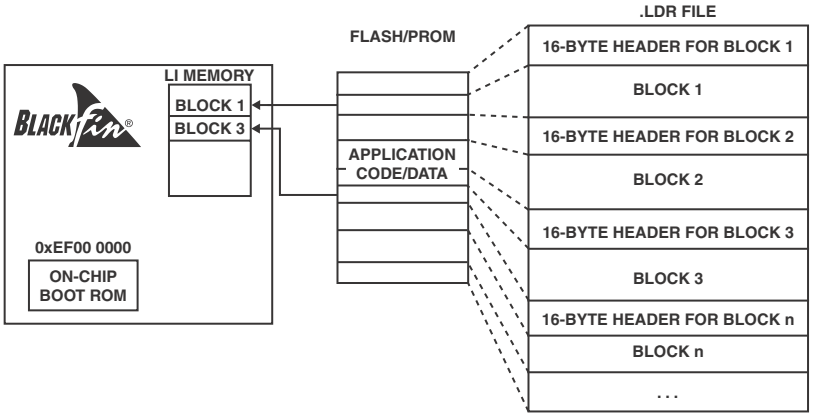


Figure 24-3. Booting Process

The 16 bytes of the block header are functionally grouped into four 32-bit words, the BLOCK CODE, the TARGET ADDRESS, the BYTE COUNT, and the ARGUMENT fields.

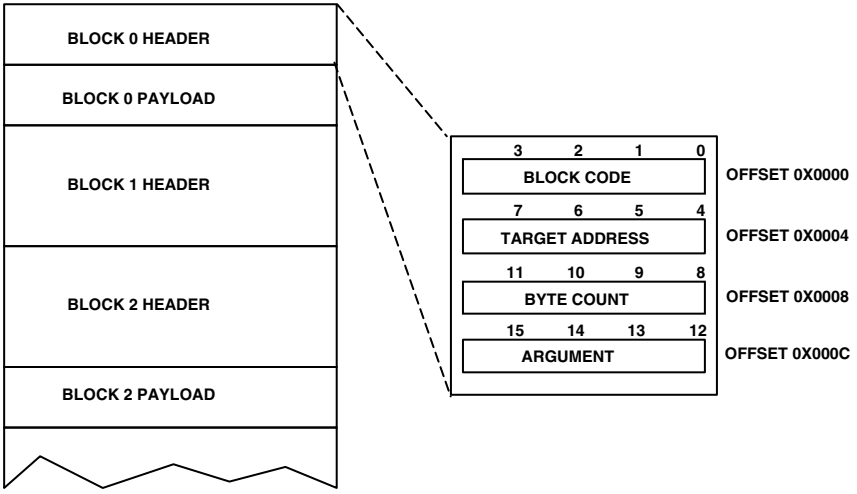


Figure 24-4. Boot Stream Headers

Block Code

The first 32-bit word is the BLOCK CODE. See Figure 24-5.

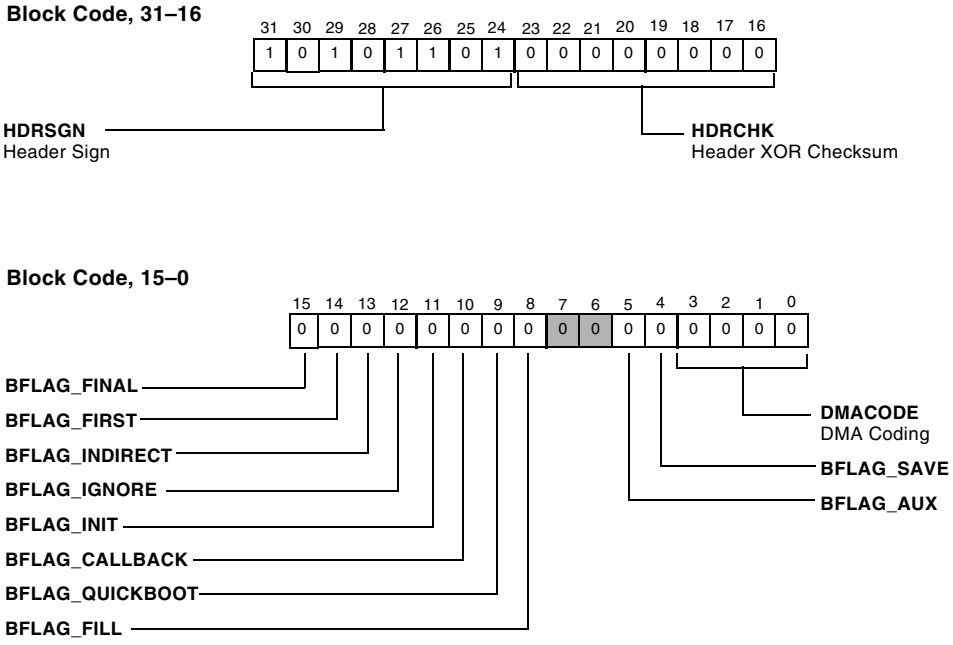


Figure 24-5. Block Code, 31–0

DMA Code Field

The DMA code (DMACODE) field instructs the boot kernel whether to use 8-bit, 16-bit or 32-bit DMA and how to program the source modifier of a memory DMA. Particularly in case of memory boot modes, this field is interrogated by the boot kernel to differentiate the 8-bit, 16-bit, and 32-bit cases.

The boot kernel tests this field only on the first block and ignores the field in further blocks (See Table 24-3).

Table 24-3. Bus and DMA Width Coding

DMA Code	DMA Width	Source DMA Modify	Application
0	reserved[1]		
1	8-bit	1	Default 8-bit boot from 8-bit source[2]
2	8-bit	2	Zero-padded 8-bit boot from 16-bit EBIU
3	8-bit	4	Zero-padded 8-bit boot from 32-bit EBIU[3]
4	8-bit	8	Zero-padded 8-bit boot from 64-bit EBIU[4]
5	8-bit	16	Zero-padded 8-bit boot from 128-bit EBIU[4]
6	16-bit	2	Default 16-bit boot from 16-bit source
7	16-bit	4	Zero-padded 16-bit boot from 32-bit EBIU[3]
8	16-bit	8	Zero-padded 16-bit boot from 64-bit EBIU[4]
9	16-bit	16	Zero-padded 16-bit boot from 128-bit EBIU[4]
10	32-bit	4	Default 32-bit boot from 32-bit source[3]
11	32-bit	8	Zero-padded 32-bit boot from 64-bit EBIU[4]
12	32-bit	16	Zero-padded 32-bit boot from 128-bit EBIU[4]
13	64-bit	8	Default 64-bit boot from 64-bit source[4]
14	64-bit	16	Zero-padded 64-bit boot from 128-bit EBIU[4]
15	128-bit	16	Default 128-bit boot from 128-bit source[4]

1 Reserved to differentiate from ADSP-BF53x boot streams.
2 Used by all byte-wise serial boot modes.
3 Applicable only to memory boot modes.
4 Not supported by ADSP-BF50x Blackfin products.

Block Flags Field

Table 24-4. Block Flags

Bit	Name	Description
4	BFLAG_SAVE	Saves the memory of this block to off-chip memory in case of power failure or a hibernate request. This flag is not used by the on-chip boot kernel.
5	BFLAG_AUX	Nests special block types as required by special purpose second-stage loaders. This flag is not used by the on-chip boot kernel.
6	Reserved	
7	Reserved	
8	BFLAG_FILL	Tells the boot kernel to not process any payload data. Instead the target memory (specified by the `TARGET ADDRESS` and `BYTE COUNT` fields) is filled with the 32-bit value provided by the `ARGUMENT` word. The fill operation is always performed by 32-bit DMA; therefore target address and byte count must be divisible by four.
9	BFLAG_QUICKBOOT	Processes the block for full boot only. Does not process this block for a quick boot (warm boot).
10	BFLAG_CALLBACK	Calls a subfunction that may reside in on-chip or off-chip ROM or is loaded by an initcode in advance. Often used with the `BFLAG_INDIRECT` switch. If `BFLAG_CALLBACK` is set for any block, an initcode must register the callback function first. The function is called when either the entire block is loaded or the intermediate storage memory is full. The callback function can do advanced processing such as CRC checksum.
11	BFLAG_INIT	This flag causes the boot kernel to issue a `CALL` instruction to the target address of the boot block after the entire block is loaded. The initcode should return by an `RTS` instruction. It may or may not be overwritten by application data later in the boot process. If the code is loaded earlier or resides in ROM, the init block can be zero sized (no payload).

Table 24-4. Block Flags (Cont'd)

Bit	Name	Description
12	BFLAG_IGNORE	Indicates a block that is not booted into memory. It instructs the boot kernel to skip the number of bytes of the boot stream as specified by BYTE COUNT. In master boot modes, the boot kernel simply modifies its source address pointer. In this case the BYTE COUNT value can be seen as a 32-bit two's-complement offset value to be added to the source address pointer. In slave boot modes, the boot kernel actively loads and changes the payload of the block. In slave modes the byte count must be a positive value.
13	BFLAG_INDIRECT	Boots to an intermediate storage place, allowing for calling an optional callback function, before booting to the destination. This flag is used when the boot source does not have DMA support and either the destination cannot be accessed by the core (L1 instruction SRAM) or cannot be efficiently accessed by the core. This flag is also used when CALLBACK requires access to data to calculate a checksum, or when performing tasks such as decryption or decompression.
14	BFLAG_FIRST	This flag, which is only set on the first block of a DXE, tells the boot kernel about the special nature of the TARGET ADDRESS and the ARGUMENT fields. The TARGET ADDRESS field holds the start address of the application. The ARGUMENT field holds the offset to the next DXE.
15	BFLAG_FINAL	This flag causes the boot kernel to pass control over to the application after the final block is processed. This flag is usually set on the last block of a DXE unless multiple DXEs are merged.

The BFLAG_FIRST flag must not be combined with the BFLAG_FILL flag. The BFLAG_FIRST flag may be combined with the BFLAG_IGNORE flag to deposit special user data at the top of the boot stream. Note the special importance of the VisualDSP++ elfloader -readall switch.

Header Checksum Field

The header checksum (HDRCHK) field holds a simple XOR checksum of the other 31 bytes in the boot block header. The boot kernel jumps to the error routine if the result of an XOR operation across all 32 header bytes (including the HDRCHK value) differs from zero. The default error routine is

a simple IDLE; instruction. The user can overwrite the default error handler using the initcode mechanism.

Header Sign Field

The header signature (HDRSGN) byte always reads as 0xAD and is used to verify whether the block pointer actually points to a valid block.

Target Address

This 32-bit field holds the target address where the boot kernel loads the block payload data. When the BFLAG_FILL flag is set, the boot kernel fills the memory with the value stored in the ARGUMENT field starting at this address. If the BFLAG_INIT flag is set the kernel issues a CALL(TARGET ADDRESS) instruction after the optional payload is loaded.

If the BFLAG_FIRST flag is set, the TARGET ADDRESS field contains the start address of the application to which the boot kernel jumps at the end of the boot process. This address will also be stored in the EVT1 register. By default the VisualDSP++ elfloader utility sets this value to 0xFFA0 0000 for compatibility with other Blackfin products.

The target address should be divisible by four, because the boot kernel uses 32-bit DMA for certain operations. The target address must point to valid on-chip memory locations. When booting through peripherals that do not support DMA transfers, the BFLAG_INDIRECT flag must be set if the target address points to L1 instruction memory.

Booting to scratchpad memory is not supported. The scratchpad memory functions as a stack for the boot kernel. The L1 data memory locations 0xFF80 7FF0 to 0xFF80 7FFF are used by the boot kernel and should not be overwritten by the application. The memory range used for intermediate storage as controlled by the BFLAG_INDIRECT switch should only be booted after the last BFLAG_INDIRECT bit is processed. By default the address range 0xFF80 7F00–0xFF80 7FEF is used for intermediate storage.

For normal boot operation, the target address points to RAM memory. There are however a few exceptions where the target address can point to on-chip or off-chip ROM. For example a zero-sized BFLAG_INIT block would instruct the boot kernel to call a subroutine residing in ROM or flash memory. This method is used to activate the CRC32 feature.

Byte Count

This 32-bit field tells the boot kernel how many bytes to process. Normally, this is the size of the payload data of a boot block. If the BFLAG_FILL flag is set there is no payload. In this case the BYTE COUNT field uses the value in its ARGUMENT field to tell the boot kernel how many bytes to process.

The byte count is a 32-bit value that should be divisible by four. Zero values are allowed in all block types. Most boot modes are based upon DMA operation which are only 16-bit words for Blackfin processors. The boot kernel may therefore start multiple DMA work units for large boot blocks. This enables a single block to fill to zero the memory, for example, resulting in compact boot streams. The HWAIT signal may toggle for each work unit.

If the BFLAG_IGNORE flag is set, the byte count is used to redirect the boot source pointer to another memory location. In master boot modes, the byte count is a two's-complement (signed long integer) value. In slave boot modes, the value must be positive.

Argument

This 32-bit field is a user variable for most block types. The value is accessible by the initcode or the callback routine and can therefore be used for optional instructions to these routines. When the CRC32 feature is activated, the ARGUMENT field holds the checksum over the payload of the block.

When the BFLAG_FILL flag is set there is no payload. The argument contains the 32-bit fill value, which is most likely a zero.

If the BFLAG_FIRST flag is set, the argument contains the relative next-DXE pointer for multi-DXE applications. For single-DXE applications the field points to the next free boot source address after the current DXE's boot stream.

Boot Host Wait (HWAIT) Feedback Strobe

The HWAIT feedback strobe is a handshake signal that is used to hold off the host device from sending further data while the boot kernel is busy.

On ADSP-BF50x processors, this feature is implemented by a GPIO that is toggled by the boot kernel as required. The PG3 GPIO is used for this purpose.

The signal polarity of the HWAIT strobe is programmable by an external resistor in the 10 kΩ range.

A pull-up resistor instructs the HWAIT signal to be active high. In this case the host is permitted to send header and footer data when HWAIT is low, but should pause while HWAIT is high. This is the mode used in SPI slave boot on other Blackfin products.

Similarly, a pull-down resistor programs active-low behavior.

Note that the HWAIT signal is implemented slightly differently than on ADSP-BF53x Blackfin processors. In the ADSP-BF50x processors, the meaning of the pulling resistor is inverted and HWAIT is asserted by default during reset.

The boot kernel first senses the polarity on the respective HWAIT pin. Then it enables the output driver but keeps the signal in its asserted state. The signal is not released until the boot kernel is ready for data, or when a receive DMA is started. As soon as the DMA completes, HWAIT becomes active again.

The boot host wait signal holds the host from booting in any slave boot mode and prevents it from being overrun with data. The HWAIT signal is, however, available in all boot modes.

In general the host device must interrogate the HWAIT signal before every word that is sent. This requirement can be relaxed for boot modes using on-chip peripherals that feature larger receive FIFOs. However, the host must not rely on the DMA FIFO since its content is cleared at the end of a DMA work unit.

While the HWAIT signal is only used for boot purposes, it may also play a significant role after booting. In slave boot modes, for example, the host device does not necessarily know whether the Blackfin processor is in an active mode or a power-down mode. For example, the HWAIT signal can be used to signal when the processor is in hibernate mode.

Using HWAIT as Reset Indicator

While the HWAIT signal is mandatory in some boot modes, it is optional in others.

If using a pull-up resistor, the HWAIT signal is driven low for the rest of the boot process (and beyond). If using a pull-down resistor, HWAIT is driven high.

With a pull-down resistor, this feature can be used to simulate an active-low reset output. When the processor is reset, or in hibernate, the GPIO is in a high impedance state and HWAIT is pulled low by the resistor. As soon as the processor recovers and has settled the PLL again, the HWAIT is driven high and can alert external circuits.

Boot Termination

After the successful download of the application into the bootable memory, the boot kernel passes control to the user application. By default this is performed by jumping to the vector stored in the EVT1 register. The

boot kernel provides options to execute an RTS instruction or a RAISE 1 instruction instead. The default behavior can be changed by an initcode routine. The EVT1 register is updated by the boot kernel when processing the BFLAG_FIRST block. See "Servicing Reset Interrupts" on page 24-6 to learn how the application can take control.

Before the boot kernel passes program control to the application it does some housekeeping. Most of the registers that were used are changed back to their default state but some register values may differ for individual boot modes. DMA configuration registers and primary register control registers (UARTx_LCR, SPIx_CTL, etc.) are restored, while others are purposely not restored. For example SPIx_BAUD, UARTx_DLH and UARTx_DLL remain unchanged so that settings obtained during the booting process are not lost.

Single Block Boot Streams

The simplest boot stream consists of a single block header and one contiguous block of instructions. With appropriate flag instructions the boot kernel loads the block to the target address and immediately terminates by executing the loaded block.

Table 24-5 shows an example of a single block boot stream header that could be loaded from any serial boot mode. It places a 256-byte block of instructions at L1 instruction SRAM address 0xFFA0 0000. The flags BFLAG_FIRST and BFLAG_FINAL are both set at the same time. Advanced flags, such as BFLAG_IGNORE, BFLAG_INIT, BFLAG_CALLBACK and BFLAG_FILL, do not make sense in this context and should not be used.

Table 24-5. Header for a Single Block Boot Stream

Field	Value	Comments
BLOCK CODE	0xAD33 C001	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | (DMACODE & 0x1)
TARGET ADDRESS	0xFFA0 0000	Start address of block and application code

Table 24-5. Header for a Single Block Boot Stream (Cont'd)

Field	Value	Comments
BYTE COUNT	0x0000 0100	256 bytes of code
ARGUMENT	0x0000 0100	Functions as next-DXE pointer in multi-DXE boot streams

With the BFLAG_FIRST flag set, the ARGUMENT field functions as the next-DXE pointer. This is a relative pointer to the next free source address or to the next DXE start address in a multi-DXE stream.

Direct Code Execution

Applications may want to avoid long booting times and start code execution directly from 16-bit flash. This feature is called direct code execution. This is a special case of boot termination that replaces the no-boot/bypass mode in the ADSP-BF53x Blackfin processors.

An initial boot block header is needed for the processor to fetch and execute program code from the boot device as early as possible. The safety mechanisms of the block, such as the header signature and the XOR checksum, avoid unpredictable processor behavior due to the boot memory not being programmed with valid data yet. The boot kernel first loads the first block header and checks it for consistency. If the block header is corrupted, the boot kernel goes into a safe idle state and does not start code execution.

If the initial block header checks good, the boot kernel interrogates the block flags. If the block has the BFLAG_FINAL flag set, the boot kernel immediately terminates and jumps directly to the address stored in the EVT1 register. To cause the boot kernel to customize the EVT1 register in advance, the initial blocks must also have the BFLAG_FIRST flag set. The TARGET ADDRESS field is then copied to the EVT1 register. In this way, the TARGET ADDRESS field of the initial block defines the start address of the application.

For example in BMODE = 001, when the block header described in Table 24-6 is placed at address 0x2000 0000, the boot kernel is instructed to issue a JUMP command to address 0x2000 0020.

The development tools must be instructed to link the above block to address 0x2000 0000 and the application code to address 0x2000 0020. An example shown in "Example Direct Code Execution" on page 24-87 illustrates how this is accomplished using the VisualDSP++ tools suite.

Table 24-6. Initial Header for Direct Code Execution in BMODE = 001

Field	Value	Comments
BLOCK CODE	0xAD7B D006	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | BFLAG_IGNORE | (DMACODE & 0x6)
TARGET ADDRESS	0x2000 0020	Start address of application code
BYTE COUNT	0x0000 0010	Ignores 16 bytes to provide space for control data such as version code and build data. This is optional and can be zero.
ARGUMENT	0x0000 0010	Functions as next-DXE pointer in multi-DXE boot streams

For multi-DXE boot streams, Figure 24-9 on page 24-41 shows a linked list of initial blocks that represent different applications.

Advanced Boot Techniques

The following sections describe advanced boot techniques. These techniques are useful for customers developing custom boot routines.

Initialization Code

Initcode routines are subroutines that the boot kernel calls during the booting process. The user can customize and speed up the booting

mechanisms using this feature. Traditionally, an initcode is used to set up system PLL, bit rates, wait states. If executed early in the boot process, the boot time can be significantly reduced.

After the payload data is loaded for a specific boot block, if the BFLAG_INIT flag is set, the boot kernel issues a CALL instruction to the target address of the block.

On ADSP-BF50x Blackfin processors, initcode routines follow the C language calling convention so they can be coded in C language or assembly.

The expected prototype is:

```
void initcode(ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the ADI_BOOT_INITCODE_FUNC type:

```
typedef void ADI_BOOT_INITCODE_FUNC (ADI_BOOT_DATA* ) ;
```

Optionally, the initcode routine can interrogate the formatting structure and customize its own behavior or even manipulate the regular boot process. A pointer to the structure is passed in the R0 register. Assembly coders must ensure that the routine returns to the boot kernel by a terminating RTS instruction.

Initcodes can rely on the validity of the stack, which resides in scratchpad memory. The ADI_BOOT_DATA structure resides on the stack. Rules for register usage conform to the compiler conventions. See the *VisualDSP++ C/C++ Compiler and Library Manual* for more information.

In the simple case, initcodes consist of a single instruction section and are represented by a single block within the boot stream. This block has the BFLAG_INIT bit set.

An init block can consist of multiple sections where multiple boot blocks represent the initcode within the boot stream. Only the last block has the BFLAG_INIT bit set.

The VisualDSP++ elfloader utility ensures that the last of these blocks vectors to the initcode entry address. The utility instructs the on-chip boot ROM to execute a `CALL` instruction to the given target address.

When the on-chip boot ROM detects a block with the `BFLAG_INIT` bit set, it boots the block into Blackfin memory and then executes it by issuing a `CALL` to its target address. For this reason, every initcode must be terminated by an `RTS` instruction to ensure that the processor vectors back to the on-chip boot ROM for the rest of the boot process.

Sometimes initcode boot blocks have no payload and the `BYTE COUNT` field is set to zero. Then the only purpose of the block may be to instruct the boot kernel to issue the `CALL` instruction.

Initcode routines can be very different in nature. They might reside in ROM or SRAM. They might be called once during the booting process or multiple times. They might be volatile and be overwritten by other boot blocks after executing, or they might be permanently available after boot time. The boot kernel has no knowledge of the nature of initcodes and has no restrictions in this regard. Refer to the *VisualDSP++ Loader and Utilities Manual* for how this feature is supported by the tools chain.

It is the user's responsibility to ensure that all code and data sections that are required by the initcode are present in memory by the time the initcode executes. Special attention is required if initcodes are written in C or C++ language. Ensure that the initcode does not contain calls to the runtime libraries. Do not assume that parts of the runtime environment, such as the heap are fully functional. Ensure that all runtime components are loaded and initialized before the initcode executes.

The VisualDSP++ elfloader utility provides two different mechanisms to support the initcode feature.

- The `-init initcode.dxe` command line switch
- The `-initcall address/symbol` command line switch

If enabled by the VisualDSP++ elfloader `-init initcode.DXE` command line switch, the initcode is added to the beginning of the boot stream. Here, `initcode.DXE` refers to the user-provided custom initialization executable— a separate VisualDSP++ project. Figure 24-6 shows a boot stream example that performs the following steps.

1. Boot initcode into L1 memory.
2. Execute initcode.
3. Overwrite initcode with final application code.
4. Boot data/code into memory.
5. Continue program execution with block n.

Although `initcode.DXE` files are built as VisualDSP++ projects, they differ from standard projects. Initcodes provide only a callable sub-function, so they look more like a library than an application. Nevertheless, unlike library files (`.DLB` file extension), the symbol addresses have already been resolved by the linker.

An initcode is always a heading for the regular application code. Consequently whether the initcode consists of one or multiple blocks, it is not terminated by a `BFLAG_FINAL` bit indicator—this would cause the boot ROM to terminate the boot process.

It is advantageous to have a clear separation between the initcode and the application by using the `-init` switch. If this separation is not needed, the elfloader `-initcall` command-line switch might be preferred. It enables fractions of the application code to be traded as initcode during the boot process. See the *VisualDSP++ Loader and Utilities Manual* for further details.

Initcode examples are shown in "Programming Examples" on page 24-81.

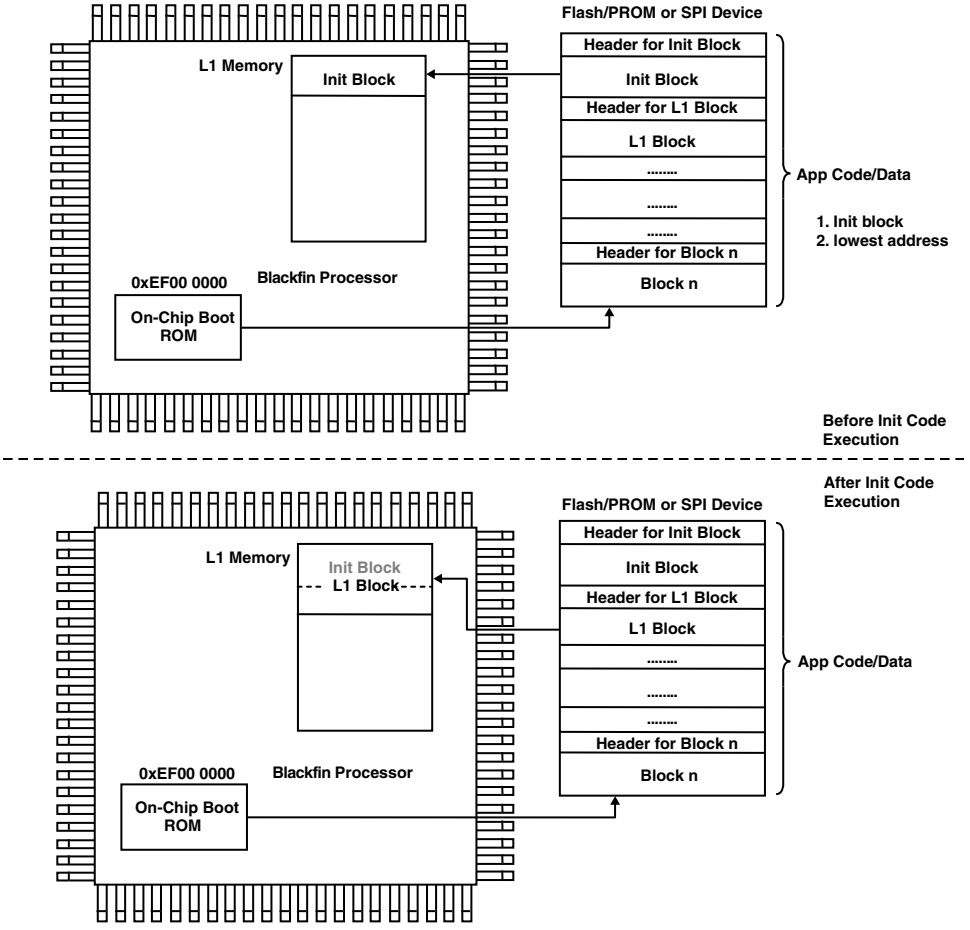


Figure 24-6. Initialization Code Execution/Boot

Quick Boot

In some booting scenarios, not all memories need to be re-initialized.

The ADSP-BF50x processor's boot kernel can conditionally process boot blocks. The normal scenario is all boot, the shortened version is quick boot. It relies on the following primitives.

- The SYSCR register is read to determine what kind of boot is expected from the boot kernel. Refer to Figure 24-22 on page 24-60.

  The WURESET bit is used to distinguish between cold boot and warm boot situations and to identify wake-up from hibernate situations.

  The BCODE bit field in the SYSCR register can overrule the native decision of the boot kernel for a software boot. See the flowchart in Figure 24-1 on page 24-7.

- The BFLAG_WAKEUP bit in the dFlag word of the ADI_BOOT_DATA structure indicates that the final decision was to perform a quick boot. If the boot kernel is called from the application, then the application can control the boot kernel behavior by setting the BFLAG_WAKEUP flag accordingly. See the dFlags variable on Figure 24-27 on page 24-71.

- The BFLAG_QUICKBOOT flag in the BLOCK CODE word of the block header controls whether the current block is ignored for quick boot.

If both the global BFLAG_WAKEUP and the block-specific BFLAG_QUICKBOOT flags are set, the boot kernel ignores those blocks. But since the BFLAG_INIT, BFLAG_CALLBACK, BFLAG_FINAL, and BFLAG_AUX flags are internally cleared and the BFLAG_IGNORE flag is toggled, through double negation, the "ignore the ignore block" command instructs the boot kernel to process the block.

Although the BFLAG_INIT flag is suppressed in quick boot, the user may not want to combine the BFLAG_INIT flag with the BFLAG_QUICKBOOT flag. The initialization code can interrogate the BFLAG_WAKEUP flag and execute conditional instructions.

Indirect Booting

The processor's boot kernel provides a control mechanism to let blocks either boot directly to their final destination or load to an intermediate storage place, then copy the data to the final destination in a second step.

This feature is motivated by the following requirements:

- Some boot modes do not use DMA. They load data by core instruction. The core cannot access some memories directly (for example L1 instruction SRAM), or is less efficient than the DMA in accessing some memories.
- In some advanced booting scenarios, the core needs to access the boot data during the booting process, for example in processing decompression, decryption and checksum algorithms at boot time. The indirect booting option helps speed-up and simplify such scenarios. Software accesses off-chip memory less efficiently and cannot access data directly if it resides in L1 instruction SRAM.

Indirect booting is not a global setting. Every boot block can control its own processing by the BFLAG_INDIRECT flag in the block header.

In general a boot block may not fit into the temporary storage memory so the boot kernel processes the block in multiple steps. The larger the temporary buffer, the faster the boot process. By default the L1 data memory region between 0xFF80 7F00 and 0xFF80 7FEF is used for intermediate storage. Initialization code can alter this region by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure. The default region is at the upper end of a physical memory block.

When increasing the dTempByteCount value, pTempBuffer also has to change.

Callback Routines

Callback routines, like initialization codes, are user-defined subroutines called by the boot kernel at boot time. The BFLAG_CALLBACK flag in the block header controls whether the callback routine is called for a specific block.

There are several differences between initcodes and callback routines. While the BFLAG_INIT flag causes the boot kernel to issue a CALL instruction to the target address of the specific boot block, the BFLAG_CALLBACK flag causes the boot kernel to issue a CALL instruction to the address held by the pCallBackFunction pointer in the ADI_BOOT_DATA structure. While a boot stream can have multiple individual initcodes, it can have just one callback routine. In the standard boot scenario, the callback routine has to be registered by an initcode prior to the first block that has the BFLAG_CALLBACK flag set.

The purpose of the callback routine is to apply standard processing to the block data. Typically, callback routines contain checksum, decryption, decompression, or hash algorithms. Checksum or hash words can be passed through the block header ARGUMENT field.

Since callback routines require access to the payload data of the boot blocks, the block data must be loaded before it can be processed. Unlike initcodes, a callback usually resides permanently in memory. If the block is loaded to L1 instruction memory or off-chip memory, the BFLAG_CALLBACK flag is likely combined with the BFLAG_INDIRECT bit. The boot kernel performs these steps in the following order.

1. Data is loaded into the temporary buffer defined by the pTempBuffer variable.
2. The CALL to the pCallBackFunction is issued.

3. After the callback routine returns, the memory DMA copies data to the destination.

If a block does not fit into the temporary buffer, for example when the BLOCK COUNT is greater than the dTempByteCount variable, the three steps are executed multiple times until all payload data is loaded and processed. The boot kernel passes the parameter dCbFlags to the callback routine to tell it that it is being invoked the first or the last time for a specific block. To store intermediate results across multiple calls the callback routine can use the uwUserShort and dUserLong variables in the ADI_BOOT_DATA structure.

Callback routines meet C language calling conventions for subroutines. The prototype is as follows.

```
s32 CallBackFunction (ADI_BOOT_DATA* pBootStruct,
ADI_BOOT_BUFFER* pCallbackStruct, s32 dCbFlags);
```

The VisualDSP++ header file defines the ADI_BOOT_CALLBACK_FUNC type the following way:

```
typedef s32 ADI_BOOT_CALLBACK_FUNC (ADI_BOOT_DATA*,
ADI_BOOT_BUFFER*, s32 ) ;
```

The pBootStruct argument is passed in R0 and points to the ADI_BOOT_DATA structure used by the boot kernel. These are handled by the pTempBuffer and dTempByteCount variables as well as the pHeader pointer to the ARGUMENT field. The callback routine may process the block further by modifying the pTempBuffer and dTempByteCount variables.

The pCallbackStruct structure passed in R1 provides the address and length of the data buffer. When the BFLAG_INDIRECT flag is not set, the pCallbackStruct contains the target address and byte count of the boot block. If the BFLAG_INDIRECT flag is set, the pCallbackStruct contains a copy of the pTempBuffer. Depending on the size of the boot block and processing progress, the byte count provided by pCallbackStruct equals either dTempByteCount or the remainder of the byte count.

When the BFLAG_INDIRECT flag is set along with the BFLAG_CALLBACK flag, memory DMA is invoked by the boot kernel after the callback routine returns. This memory DMA relies on the pCallbackStruct structure not the global pTempBuffer and dTempByteCount variables.

The callback routine can control the source of the memory DMA by altering the content of the pCallbackStruct structure, as may be required if the callback routine performs data manipulation such as decompression.

The dCbFlags parameter passed in R2 tells the callback routine whether it is invoked the first time (CBFLAG_FIRST) or whether it is called the last time (CBFLAG_FINAL) for a specific block. The CBFLAG_DIRECT flag indicates that the BFLAG_INDIRECT bit is not active so that the callback routine will only be called once per block. When the CBFLAG_DIRECT flag is set, the CBFLAG_FIRST and CBFLAG_FINAL flags are also set.

```
#define CBFLAG_FINAL        0x0008
#define CBFLAG_FIRST        0x0004
#define CBFLAG_DIRECT       0x0001
```

A callback routine also has a boolean return parameter in register R0. If the return value is non-zero, the subsequent memory DMA does not execute. When the CBFLAG_DIRECT flag is set, the return value has no effect.

Error Handler

While the default handler simply puts the processor into idle mode, an initcode routine can overwrite this pointer to create a customized error handler. The expected prototype is

```
void ErrorFunction (ADI_BOOT_DATA* pBootStruct, void
*pFailingAddress);
```

Use an initcode to write the entry address of the error routine to the pErrorFunction pointer in the ADI_BOOT_DATA structure. The error handler has access to the boot structure and receives the instruction address that triggered the error.

CRC Checksum Calculation

The ADSP-BF50x Blackfin processors provide an initcode and a callback routine in ROM that can be used for CRC32 checksum generation during boot time. The checksum routine only verifies the payload data of the blocks. The block headers are already protected by the native XOR checksum mechanism.

Before boot blocks can be tagged with the `BFLAG_CALLBACK` flag to enable checksum calculation on the blocks, the boot stream must contain an initcode block with no payload data and with the CRC32 polynomial in the block header `ARGUMENT` word.

The initcode registers a proper CRC32 wrapper to the `pCallBackFunction` pointer. The registration principle is similar to the XOR checksum example shown in "Programming Examples" on page 24-81.

Load Functions

All boot modes are processed by a common boot kernel algorithm. The major customization is done by a subroutine that must be registered to the `pLoadFunction` pointer in the `ADI_BOOT_DATA` structure. Its simple prototype is as follows.

```
void LoadFunction (ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the following type:

```
typedef void ADI_BOOT_LOAD_FUNC (ADI_BOOT_DATA* ) ;
```

For a few scenarios some of the flags in the `dFlags` word of the `ADI_BOOT_DATA` structure, such as `BFLAG_PERIPHERAL` and `BFLAG_SLAVE`, slightly modify the boot kernel algorithm.

The boot ROM contains several load functions. One performs a memory DMA for flash boot, others perform peripheral DMAs or load data from booting source by polling operation. The first is reused for fill operation and indirect booting as well.

In second-stage boot schemes, the user can create customized load functions or reuse the original BFROM_PDMA routine and modify the pDmaControlRegister, pControlRegister and dControlValue values in the ADI_BOOT_DATA structure. The pDmaControlRegister points to the DMAx_CONFIG or MDMA_Dx_CONFIG register. When the BFLAG_SLAVE flag is not set, the pControlRegister and dControlValue variables instruct the peripheral DMA routine to write the control value to the control register every time the DMA is started.

Load functions written by users must meet the following requirements.

- Protect against dByteCount values of zero.
- Multiple DMA work units are required if the dByteCount value is greater than 65536.
- The pSource and pDestination pointers must be properly updated.

In slave boot modes, the boot kernel uses the address of the dArgument field in the pHeader block as the destination for the required dummy DMAs when payload data is consumed from BFLAG_IGNORE blocks. If the load function requires access to the block's ARGUMENT word, it should be read early in the function.

The most useful load functions BFROM_MDMA and BFROM_PDMA are accessible through the jump table. Others, do not have entries in the jump table. Their start address can be determined with the help of the hook routine when calling the respective BFROM_SPIBOOT or other functions. In this way, they can be re-purposed for runtime utilization.

Calling the Boot Kernel at Runtime

The boot kernel's primary purpose is to boot data to memory after power-up and reset cycles. However some of the routines used by the boot kernel might be of general value to the application. The boot ROM supports reuse of these routines as C-callable subroutines. Programs such as

second-stage boot kernels, boot managers, and firmware update tools may call the function in the ROM at runtime. This could load entirely different applications or a fraction of an application, such as a code overlay or a coefficient array.

To call these boot kernel subroutines, the boot ROM provides an API at address 0xEF00 0000 in the form of a jump table.

When calling functions in the boot ROM, the user must ensure the presence of a valid stack following C language conventions. See the VisualDSP++ Compiler documentation for details.

Debugging the Boot Process

If the boot process fails, very little information can be gained by watching the chip from outside. In master boot modes, the interface signals can be observed. In slave boot modes only the HWAIT or the $\overline{\text{RTS}}$ signals tell about the progress of the boot process.

However, by using the emulator, there are many possibilities for debugging the boot process. The entire source code of the boot kernel is provided with the VisualDSP++ installation. This includes the project executable (DXE) file. The LOAD SYMBOLS feature of the VisualDSP++ IDDE helps to navigate the program. Note that the content of the ROM might differ between silicon revisions. Hardware breakpoints and single-stepping capabilities are also available.

Table 24-7 identifies the program symbols in the boot kernel for debug.

Table 24-7. Boot Kernel Symbols for Debug

Symbol	Comment
_bootrom.assert.default	If the program counter halts at the IDLE instruction at the _bootrom.assert.default address, the boot kernel has detected an error condition and will not continue the boot process. A misformatted boot stream is the most likely cause of such an error. The RETS register points to the failing routine. When stepping a couple of instructions further, there is a way to ignore the error and to continue the boot process by clearing the >ASTAT register while the emulator steps over the subsequent IF CC JUMP 0 instruction.
_bootrom.bootmenu	If the emulator hits a hardware breakpoint at the _bootrom.bootmenu address, this indicates that a valid boot mode is being used.
_bootrom.bootkernel.entry	If the emulator hits a hardware breakpoint at the _bootrom.bootkernel.entry label, this indicates that device detection or autobaud returned properly.
_bootrom.bootkernel.breakpoint	This is a good address to place a hardware breakpoint. The boot kernel loads a new block header at this breakpoint. The block header can be watched at address 0xFF80 7FF0 or wherever the pHeader points to.
_bootrom.bootkernel.initcode	All payload data of the current block is loaded by the time the program passes the _bootrom.bootkernel.initcode label. The boot kernel is about to interrogate the BFLAG_INIT flag. If set, the initcode can be debugged.
_bootrom.bootkernel.exit	Once the boot kernel arrives at the _bootrom.bootkernel exit label, it detects a BFLAG_FINAL flag. After some housekeeping, it jumps to the EVT1 vector.

The boot kernel also generates a circular log file in scratch pad memory. While the pLogBuffer and the dLogByteCount variables describe the location and dimension of the log buffer, the pLogCurrent points to the next free location in the buffer. The log file is updated whenever the kernel passes the _bootrom.bootkernel.breakpoint label.

At each pass, nine 32-bit words are written to the log file, as follows.

- block code word (`dBlockCode`) of the block header
- target address (`pTargetAddress`) of the block header
- byte count (`dByteCount`) of the block header
- argument word (`dArgument`) of the block header
- source pointer (`pSource`) of the boot stream
- block count (`dBlockCount`)
- internal copy of the `dBlockCode` word OR'ed with `dFlags`
- content of the `SEQSTAT` register
- 0xFFFF FFFA (-6) constant

The ninth word is overwritten by the next entry set, so that 0xFFFF FFFA always marks the last entry in the log file.

Most of the data structures used by the boot kernel reside on the stack in scratchpad memory. While executing the boot kernel routine (excluding subroutines), the `P5` points to the `ADI_BOOT_DATA` structure. Type "`(ADI_BOOT_DATA*) $P5`" in the VisualDSP++ expression window to see the structure content.

Boot Management

Blackfin processor hardware platforms may be required to run different software at different times. An example might be a system with at least one application and one in-the-field firmware upgrade utility. Other systems may have multiple applications, one starting then terminating, to be replaced by another application. Conditional booting is called boot management. Some applications may self-manage their booting rules, while

others may have a separate application that controls the process, namely a boot manager.

In a master boot mode where the on-chip boot kernel loads the boot stream from memory, the boot manager is a piece of Blackfin software which decides at runtime what application is booted next. This may simply be based on the state of a GPIO input pin interrogated by the boot manager, or it may be the conclusion of complex system behavior.

Slave boot scenarios are different from master boot scenarios. In slave boot modes, the host masters boot management by setting the Blackfin processor to reset and then applying alternate boot data. Optionally, the host could alter the BMODE configuration pins, resulting in little impact to the Blackfin processor since the intelligence is provided by the host device.

Booting a Different Application

The boot ROM provides a set of user-callable functions that help to boot a new application (or a fraction of an application). Usually there is no need for the boot manager to deal with the format details of the boot stream.

These functions are:

- BFROM_MEMBOOT discussed in "Flash Boot Modes" on page 24-44
- BFROM_SPIBOOT discussed in "SPI Master Boot Modes" on page 24-46

The user application, the boot manager application, or an initcode can call these functions to load the requested boot data. Using the BFLAG_RETURN flag the user can control whether the routine simply returns to the calling function or executes the loaded application immediately.

These ROM functions expect the start address of the requested boot stream as an argument. For BFROM_MEMBOOT, this is a Blackfin memory address, for BFROM_SPIBOOT it is a serial address. The SPI function can also

accept the code for the GPIO pin that controls the device select strobe of the SPI memory.

Multi-DXE Boot Streams

If the start addresses of all the boot streams are predefined, the boot manager needs only to call the ROM functions directly. However since the addresses tend to vary from build to build they may have to be calculated at runtime.

In the world of the VisualDSP++ elfloader, a boot stream is always generated from a DXE file. It is therefore common to talk about multi-DXE or multi-application booting. When the elfloader utility accepts multiple DXE files on its command line, it generates a contiguous boot image by default. The second boot stream is appended immediately to the first one. Since the utility updates the `ARGUMENT` field of all `BFLAG_FIRST` blocks, the `ARGUMENT` field of a `BFLAG_FIRST` block is called next-DXE pointer (NDP).

The next-DXE pointer of the first DXE boot stream points relatively to the start address of the second DXE boot stream. A multi-DXE boot image can be seen as a linked list of boot streams. The next-DXE pointer of the last DXE boot stream points relatively to the next free address. This is illustrated by an example shown in the next two figures. Figure 24-7 shows a commented sketch as an example. Figure 24-8 shows a screenshot of the Blackfin loader file viewer utility for the same example. The `LdrViewer` utility is not part of the VisualDSP++ tools suite. It is a third-party freeware product available on www.dolomitics.com.

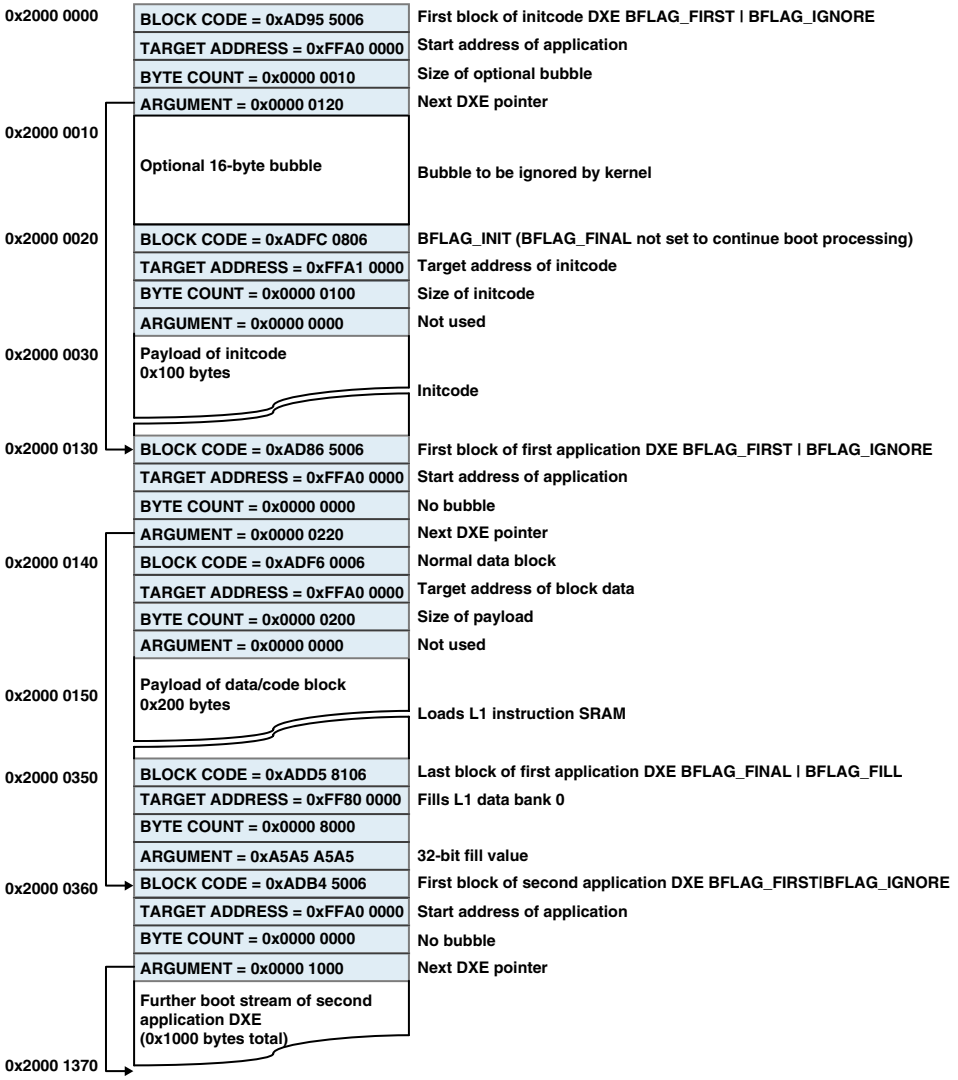


Figure 24-7. Multi-DXE Boot Stream Example for Flash Boot

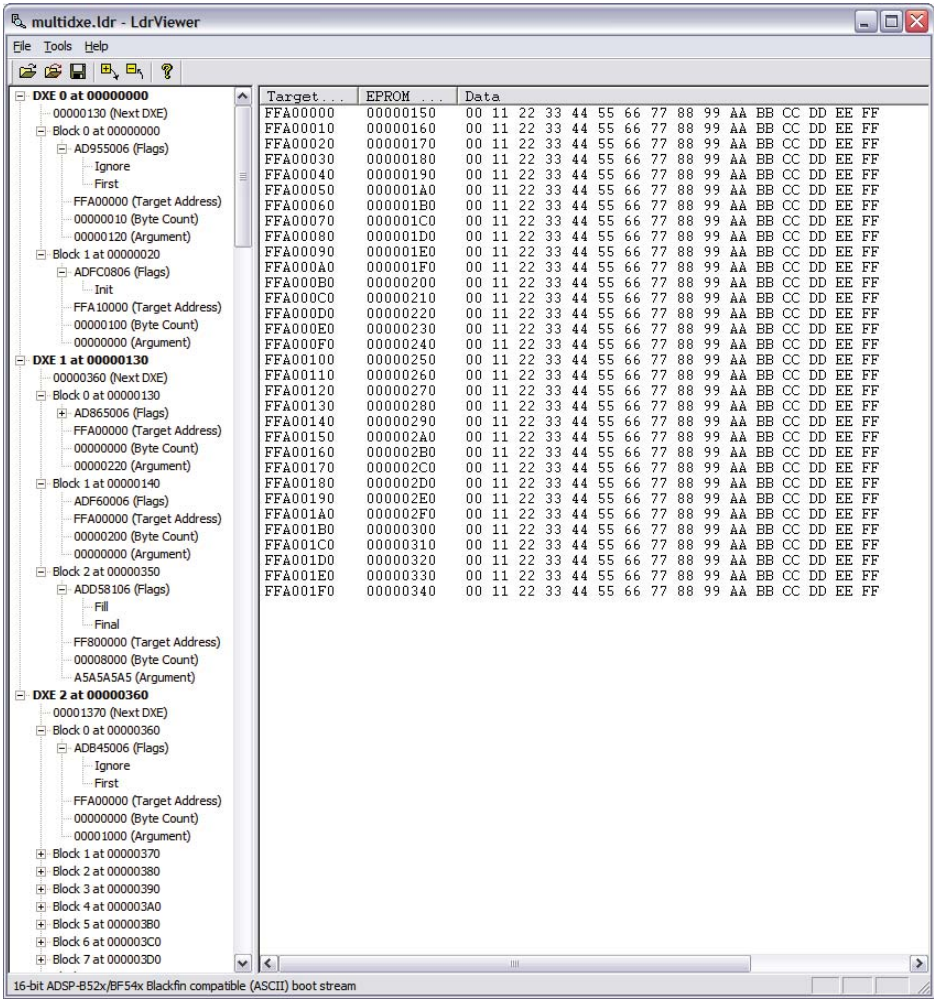


Figure 24-8. LdrViewer Screen Shot

Boot management principles are not only applicable to multi-DXE boot streams. The same scheme, as shown in Figure 24-9, can be applied to direct code executions of multiple applications. See “Direct Code Execution” on page 24-21 for more information. The example shows a linked list of initial block headers that instruct the boot kernel to terminate

immediately and to start code execution at the address provided by the TARGET ADDRESS field of the individual blocks. There is nothing in the boot ROM that prevents multi-DXE applications from mixing regular boot streams and direct code execution blocks.

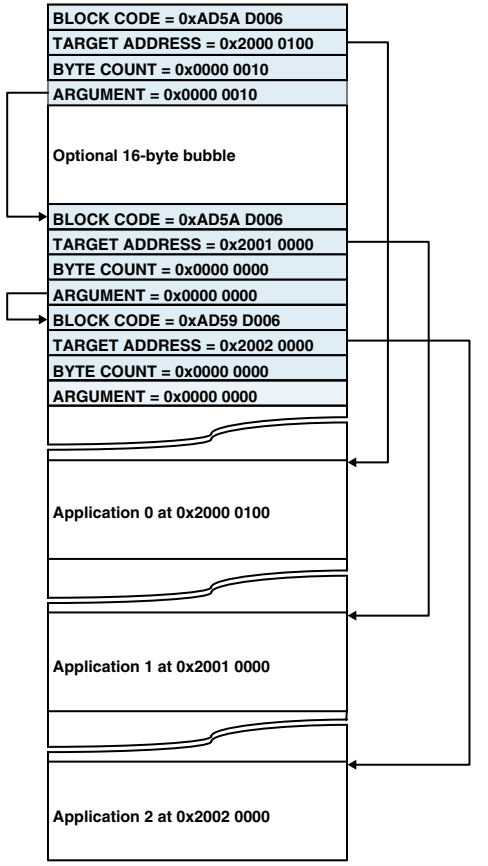


Figure 24-9. Multi-DXE Direct Code Execution Arrangement Example

Determining Boot Stream Start Addresses

The ROM functions BFROM_MEMBOOT, BFROM_SPIBOOT, etc. not only allow the application to boot a subroutine residing at a given start address, they also assist in walking through linked multi-DXE streams.

When the BFLAG_NEXTDXE bit in dFlags is set and these functions are called, the system does not boot but instead walks though the boot stream following the next-DXE pointers. The dBlockCount parameter can be used to specify the DXE of interest. The routines then return the start address of the requested DXE's boot stream.

Initialization Hook Routine

When the ROM functions BFROM_MEMBOOT, BFROM_SPIBOOT, etc. are called, they create an instance of the ADI_BOOT_DATA structure on the stack and fill the items with default values. If the BFLAG_HOOK is set, the boot kernel invokes a callback routine which was passed as the fourth argument of the ROM routines, after the default values have been filled. The hook routine can be used to overwrite the default values. Every hook routine should fit the prototype:

```
void hook (ADI_BOOT_DATA* pBS);
```

The VisualDSP++ header files define the ADI_BOOT_HOOK_FUNC type the following way:

```
typedef void ADI_BOOT_HOOK_FUNC (ADI_BOOT_DATA*);
```

The hook function also gives access to the DMA load function used by the respective boot mode, which can be used for general purposes at runtime. For example, in the BFROM_SPIBOOT case, an instance of the load function:

```
ADI_BOOT_LOAD_FUNC *pSpiLoadFunction;
```

can be initialized by equipping the hook function with the instruction:

```
pSpiLoadFunction = pBS->pLoadFunction;
```

Specific Boot Modes

This section discusses individual boot modes and the required hardware connections.

The boot modes differ in terms of the booting source— for example whether data is loaded through the SPI or the parallel interface. Boot modes can also be grouped into slave boot modes and master boot modes.

In slave boot modes, the Blackfin processor functions as a slave to any host device, which is typically another embedded processor, an FPGA device or even a desktop computer. Likely, the Blackfin processor $\overline{\text{RESET}}$ input is controlled by the host device. So, usually the host sets $\overline{\text{RESET}}$ first, then waits until the preboot routine terminates by sensing the HWAIT output, and finally provides the boot data.

If a Blackfin processor, configured to operate in any of the slave boot modes, awakens from hibernate, it cannot boot by its own control. A feedback mechanism has to be implemented at the system level to inform the host device whether the processor is in hibernate state or not. The HWAIT strobe is an important primitive in such systems.

In the master boot modes, the Blackfin processor usually does not need to be synchronized and can load the boot data by itself. Master modes typically read from memory. This can be parallel memory such as flash devices, or serial memory that is read through SPI interfaces.

Memory boot modes should also be differentiated from peripheral boot modes. Boot modes that load boot streams through memory DMA are referred to as memory boot mode, reading data from regular memory. Peripheral modes load boot data through peripherals such as UART. All memory boot modes are master modes. The boot source is typically non-volatile memory, such as a flash or EPROM device or even on-chip ROM. When supported by the system in warm boot scenarios, the boot source can also be SRAM.

Whether from the host (slave booting mode) or from memory (master booting mode), the boot source does not need to know about the structure of the boot stream.

No Boot Mode

When the BMODE pins are all tied low (BMODE = 000), the Blackfin processor does not boot. Instead, it executes an IDLE instruction, preventing it from executing any instructions provided by the regular boot source. The purpose of this mode is to bring the processor up to a clean state after reset.

When connecting an emulator and starting a debug session, the processor awakens from an idle due to the emulation interrupt and can be debugged in the normal manner.

The no boot mode is not the same as the bypass mode featured by the ADSP-BF53x Blackfin processor. To simulate that bypass mode feature using BMODE = 000, see "Direct Code Execution" on page 24-21 and "Example Direct Code Execution" on page 24-87.

Flash Boot Modes

These booting modes are intended to boot from internal parallel synchronous flash memory of ADSP-BF504F or ADSP-BF506F processor. The flash boot modes are activated by either BMODE = 001 (asynchronous mode) or BMODE = 010 (synchronous mode).

- BMODE1 – Boot from Internal Parallel Flash (Asynchronous Mode)

  In this mode conservative timing parameters are used to communicate with the flash device. The boot kernel communicates with the flash device asynchronously.

- BMODE2 – Boot from Internal Parallel Flash (Synchronous Burst Mode)

In this mode fast timing parameters are used to communicate with the flash device. The boot kernel configures the flash device for synchronous burst communication and boots from the flash synchronously.

For the flash modes, the DMA options shown in Table 24-8 are supported.

Table 24-8. DMA Options

DMACODE	DMA Width	Source Modify	Comment
1	8	1	Not recommended Provides ADSP-BF533 style 8-bit boot from 16-bit flash memory
2	8	2	8-bit MDMA boots from 8-bit flash mapped to lower byte of address bus.
6	16	2	16-bit MDMA boots from 16-bit flash
10	32	4	32-bit MDMA boots from 16-bit flash

The `DMACODE` field is filled by the elfloader utility based on boot mode, `-width` and `-dmawidth` settings. See the *VisualDSP++ Loader and Utility Manual* for details.

After the boot kernel has loaded and interpreted the first four 16-bit words, it continues loading the rest of the first block header and processes the boot stream.

Hardware configuration is shown in Table 6-1 on page 6-2. The chip select is always controlled by the $\overline{\texttt{AMS0}}$ strobe. This maps the boot stream to the Blackfin processor's address 0x2000 0000.

Internal parallel flash provides write protection mechanisms, which can be activated during the power-up and reset cycles of the Blackfin processor. For details, see "Internal Flash Memory Control Registers" on page 6-88.

The flash boot modes can also be used to instruct the boot kernel to terminate immediately and directly execute code from the 16-bit flash memory instead. Code execution from 8-bit flash memory is not supported. See "Direct Code Execution" on page 24-21 for details.

SPI Master Boot Modes

The ADSP-BF50x processors feature booting from off-chip SPI memory.

The external SPI boot mode (BMODE = 011) boots from SPI memories connected to the SPI0_SSEL1 interface. 8-, 16-, 24-, and 32-bit address words are supported. Standard SPI memories are read using either the standard 0x03 SPI read command or the 0x0B SPI fast read command.

Unlike other Blackfin processors, the ADSP-BF50x Blackfin processors have no special support for DataFlash devices from Atmel. Nevertheless, DataFlash devices can be used for booting and are sold as standard 24-bit addressable SPI memories. They also support the fast read mode. If used for booting, DataFlash memory must be programmed in the power-of-2 page mode.

For booting, the SPI memory is connected as shown in Figure 24-10.

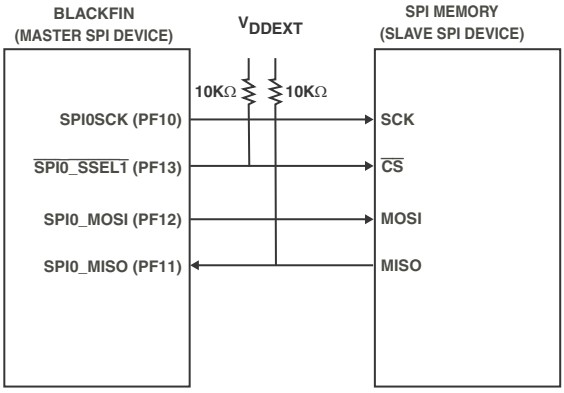


Figure 24-10. Blackfin to SPI Memory Connections

The pull-up resistor on the MISO line is required for automatic device detection. The pull-up resistor on the $\overline{\text{SPI0\_SSEL1}}$ line ensures that the memory is in a known state when the Blackfin GPIO is in a high-impedance state (for example, during reset). A pull-down resistor on the SPI0SCK line displays cleaner oscilloscope plots during debugging.

For SPI master boot, the SPE, MSTR and SZ bits are set in the SPI0_CTL register. For details see Chapter 18, "SPI-Compatible Port Controller". With TIMOD = 2, the receive DMA mode is selected. Clearing both the CPOL and CPHA bits results in SPI mode 0. The boot kernel does not allow SPI0 hardware to control the $\overline{\text{SPI0\_SSEL1}}$ pin. Instead, this pin is toggled in GPIO mode by software. Initialization code is allowed to manipulate the uwSsel variable in the ADI_BOOT_DATA structure to extend the boot mechanism to a second SPI memory connected to another GPIO pin.

By default, the boot kernel sets the SPI0_BAUD register to a value of 133, resulting in a bit rate of SCLK/266 (as shown in Table 24-9).

Table 24-9. Bit Rate

SPI_BAUD	Bit Rate
133	SCLK/(2x133) << default
Reserved	
2	SCLK/(2x2)
4	SCLK/(2x4)
8	SCLK/(2x8)
16	SCLK/(2x16)
32	SCLK/(2x32)
64	SCLK/(2x64)

Similarly, the boot kernel uses the standard 0x03 SPI read command, by default.

SPI Device Detection Routine

Since BMODE = 011 supports booting from various SPI memories, the boot kernel automatically detects what type of memory is connected. To determine whether the SPI memory device requires an 8-, 16-, 24- or 32-bit addressing scheme, the boot kernel performs a device detection sequence prior to booting. The MISO signal requires a pull-up resistor, since the routine relies on the fact that memories do not drive their data outputs unless the right number of address bytes are received.

Initially, the boot kernel transmits a read command (either 0x03 or 0x0B) on the MOSI line, which is immediately followed by two zero bytes. Once the transmission is finished, the boot kernel interrogates the data received on the MISO line. If it does not equal 0xFF (usually a DMACODE value of 0x01 is expected), then an 8-bit addressable device is assumed.

If the received value equals 0xFF, it is assumed that the memory device has not driven its data output yet and that the 0xFF value is due to the pull-up resistor. Thus, another zero byte is transmitted and the received data is tested again. If it differs from 0xFF, either a 16-bit addressable device (standard mode) or an 8-bit addressable device (fast read mode) is assumed.

If the value still equals 0xFF, device detection continues. Device detection aborts immediately if a byte different than 0xFF is received. The boot kernel continues with normal boot operation and it re-issues a read command to read from address 0 again. The first block header is loaded by two read sequences, further block headers and block payload fields are loaded by separate read sequences.

Figure 24-11 illustrates how individual devices would behave.

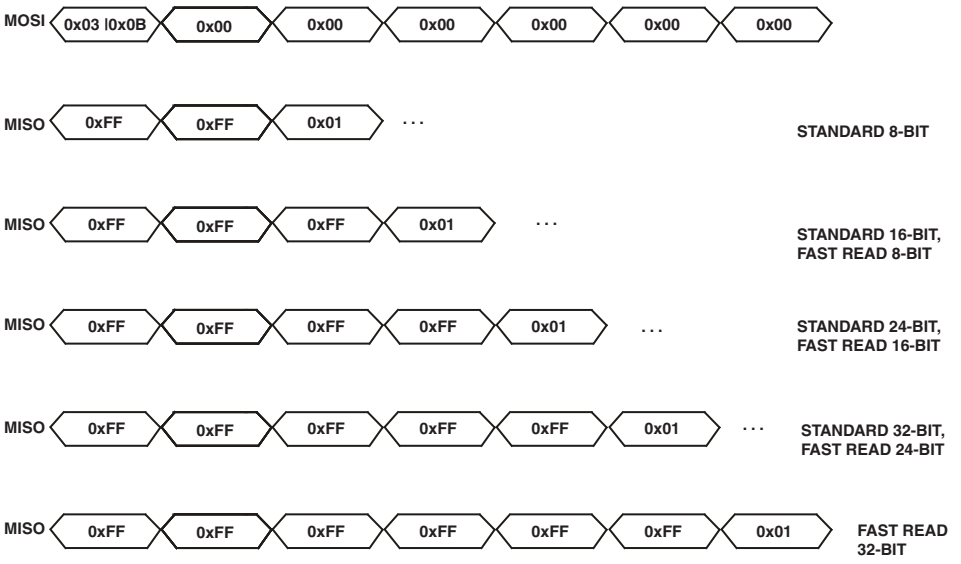


Figure 24-11. SPI Device Detection Principle

Figure 24-12 shows the initial signaling when a 24-bit addressable SPI memory is connected in SPI master boot mode. After RESET releases, a 0x03 command is transmitted to the MOSI output, followed by a number of 0x00 bytes. The 24-bit addressable memory device returns a first data byte at the fourth zero byte. Then, the device detection has completed and the boot kernel re-issues a 0x00 address to load the boot stream.

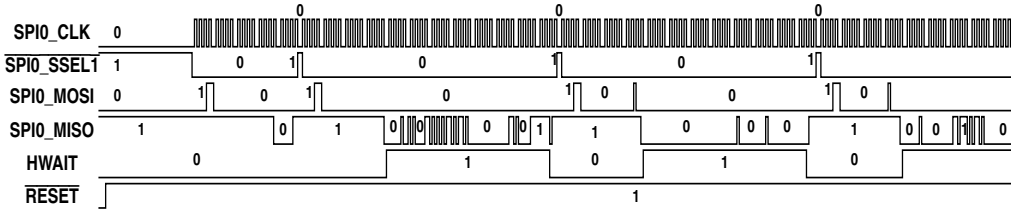


Figure 24-12. Typical SPI Master Boot Waveforms

SPI Slave Boot Mode

For SPI slave mode boot (BMODE = 100), the Blackfin processor is consuming boot data from an external SPI host device. SPI0 is configured as an SPI slave device. The hardware configuration is shown in Figure 24-13. As in all slave boot modes, the host device controls the Blackfin processor $\overline{\text{RESET}}$ input.

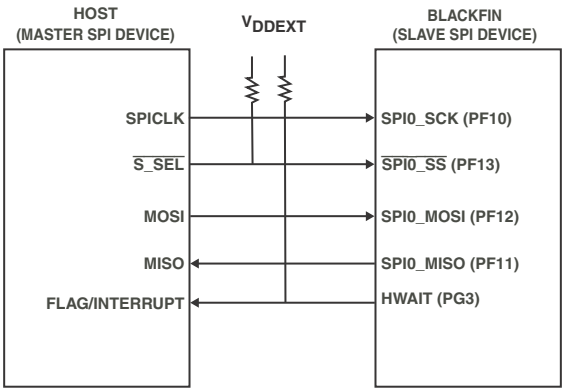


Figure 24-13. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

The host drives the SPI clock and is responsible for the timing. The host must provide an active-low chip select signal that connects to the $\overline{\text{SPIOSS}}$ input of the Blackfin processor. It can toggle with each byte transferred or remain low during the entire procedure. 8-bit data is expected. The 16-bit mode is not supported.

In SPI slave boot mode, the boot kernel sets the CPHA bit and clears the CPOL bit in the SPIO_CTL register. Therefore the MISO pin is latched on the falling edge of the MOSI pin. For details see Chapter 18, “SPI-Compatible Port Controller”. In SPI slave boot mode, HWAIT functionality is critical. When high, the resistor shown in Figure 24-13 programs HWAIT to hold off the host. HWAIT holds the host off while the Blackfin processor is in reset.

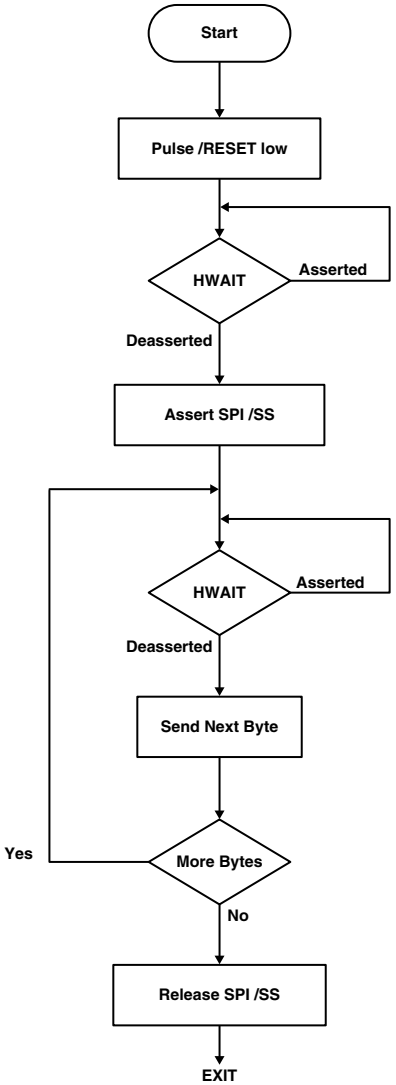


Figure 24-14. SPI Program Flow on Host Device

After HWAIT turns inactive, the host can send boot data. The SPI module does not provide very large receive FIFOs, so the host must test the HWAIT

signal for every byte. Figure 24-14 illustrates the required program flow on the host side.

Figure 24-15 shows the initial waveform for an SPI slave boot case. As soon as the Blackfin processor releases HWAIT after reset, the host device pulls the SPIOSS pin low and starts transmitting data. After the eighth data word has been received, the boot kernel asserts HWAIT again as it has to process the DMACODE field of the first block header. When the host detects the asserted HWAIT it gracefully finishes the transmission of the on-going word. Then, it pauses transmission until HWAIT releases again.

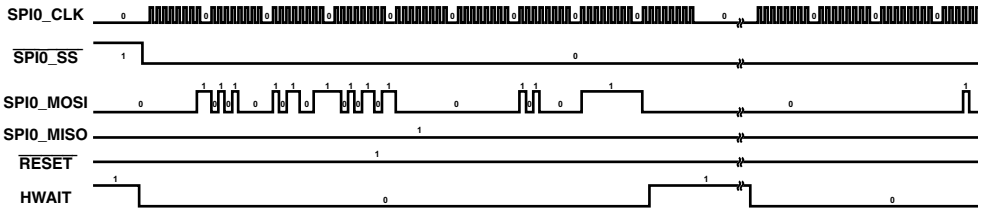


Figure 24-15. Typical SPI Slave Boot Waveforms

PPI Boot Mode

The ADSP-BF50x processors feature a 16-bit PPI boot mode (BMODE = 101). The PPI is a half-duplex bi-directional port consisting of up to 16 data lines, 3 frame synchronization signals and a clock signal.

In PPI boot mode, the PPI mode of operation is configured as follows:

- Receive mode with 1 external frame sync
- 16-bit bus width
- Data sampled on falling edge of clock
- Frame sync configured for falling edge asserted
- PPI_DELAY value of 0x0

The external frame sync signal is on PPI_FS1. This signal is driven low by the host at the start of a data transfer with a 16-bit word being transferred on each PPI_CLK cycle that the PPI_FS1 signal is asserted low.

In order to simplify the PPI host design, PPI boot mode also configures Timer1 for PWM mode of operation. The PWM circuits of the timer are configured to be clocked by the externally provided PPI_CLK signal allowing for arbitrary pulse widths and pulse periods to be programmed thus simulating an internally generated frame sync signal on the PPI_FS2 signal. This configuration lets the processor inform the host when the processor is ready to receive data and also how much data is expected. This feature removes the need for the host to process the actual contents of the boot stream to identify the size of the data transfer.

The PPI host can synchronize the PPI_FS2 signal to PPI_CLK signal and initiate all data transfers accordingly. The PPI_FS2 signal can be looped back to the PPI_FS1. (See Figure 24-16.)

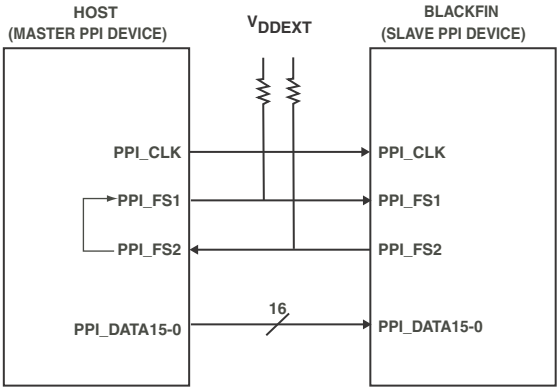


Figure 24-16. PPI Slave Boot Mode Connections

The Timer1 is configured to generate a periodic pulse as opposed to a single shot pulse. The pulse period is set to the maximum of 0xFFFFFFFF allowing for any transfer size supported by the kernel. Note the current

16-bit DMA X Count limits the maximum width of a pulse to 0xFFFF words.

After completion of the DMA transfer, the PWM_OUT out mode is terminated and cleared in the required manner. This mode of operation does impose some restrictions on the amount of time that the PPI host device can hold off a transfer. If a DMA transfer consists of 0xFFFF words, the timer period will be reached 0xFFFF0000 PPI_CLK cycles after the deassertion of the PPI_FS2/TMR1 signal. This will result in the generation of an identical PPI_FS2/TMR1 pulse if the DMA transfer has not completed and the PWM_OUT timer has not been disabled.

In the unlikely event that a user requires a transfer to be held off for this significant amount of time, the PPI host must be able to ignore any further PPI_FS2/TMR1 assertions until the currently pending transaction that was delayed has completed. If the master is not capable of ignoring further PPI_FS2/TMR1 assertions, the master must ensure that the DMA completes allowing for the PWM_OUT timer to be disabled prior to the completion of the timer pulse period of 0xFFFFFFFF PPI_CLK cycles.

After PPI boot completion the PPI interface is disabled and the PPI_CONTROL register is cleared, this register-clearing operation is not done for the Timer1 registers. Although the timer is disabled, the TIMER1_CONFIG register is not reloaded with the default reset value.

UART Slave Mode Boot

Figure 24-17 shows the interconnection required for booting. The figure does not show physical line drivers and level shifters that are typically required to meet the individual UART-compatible standards.

For BMODE = 111, the ADSP-BF50x processor consumes boot data from a UART host device connected to the UART0 interface. Automatic control of the $\overline{\text{UA0\_RTS}}$ output provides flow control.

The host downloads programs formatted as boot streams using an auto-baud detection sequence. The host selects a bit rate within the UART clocking capabilities. To determine the bit rate when performing the auto-baud, the boot kernel expects an "@" character (0x40, eight data bits, one start bit, one stop bit, no parity bit) on the UART UA0_RX input. The boot kernel acknowledges, and the host then downloads the boot stream. The acknowledgement consists of four bytes: 0xBF, UARTx_DLL, UARTx_DLH, 0x00. The host is requested to not send further bytes until it has received the complete acknowledge string. Once the 0x00 byte is received, the host can send the entire boot stream. The host should know the total byte count of the boot stream, but it is not required to have any knowledge about the content of the boot stream. Further information regarding auto-baud detection is given in "Autobaud Detection" on page 15-20.

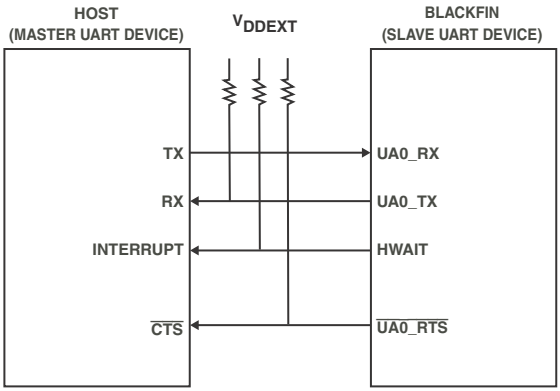


Figure 24-17. UART Slave Boot Mode Connections

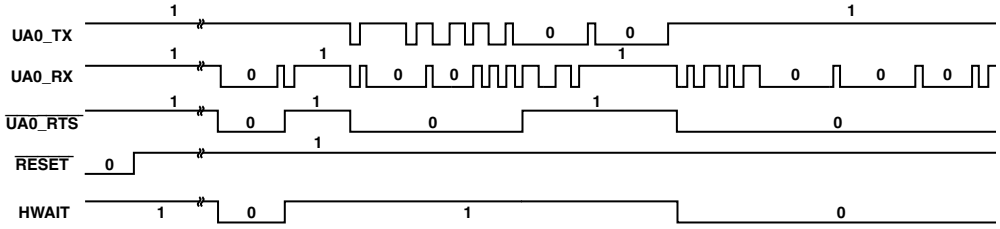


Figure 24-18. UART Autobaud Waveform

When the boot kernel is processing fill or initcode blocks it might require extra processing time and needs to hold the host off from sending more data. This is signalled with the HWAIT output as well as by the RTS output. When equipped with a pull-up resistor the HWAIT signal imitates the behavior of an $\overline{\text{UA0\_RTS}}$ output and could be connected to the $\overline{\text{CTS}}$ input of the booting host.The host is not allowed to send data until HWAIT turns inactive after a reset cycle. Therefore a pulling resistor on the HWAIT signal is required.

If the resistor pulls to ground, the host must pause transmission when HWAIT is low and is permitted to send when HWAIT is high. A pull-up resistor inverts the signal polarity of HWAIT. The host should test HWAIT at every transmitted byte.

During ADSP-BF50x boot operation, the host device more likely relies on the RTS output of UART0. Then, the use of HWAIT becomes optional. At boot time the Blackfin processor does not evaluate RTS signals driven by the host and the UART0 $\overline{\text{UA0\_CTS}}$ input is inactive. Since the $\overline{\text{UA0\_RTS}}$ is in a high impedance state when the Blackfin processor is in reset or while executing preboot, an external pull-up resistor to V_{DDEXT} is recommended.

Figure 24-19 and Figure 24-20 show the initial case of the UART boot mode. As soon as HWAIT releases after reset, the boot kernel expects to receive a 0x40 byte for bit rate detection. After the bit rate is known, the UART is enabled and the kernel transmits for bytes.

Figure 24-19 and Figure 24-20 compare $\overline{\text{UA0\_RTS}}$ and HWAIT timing when an extended initcode executes. Since code execution distracts from data loading, the host device should be prevented from sending more data. The HWAIT timing is much more conservative than the $\overline{\text{RTS}}$. If the host relies on HWAIT, the UART receive buffer may not be filled over watermark level and $\overline{\text{UA0\_RTS}}$ might not be deasserted at all. If, however, the host relies on $\overline{\text{UA0\_RTS}}$, it will be stalled a couple of bytes later. Both methods are valid.

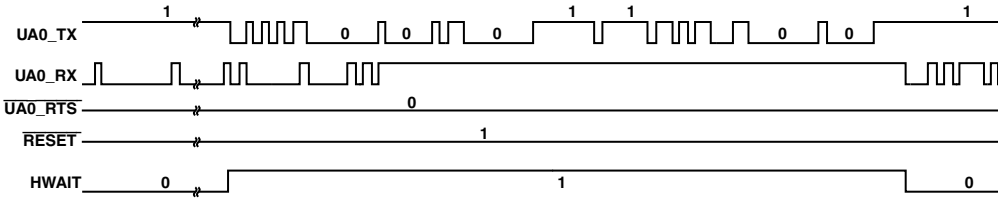


Figure 24-19. UART Boot - Host Relying on HWAIT

As shown in Figure 24-20, when the UART is enabled, UA0_RTS goes low, encouraging the host to send the boot stream data immediately. With a half-duplex UART connection this must be avoided. The host should either rely on the HWAIT signal or wait until it has received the four bytes from the Blackfin processor, before sending any data.

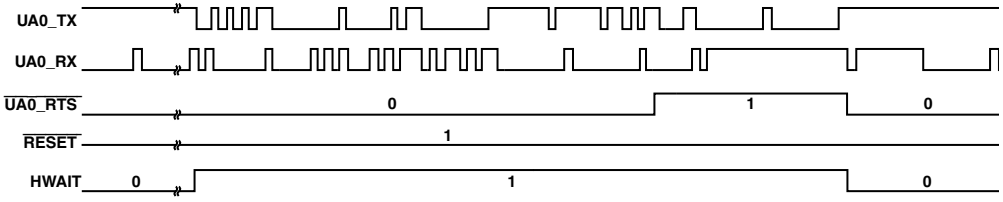


Figure 24-20. UART Boot - Host Relying on RTS

For UART boot, it is not obvious on how to change the PLL by an initcode routine. This is because the UARTx_DLL and UARTx_DLH registers have to be updated to keep the required bit rate constant after the SCLK frequency has changed. It must be ensured that the host does not send data while the PLL is changing. The initcode examples provided along with the VisualDSP++ tools installation demonstrate how this can be accomplished.

Reset and Booting Registers

Two registers are used for reset and booting—the software reset register (SWRST) and the system reset configuration register (SYSCR).

Software Reset (SWRST) Register

A software reset can be initiated by setting bits [2:0] in the system software reset field in the software reset register (SWRST) shown in Figure 24-21.

Bit 3 can be used to generate a reset upon core-double-fault. A core-double-fault resets both the core and the peripherals, but not the RTC block and most of the DPMC. Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 indicates the software watchdog timer has generated the software reset. Bit 13 indicates the core-double-fault has generated the software reset. Bits [15:13] are read-only and cleared when the register is read. Reading the SWRST also clears bits [15:13] in the SYSCR register. Bits [3:0] are read/write.

Only writing to bits[2:0], resets only the modules in the SCLK domain. It does not clear the core. The program executes normally at the instruction after the MMR write to SWRST. The system is kept in the reset state as long as the bits[2:0] are set to b#111. To release reset, write a zero again. Examples for this are available in assembly (Listing 24-1 on page 24-81) and C (Listing 24-2 on page 24-82). It is not recommended that this functionality be used directly. Rather, call the ROM function bfrom_SysControl() to perform a system reset.

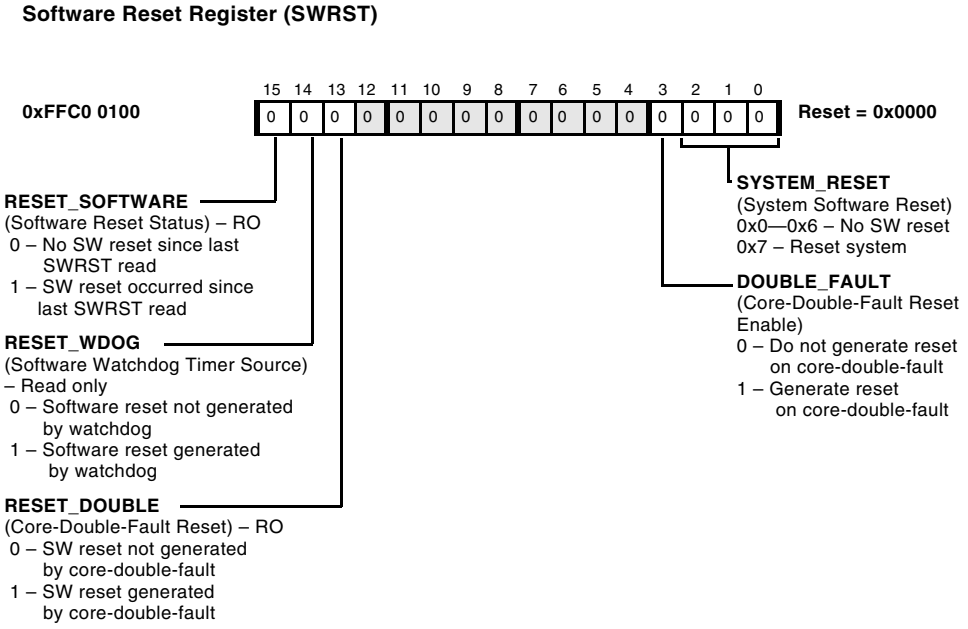


Figure 24-21. Software Reset Register

System Reset Configuration (SYSCR) Register

The software reset configuration register (SYSCR) is shown in Figure 24-22.

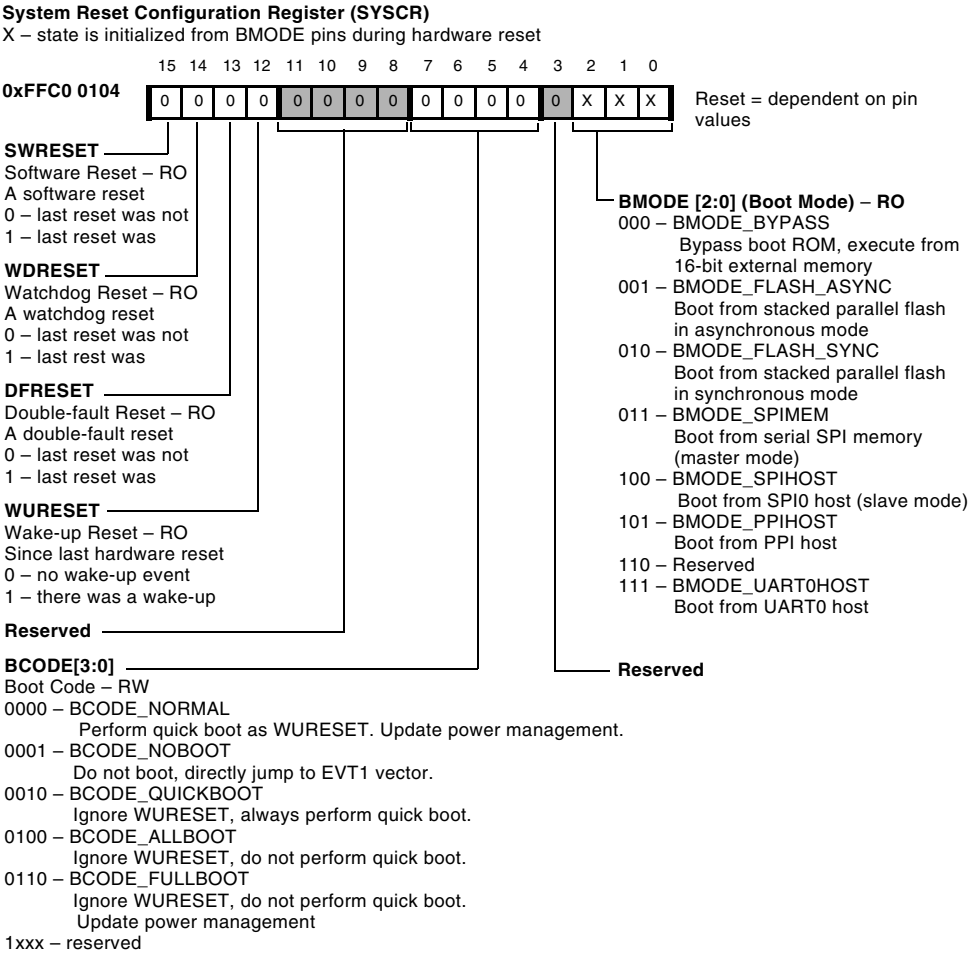


Figure 24-22. System Reset Configuration Register

The values sensed from the BMODE[2:0] pins are mirrored into the system reset configuration register (SYSCR). The values are available for software access and modification after the hardware reset sequence. Software can modify only bits[7:4] in this register to customize boot processing upon a software reset.

The WURESET indicates whether there was a wake up from hibernate since the last hardware reset. The bit cannot be cleared by software.

The bits [15:13] are exact copies of the same bits in the SWRST register. Unlike the SWRST register, SYSCR can be read without clearing these bits. Reading SWRST also causes SYSCR[15:13] to clear.

Boot Code Revision Control (BK_REVISION)

The boot ROM reserves the 32-bits at address 0xEF00 0040 for a four byte version code as shown in Figure 24-23.

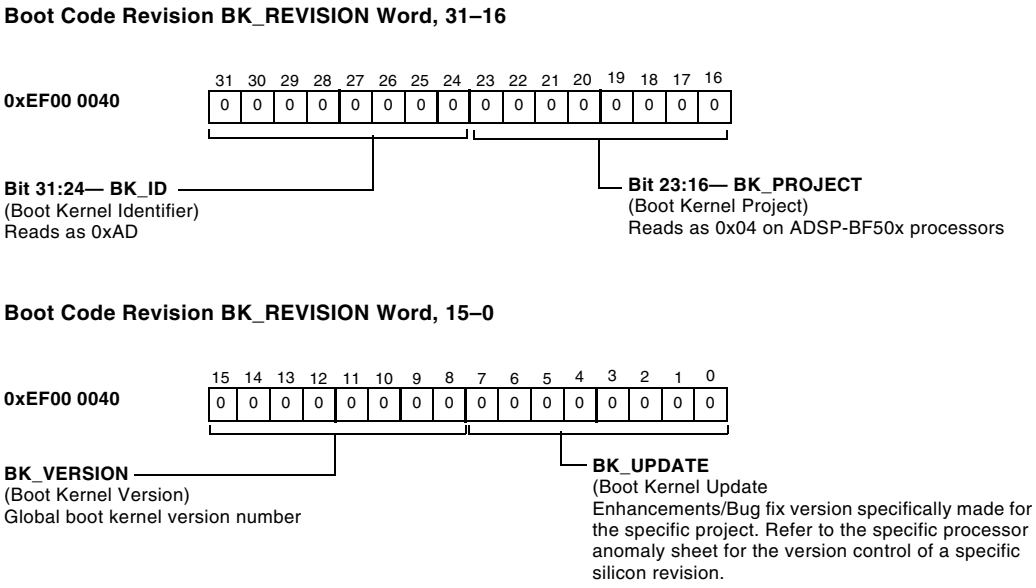


Figure 24-23. Boot Code Revision Code (BK_REVISION)

Boot Code Date Code (BK_DATECODE)

The boot ROM reserves the 32-bits at address 0xEF00 0050 for the build date as shown in Figure 24-24.

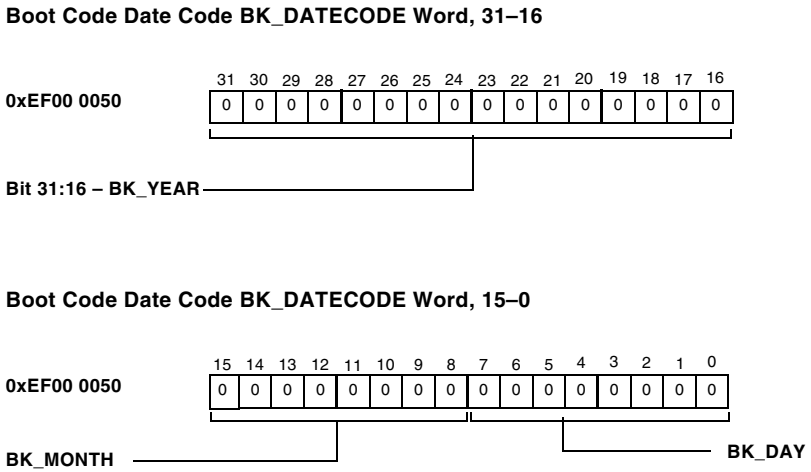


Figure 24-24. Boot Code Date Code (BK_DATECODE)

Zero Word (BK_ZEROS)

The boot ROM reserves the 32-bits at address 0xEF00 0048 which always reads as 0x0000 000 as shown in Figure 24-25.

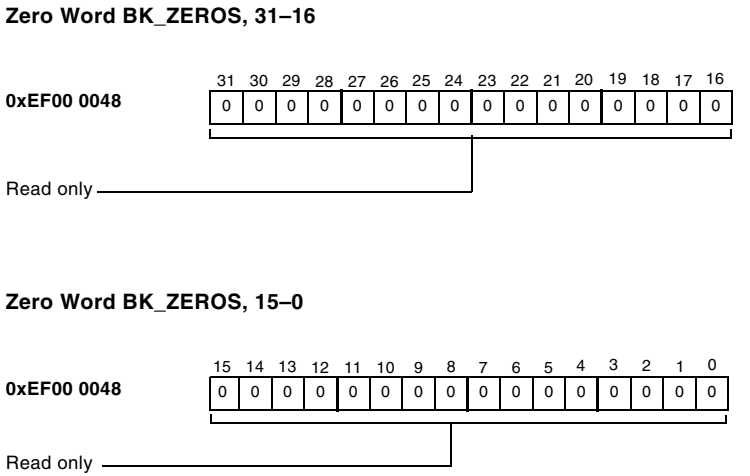


Figure 24-25. Zero Word (BK_ZEROS)

Ones Word (BK_ONES)

The boot ROM reserves the 32-bits at address 0xEF00 004C which always reads 0xFFFF FFFF as shown in Figure 24-26.

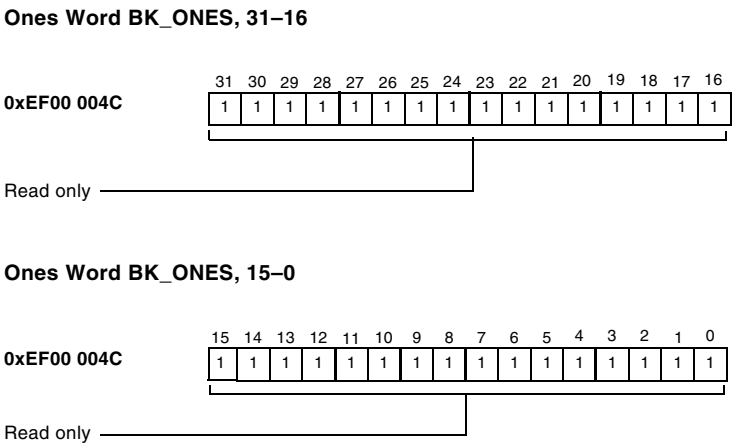


Figure 24-26. Ones Word (BK_ONES)

Data Structures

The boot kernel uses specific data structures for internal processing. Advanced users can customize the booting process by changing the content of the structure within the initcode routines. This section uses C language definitions for documentation purposes. VisualDSP++ users can use these structures directly in assembly programs by using the `.IMPORT` directive. The structures are supplied by the `bfrom.h` header file in your VisualDSP++ installation directory.

ADI_BOOT_HEADER

```
typedef struct {
    s32  dBlockCode;
    void* pTargetAddress;
    s32  dByteCount;
    s32  dArgument;
} ADI_BOOT_HEADER;
```

The structure `ADI_BOOT_HEADER` is used by the boot kernel to load and process a block header.

Every block header is loaded to L1 data memory location 0xFF80 7FF0–0xFF80 7FFF first or where `pHeader` points to. There it is analyzed by the boot kernel.

ADI_BOOT_BUFFER

```
typedef struct {
    void*  pSource;
    s32    dByteCount;
} ADI_BOOT_BUFFER;
```

The structure `ADI_BOOT_BUFFER` is used for any kind of buffer. For the user, this structure is important when implementing advanced callback mechanisms.

ADI_BOOT_DATA

```
typedef struct {
    void*  pSource;
    void*  pDestination;
    s16*   pControlRegister;
    s16*   pDmaControlRegister;
    s32    dControlValue;
```

```
    s32    dByteCount;
    s32    dFlags;
    s16    uwDataWidth;
    s16    uwSrcModifyMult;
    s16    uwDstModifyMult;
    s16    uwHwait;
    s16    uwSsel;
    s16    uwUserShort;
    s32    dUserLong;
    s32    dReserved;

    ADI_BOOT_ERROR_FUNC*  pErrorFunction;
    ADI_BOOT_LOAD_FUNC*  pLoadFunction;
    ADI_BOOT_CALLBACK_FUNC*  pCallBackFunction;
    ADI_BOOT_HEADER* pHeader;
    void*  pTempBuffer;
    void*  pTempCurrent;
    s32    dTempByteCount;
    s32    dBlockCount;
    s32    dClock;
    void*  pLogBuffer;
    void*  pLogCurrent;
    s32    dLogByteCount;
} ADI_BOOT_DATA;
```

The structure `ADI_BOOT_DATA` is the main data structure. A pointer to a `ADI_BOOT_DATA` structure is passed to most complex subroutines, including load functions, initcode, and callback routines. The structure has two parts. While the first is closely related to internal memory load routines, the second provides access to global boot settings.

Table 24-10 describes the data structures.

Table 24-10. Structure Variables, ADI_BOOT_DATA

Variable	Description
pSource	In the context of the boot kernel, the pSource pointer points either to the start address of the entire boot stream or to the header of the next boot block. In the context of memory load routines pSource points to the source address of the DMA work unit.
pDestination	The pDestination pointer is only used in memory load routines. It points to the destination address of the DMA work unit. It points to either 0xFF80 7FF0 when a header is loaded, or the target address when the payload data is loaded.
pControlRegister	This pointer holds the MMR address of the peripheral's main control register (for example UARTx_LCR or SPIx_CTL)
pDmaControlRegister	This pointer holds the MMR address of the DMAx_CONFIG register for the DMA channel in use.
dControlValue	The lower 16 bits of this value are written to the pControlRegister location each time a DMA work unit is started.
dByteCount	Number of bytes to be transferred.
dFlags	The lower 16 bits of this variable hold the lower 16 bits of the current block code. The upper 16 bits hold global flags. See "dFlags Word" on page 24-71.
uwDataWidth	This instructs the memory load routine to use: 0 – 8-bit DMA 1 – 16-bit DMA 2 – 32-bit DMA
uwSrcModifyMult	This is the multiplication factor used by the DMA source. A value of 1 sets the source modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwDstModifyMult	This is the multiplication factor used by the DMA destination. A value of 1 sets the destination modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwHwait	This 16-bit value holds the GPIO used for HWAIT signaling. The PG3 pin is configured as HWAIT signal on ADSP-BF50x processors. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.

Table 24-10. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
uwSsel	This 16-bit value holds the GPIO used for SPI slave select. The PF13 pin is configured as SPI slave select signal on ADSP-BF50x processors. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.
uwUserShort	The programmer can use this 16-bit value for passing parameters between modules of a customized booting scheme.
dUserLong	The programmer can use this 32-bit value for passing parameters between modules of a customized booting scheme.
dReserved	This 32-bit value is reserved for future development.
pErrorFunction	This is the pointer to the error handler. See "Error Handler" on page 24-31.
pLoadFunction	This is the pointer to the function responsible for loading data. See "Load Functions" on page 24-32
pCallBackFunction;	This is the pointer to the callback function. See "Callback Routines" on page 24-29.
pHeader	The pHeader pointer holds the address for intermediate storage of the block header. By default this value is set to 0xFF80 7FF0.
pTempBuffer	This pointer tells the boot kernel what memory to use for intermediate storage when the BFLAG_INDIRECT flag is set for a given block. The pointer defaults to 0xFF80 7F00. The value can be modified by the initcode routine, but there would be some impact to the VisualDSP++ tools.
pTempCurrent	Defaults to the pTempBuffer value. A load function can modify this value to manipulate subsequent callback and memory DMA routines.
dTempByteCount	This is the size of the intermediate storage buffer used when the BFLAG_INDIRECT flag is set for a given block. This value defaults to 256 and can be modified by an initcode routine. When increasing this value, the pTempBuffer must also be changed since by default the block is at the end of a physical data memory block.
dBlockCount	This 32-bit variable counts the boot blocks that are processed by the boot kernel. If the user sets this value to a negative value, the boot kernel exits when the variable increments to zero.

Table 24-10. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
dClock	The dClock variable holds information about the clock divider used by individual (serial) boot modes.
pLogBuffer	Pointer to the circular log buffer. By default the log buffer resides in L1 scratch pad memory at address 0xFFB0 0400.
pLogCurrent	Pointer to the next free entry of the circular log buffer.
dLogByteCount	Size of the circular log buffer, default is 0x400 bytes.

dFlags Word

Figure 24-27 and Figure 24-28 describe the `dFlags` word. `dFlags [15-0]` is a copy of Block Code[15–0] of the block currently being processed.

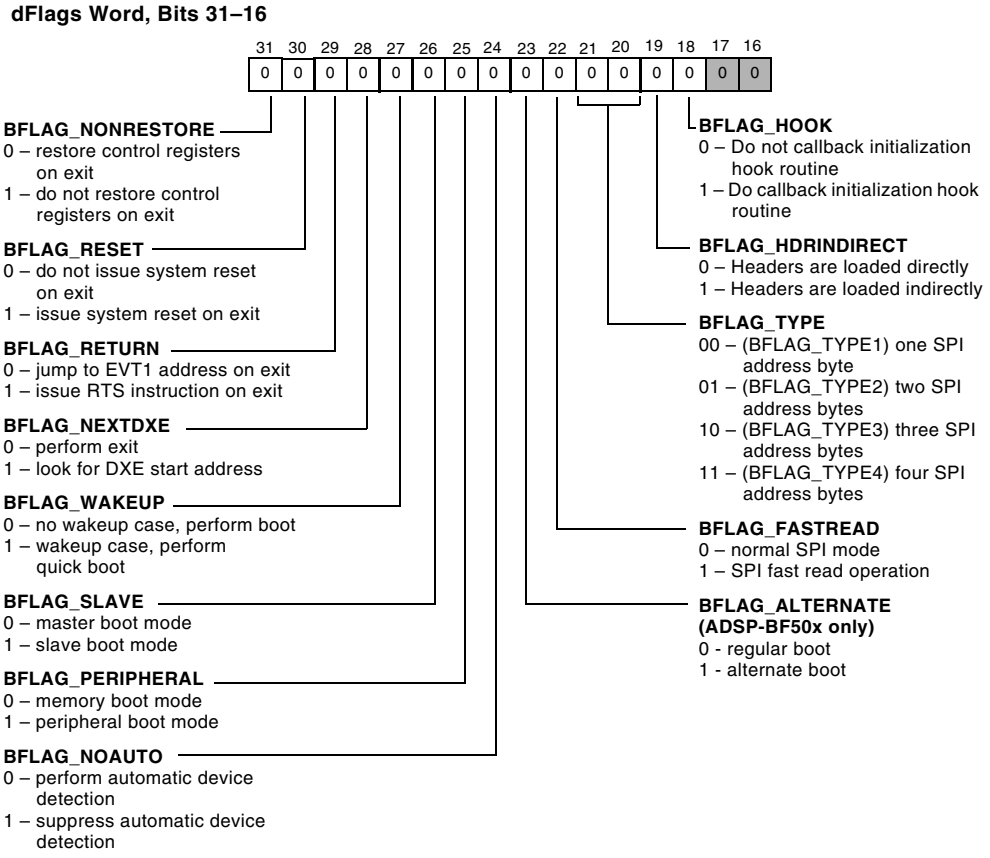


Figure 24-27. dFlags Word (Bits 31–16)

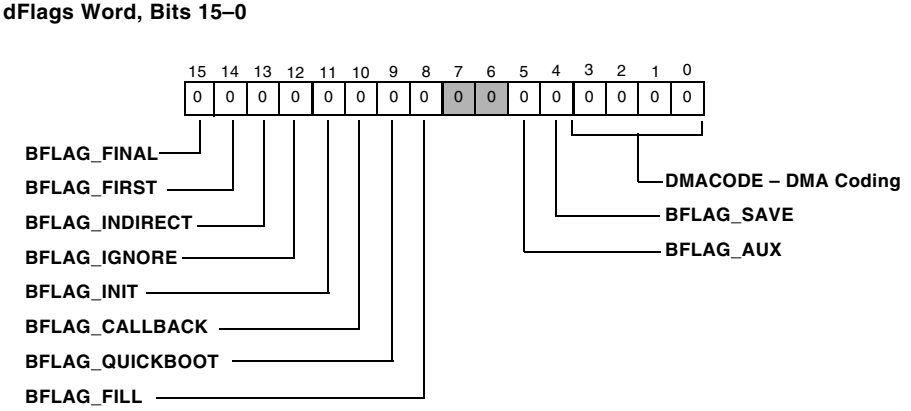


Figure 24-28. dFlags Word (Bits 15–0)

Callable ROM Functions for Booting

The following functions support boot management.

BFROM_FINALINIT

Entry address:

0xEF00 0002

Arguments:

no arguments

C prototype:

```
void bfrom_FinalInit (void);
```

The `bfrom_FinalInit` function never returns. It only executes a `JUMP` to the address stored in `EVT1`.

BFROM_PDMA

Entry address:

0xEF00 0004

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_PDma (ADI_BOOT_DATA *p);
```

This is the load function for peripherals such as SPI and UART that support DMA in their boot modes.

BFROM_MDMA

Entry address:

0xEF00 0006

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_MDma (ADI_BOOT_DATA *p);
```

This is the load function used for memory boot modes. This routine is also reused when the `BFLAG_FILL` or the `BFLAG_INDIRECT` flags are specified.

BFROM_MEMBOOT

Entry address:

0xEF00 0008

Arguments:

pointer to boot stream in `R0`, `dFlags` in `R1`, `dBlockCount` in `R2`, `pCallHook` passed over the stack in [FP+0x14], and updated block count returned in `R0`

C prototype:

```
s32 bfrom_MemBoot (
    void* pBootStream,
    s32 dFlags,
    s32 dBlockCount,
    ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes any boot stream that maps to the Blackfin memory starting from address `pBootStream`.

To boot a new application that may overwrite the calling application, the `dFlags` word is usually zero. When done, the routine does not return, but jumps to the `EVT1` vector address. If the `BFLAG_RETURN` flag is set, an `RTS` is executed instead and the routine returns to the parent function. In this way, fractions of an application can be loaded.

If the `dBlockCount` parameter is zero or a positive value, all boot blocks are processed until the `BFLAG_FINAL` flag is detected. If `dBlockCount` is a negative value, the negative number represents the number of blocks to be booted. For example, –1 causes the kernel to return immediately, –2 processes only one block.

The routine returns the updated source address `pSource` of the boot stream (for example, the first unused address after the processed boot stream).

The BFLAG_NEXTDXE flag suppresses boot loading. The boot kernel steps through the boot stream by analyzing the next-DXE pointers (in the ARGUMENT field of a BFLAG_FIRST block) and jumping to the next DXE. Assuming that the boot image is a chained list of boot streams, the boot kernel returns the absolute start address of the requested boot stream. In this example, the start address of the third boot stream (DXE) in a flash device is returned.

```
bfrom_MemBoot(
    (void*)0x20000000,
    BFLAG_RETURN|BFLAG_NEXTDXE,
    -3,
    NULL);
```

In the above example, the routine would return 0x2000 0000 when dBlockCount was set to –1. If the parameter dBlockCount is zero or positive when used along with the BFLAG_NEXTDXE command, the kernel returns when the BFLAG_FIRST flag on a header in the next-DXE chain is not set.

If the BFLAG_HOOK switch is set, the memboot routine call (pCallHook routine) after the ADI_BOOT_DATA structure is filled with default values. It then can overrule the default settings of the structure.

The bfrom_MemBoot() uses both MDMA channel pairs. Respective wake-up bits must be set in the SIC_IWRx registers.

BFROM_SPIBOOT

Entry address:

0xEF00 000A

Arguments:

- SPI address in R0
- dFlags in R1
- dBlockCount in R2
- pCallHook passed over the stack in [FP+0x14]
- updated block count returned in R0

C prototype:

```
s32 bfrom_SpiBoot (
    s32 dSpiAddress,
    s32 dFlags,
    s32 dBlockCount,
    ADI_BOOT_HOOK_FUNC* pCallHook);
```

This SPI master boot routine processes boot streams residing in SPI memories, using the SPI0 controller. The fourth argument pCallHook is passed over the stack. It provides a hook to call a callback routine after the ADI_BOOT_DATA structure is filled with default values. For example, the pCallHook routine may overwrite the default value of the uwSsel value in the ADI_BOOT_DATA structure. The coding follows the rules of uwHWAIT (see "Boot Host Wait (HWAIT) Feedback Strobe" on page 24-18). A value of 0x060D represents GPIO PF13 (SPI0_SSEL1).

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SLAVE bit. The BFLAG_NOAUTO flag instructs the system to skip the SPI device detection

routine. The BFLAG_TYPE then tells the boot kernel what addressing mode is required for the SPI memory. (See "SPI Device Detection Routine" on page 24-48.) The BFLAG_FASTREAD flag controls whether standard SPI read (0x3 command) or fast read (0xB) is performed. The three lower bits of the dFlags word are translated by the boot kernel into specific values to the SPI0_BAUD registers. This follows the truth table shown in Table 24-9 on page 24-47.

When called with the BFLAG_ALTERNATE flag, the bfrom_SpiBoot() function attempts to boot from external SPI memory device. Unless the uwSsel variable in the ADI_BOOT_DATA structure is altered by a hook routine, the memory is expected to be connected to $\overline{\text{SPI0\_SSEL1}}$. A pull-up resistor on this signal is required when automatic device detection is desired.

The bfrom_SpiBoot() routine does not deal with port muxing at all. When a part has been booted via SPI master mode after reset, the port muxing configuration is typically already ready for a runtime call to the bfrom_SpiBoot() routine. Otherwise ensure that the SPI0MISO, SPI0MOSI and SPI0SCK signals are properly activated in the PORTx_FER and PORTx_MUX registers. The $\overline{\text{SPI0\_SSEL1}}$ signal requires, however, that the respective PORTx_FER bit be cleared, as the boot kernel toggles the signal in GPIO mode.

Similarly, the user shall set the PF13 bit in the PORTF_FER register when booting from an external device.

The bfrom_SpiBoot() routine uses the MDMA0 memory DMA channel pair and the DMA7 peripheral DMA. Respective wake-up bits must be set in the SIC_IWRx registers. If a different peripheral DMA channel has been assigned to the SPI0 controller, use the hook routine to store the MMR address of the respective DMAx_CONFIG register into the pDmaControlRegister variable in the ADI_BOOT_DATA structure. Similarly, when using a different SPI controller than SPI0, write the MMR address of the relevant SPIx_CTL register into the pControlRegister variable.

BFROM_BOOTKERNEL

Entry address:

0xEF00 0020

Arguments:

- pointer to `ADI_BOOT_DATA` in `R0`
- returns updated source address `pSource` in `R0`

C prototype:

```
s32 bfrom_BootKernel (
    ADI_BOOT_DATA *p);
```

This ROM entry provides access to the raw boot kernel routine. It is the user's responsibility to initialize the items passed in the `ADI_BOOT_DATA` structure. Pay particular attention that the function pointers (`pLoadFunction`, and `pErrorFunction`) point to functional routines.

BFROM_CRC32

Entry address:

0xEF00 0030

Arguments:

- pointer to look-up table in `R0`
- pointer to data in `R1`
- `dByteCount` in `R2`
- initial CRC value in `R0`
- CRC value returned in `R0`

C prototype:

```
s32 bfrom_Crc32 (
    s32 *pLut,
    void *pData,
    s32 dByteCount,
    s32 dInitial);
```

This routine calculates the CRC32 checksum for a given array of bytes. The look-up table is typically generated by the `BFROM_CRC32POLY` routine. During the boot process this routine is called by the `BFROM_CRC32CALLBACK` routine. The `dInitial` value is normally set to zero unless the CRC32 routine is called in multiple slices. Then, the `dInitial` parameter expects the result of the former run.

BFROM_CRC32POLY

Entry address:

0xEF00 0032

Arguments:

- pointer to look-up table in `R0`
- polynomial in `R1`
- updated block count returned in `R0`

C prototype:

```
s32 bfrom_Crc32Poly (
    unsigned s32 *pLut,
    s32 dPolynomial);
```

This function generates a 1024-byte look-up table from a given CRC polynomial. During the boot process this routine is hidden by the `BFROM_CRC32INITCODE` routine.

BFROM_CRC32CALLBACK

Entry address:

0xEF00 0034

Arguments:

- pointer to `ADI_BOOT_DATA` in `R0`
- pointer to `ADI_BOOT_BUFFER` in `R1* Callback Flags in R2`

C prototype:

```
s32 bfrom_Crc32Callback (
    ADI_BOOT_DATA *pBS,
    ADI_BOOT_BUFFER *pCS,
    s32 dCbFlags);
```

This is a wrapper function that ensures the `BFROM_CRC32` subroutine fits into the boot process.

BFROM_CRC32INITCODE

Entry address:

0xEF00 0036

Arguments:

pointer to

`ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_Crc32Initcode (
    ADI_BOOT_DATA *p);
```

This is an initcode residing in ROM with the following jobs:

- Register `BFROM_CRC32CALLBACK` as a callback routine to the `pCallback` pointer in `ADI_BOOT_DATA`.
- Call `BFROM_CRC32POLY` to generate the look-up table.

This function is unlikely to be called by user code directly. This function is called as an initcode during the boot process when the CRC calculation is desired. See "CRC Checksum Calculation" on page 24-32 for details.

Programming Examples

This section provides programming examples that demonstrate a number of system reset and booting techniques.

Example System Reset

To perform a system reset, use the code shown in Listing 24-1 or Listing 24-2.

Listing 24-1. System Reset in Assembly

```
#include <blackfin.h>
P0.L = LO(BFROM_SYSCONTROL);
P0.H = HI(BFROM_SYSCONTROL);
R0.L = LO(SYSCTRL_SYSRESET);
R0.H = HI(SYSCTRL_SYSRESET);
R1 = 0;
R2 = 0;
CALL (P0);
```

Listing 24-2. System Reset in C Language

```
bfrom_SysControl(
    SYSCTRL_SYSRESET,
    0,
    NULL);
```

Example Exiting Reset to User Mode

To exit reset while remaining in user mode, use the code shown in Listing 24-3.

Listing 24-3. Exiting Reset to User Mode

```
_reset:   P1.L = LO(_usercode);
            /* Point to start of user code */
     P1.H = HI(_usercode);
RETI = P1;  /* Load address of _start into RETI */
   RTI;     /* Exit reset priority */
_reset.end:
_usercode:  /* Place user code here */
...
```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and `EVTx` registers are initialized here.

Example Exiting Reset to Supervisor Mode

To exit reset while remaining in supervisor mode, use the code shown in Listing 24-4.

Listing 24-4. Exiting Reset by Staying in Supervisor Mode

```
_reset:
   P0.L = LO(EVT15);
  /* Point to IVG15 in Event Vector Table */
   P0.H = HI(EVT15);
   P1.L = LO(_isr_IVG15); /* Point to start of IVG15 code */
   P1.H = HI(_isr_IVG15);
   [P0] = P1;          /* Initialize interrupt vector EVT15 */
   P0.L = LO(IMASK);   /* read-modify-write IMASK register */
   R0 = [P0];          /* to enable IVG15 interrupts */
   R1 = EVT_IVG15 (Z);
   R0 = R0 | R1;      /* set IVG15 bit */
   [P0] = R0;         /* write back to IMASK */
   RAISE 15;          /* generate IVG15 interrupt request */
                      /* IVG 15 is not served until reset
                         handler returns */
   P0.L = LO(_usercode);
   P0.H = HI(_usercode);
   RETI = P0;         /* RETI loaded with return address */
   RTI;               /* Return from Reset Event */
_reset.end:
_usercode:             /* Wait in user mode till IVG15 */
   JUMP _usercode;     /* interrupt is serviced */
_isr_IVG15:            /* IVG15 vectors here due to EVT15 */
...
```

Example Power Management with Initcode

Listing 24-5 and Listing 24-6 show how to change PLL and the voltage regulator within an initcode.

The ADSP-BF50x processors do not have an on-chip voltage regulator. Set the `bfrom_SysControl` option to `SYSCTRL_EXTVOLTAGE`.

Listing 24-5. Changing PLL and Voltage Regulator in C Language

```
#include <ccblkfn.h>
#include <bfrom.h>
void init_DPM(ADI_BOOT_DATA* pBS)
{
    ADI_SYSCTRL_VALUES init_DPM;
    init_DPM.uwPllCtl = SET_MSEL(12);
    init_DPM.uwPllDiv = (SET_SSEL(4) | CSEL_DIV1);
    init_DPM.uwPllLockCnt = 0x0200;
    bfrom_SysControl(SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
    SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT | SYSCTRL_WRITE, &init_DPM,
NULL);
}
```

Listing 24-6. Changing PLL and Voltage Regulator in Assembly

```
#include <blackfin.h>
#include <bfrom.h>
.import "bfrom.h";
/* Load Immediate 32-bit value into data or address register */
#define IMM32(reg,val) reg##.H=hi(val); reg##.L=lo(val)
.SECTION L1_code;
init_DPM:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:5);
SP += -12;
R0.L = SET_MSEL(12);
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0.L = (SET_SSEL(4) | CSEL_DIV1);
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;
R0.L = 0x0200;
```

```
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;
R0 = (SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_LOCKCNT | SYSCTRL_WRITE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P5,BFROM_SYSCONTROL);
call(P5);
SP += 12;
(R7:0,P5:5) = [SP++];
unlink;
rts;
init_DPM.end:
```

Care must be taken that the reprogramming of the PLL does not break the communication with the booting host. For example, in the case of UART boot, the `UARTx_DLL` and `UARTx_DLH` registers must be updated to keep the old bit rate.

Example XOR Checksum

Listing 24-7 illustrates how an initcode can be used to register a callback routine. The routine is called after each boot block that has the `BFLAG_CALLBACK` flag set. The calculated XOR checksum is compared against the block header `ARGUMENT` field. When the checksum fails, this example goes into idle mode. Otherwise control is returned to the boot kernel.

Since this callback example accesses the data after it is loaded, it would fail if the target address were in L1 instruction space. Therefore the `BFLAG_INDIRECT` flag should also be set. The `xor_callback` routine could then perform the checksum calculation at an intermediate storage place. The boot kernel transfers the data from the temporary buffer to the final destination after the callback routine returns.

In general, the block size is bigger than the size of the temporary buffer. Therefore, the boot kernel may need to divide the processing of a single block into multiple steps. The callback routine may also need to be invoked multiple times—every time the temporary buffer is filled up and once for the remaining bytes. The boot kernel passes the `dFlags` parameter, so that the callback routines knows whether it is called the first time, the last time or neither. The `dUserLong` variable in the `ADI_BOOT_DATA` structure is used to store the intermediate results between function calls.

Listing 24-7. XOR Checksum

```
s32 xor_callback(
    ADI_BOOT_DATA* pBS,
    ADI_BOOT_BUFFER* pCS,
    s32 dFlags)
{
    s32 i;
    if ((pCS!= NULL) && (pBS->pHeader!= NULL)) {
        if (dFlags & CBFLAG_FIRST) {
            pBS->dUserLong = 0;
        }
        for (i=0; i<pCS->dByteCount/sizeof(s32); i++) {
            pBS->dUserLong^= ((s32 *)pCS->pSource)[i];
        }
        if (dFlags & CBFLAG_FINAL) {
            if (pBS->dUserLong!= pBS->pHeader->dArgument) {
                idle ();
            }
        }
    }
    return 0;
}
void xor_initcode (ADI_BOOT_DATA *pBS)
{
```

```
    pBS->pCallBackFunction = xor_callback;
}
```

Note that the callback routine is not volatile. It should not be overwritten by subsequent boot blocks. It can, however, be overwritten after processing the last block with `BFLAG_CALLBACK` flag set.

The checksum algorithm must be booted first and cannot protect itself. Problems can be avoided by letting initcode and callback execute directly from off-chip flash memory. The ADSP-BF50x processors provide a CRC32 checksum algorithm in the on-chip L1 instruction ROM, that can be used for booting under this scenario. For more information see "CRC Checksum Calculation" on page 24-32.

Example Direct Code Execution

This code example illustrates how to instruct the VisualDSP++ tools to generate a flash image that causes the boot kernel to start code execution at flash address 0x2000 0020 rather than performing a regular boot. See "Direct Code Execution" on page 24-21.

First, a 32-byte data block is defined in an assembly file that contains the initial block.

```
.section bootblock;
.global _firstblock;
.var _firstblock[4] = 0xAD7BD006,
                      0x20000020,
                      0x00000010,
                      0x00000010;
```

Then, the linker is instructed to map the initial block to address 0x2000 0000 in the LDF file.

```
MEMORY
{
    MEM_ASYNC0
    {
            START(0x20000000)
            END(0x23FFFFFF)
            TYPE(ROM)
    WIDTH(8)
}
}
PROCESSOR p0
{
    RESOLVE(_firstblock,0x20000000)
    RESOLVE(start,0x20000020)
    KEEP(start,_firstblock)
    SECTIONS
    {
        flash
        {
            INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
            INPUT_SECTIONS($OBJECTS(bootblock))
} >MEM_ASYNC0
}
}
```

To invoke the elfloader utility, activate the meminit feature and use the command-line switches `-romsplitter` and `-maskaddr`. Refer to the application note *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)* for further details.

25 SYSTEM DESIGN

This chapter provides hardware, software and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to the processor data sheet for pin information, including pin numbers.

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's `CLKIN` pin. It is not possible to halt, change, or operate `CLKIN` below the specified frequency during normal operation. The processor uses the clock input (`CLKIN`) to generate on-chip clocks. These include the core clock (`CCLK`) and the peripheral clock (`SCLK`).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 16, “Dynamic Power Management”.

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the system interrupt assignment registers (SIC_IARx). For more information, see the *System Interrupts* chapter.

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, please see the *Blackfin Processor Programming Reference*.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the test and set byte (atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 17-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 25-1. Query Semaphore

```
/* Query semaphore. Denotes “Busy” if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread,
and all other contending threads are postponed because semaphore
value at [P0] is nonzero. Current thread could write thread_id to
semaphore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an
operating system call to reschedule the current thread. */
```

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to the *Chip Bus Hierarchy* chapter.

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to the *Chip Bus Hierarchy* chapter.

High-Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Capacitive loading and signal length of buses can be reduced by using a buffer for devices that operate with wait states (for example, SDRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

Use simple signal integrity methods to prevent transmission line reflections that may cause extraneous extra clock and sync signals. Additionally, avoid overshoot and undershoot that can cause long term damage to input pins.

Some signals are especially critical for short trace length and usually require series termination. The CLKIN pin should have impedance matching series resistance at its driver. SPORT interface signals TCLK, RCLK, RFS, and TFS should use some termination. Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers often require resistive termination located at the source. (Note also that TFS and RFS should not be shorted in multi-channel mode.) On the PPI interface, the PPI_CLK and SYNC signals also benefit from these standard signal integrity techniques. If these pins have multiple sources, it will be difficult to keep the traces short.

Adding termination to fix a problem on an existing board requires delays for new artwork and new boards. A transmission line simulator is recommended for critical signals. IBIS models are available from Analog Devices Inc. that will assist signal simulation software. Some signals can be corrected with a small zero or 22 ohm resistor located near the driver. The resistor value can be adjusted after measuring the signal at all endpoints.

For details, see the reference sources in "Recommended Reading" on page 17-13 for suggestions on transmission line termination.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the Printed Circuit Board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the `VDDEXT` and `VDDINT` pins of the package as shown in Figure 17-4. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Connect the power plane to the power supply pins directly with minimum trace length. A ground plane should be located near the component side of the board to reduce the distance that ground current must travel through vias. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced.

VDDINT is the highest frequency and requires special attention. Two things help power filtering above 100 MHz. First, capacitors should be physically small to reduce the inductance. Surface mount capacitors of size 0402 give better results than larger sizes. Secondly, lower values of capacitance will raise the resonant frequency of the LC circuit. While a cluster of 0.1µF is acceptable below 50 MHz, a mix of 0.1µF, 0.01µF, 0.001µF and even 100 pF is preferred in the 500 MHz range.

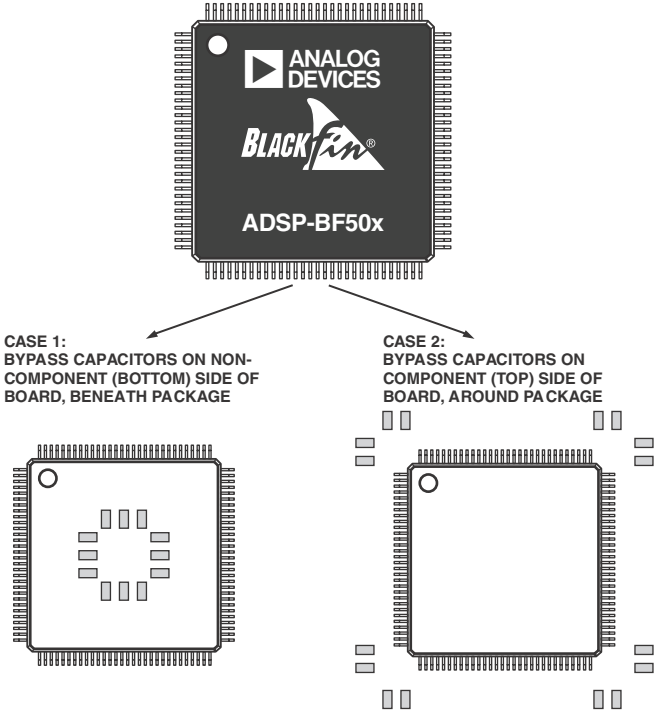


Figure 25-1. Bypass Capacitor Placement

Note that the instantaneous voltage on both internal and external power pins must at all times be within the recommended operating conditions as specified in the product data sheet. Local “bulk capacitance” (many microfarads) is also necessary. Although all capacitors should be kept close to the power consuming device, small capacitance values should be the closest and larger values may be placed further from the chip.

5 Volt Tolerance

Outputs that connect to inputs on 5 V devices can float or be pulled up to 5 V. Most Blackfin pins are not 5 V tolerant. There are a few exceptions such as the TWI pins. Level shifters are required on all other Blackfin pins to keep the pin voltage at or below absolute maximum ratings.

Test Point Access

The debug process is aided by test points on signals such as CLKOUT or SCLK, bank selects, PPICLK, and $\overline{\text{RESET}}$. If selection pins such as boot mode are connected directly to power or ground, they are inaccessible under a BGA chip. Use pull-up and pull-down resistors instead.

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent

source of information and practical ideas. Topics covered in the book include:

- High-speed properties of logic gates
- Measurement techniques
- Transmission lines
- Ground planes and layer stacking
- Terminations
- Vias
- Power systems
- Connectors
- Ribbon cables
- Clock distribution
- Clock oscillators

Consult your CAD software tools vendor. Some companies offer demonstration versions of signal integrity software. Simply by using their free software, you can learn:

- Transmission lines are real
- Un-terminated printed circuit board traces will ring and have overshoot and undershoot
- Simple termination will control signal integrity problems

Resetting the Processor

The reset pin requires a monotonic rise and fall. Therefore the pin should not be connected directly to an R/C time delay because such a circuit could be noise sensitive. In addition to the hardware reset mode provided via the RESET pin, the processor supports several software reset modes. For detailed information on the various modes, see *Blackfin Processor Programming Reference*. The processor state after reset is also described in the programming reference.

Recommendations for Unused Pins

Most often, there is no need to terminate unused pins, but the handful that do require termination are listed at the end of the pin list description section of the product data sheet.

Also note that unused peripherals may have separate power connections. These should be driven to the specified value.

Programmable Outputs

During power up, each GPIO pin is set to an input and any pins used in the system as an output should be connected to a pullup or pulldown resistor to maintain the desired state.

This would be particularly important in motor drive applications. It is also important for UART TX and RTS, SPI and serial TWI, or other communications interfaces. Some memory enable pull-ups may also be desired.

After the boot cycle, each GPIO pin may be set to input or output depending on ADSP-BF50x model number and the boot cycle chosen. The I/O / GPIO muxing of all pins may need to be reprogrammed to support the users application. Care should be taken for compatibility of function and state, before boot, during boot, and application pin usage.

Voltage Regulation Interface

ADSP-BF50x processors must use an external voltage regulator to power the V_{DDINT} domain. The EXT_WAKE and $\overline{PG}$ signals can facilitate communication with the external voltage regulator. EXT_WAKE is high-true for power-up and low only when the processor is in the hibernate state. EXT_WAKE may be connected directly to the low-true shut down input of many common regulators.

The $\overline{PG}$ (power-good, low-true) signal that allows the processor to start only after the internal voltage has reached a chosen level. In this way, the startup time of the external regulator will be detected after hibernation.

If the processor never will enter the hibernate state, the $\overline{PG}$ signal can be grounded in this mode. This will always indicate “power good”, meaning that V_{DDINT} is at a safe operating level. Any delay required at initial power-on, to guarantee a safe operating level for V_{DDINT}, will be provided by the RESET signal.

If the external regulator for V_{DDINT} has a power-good signal output, it can be used to help the processor recover properly from it's hibernate state. This signal may need to be inverted, as the processor's input should be low-true in order to indicate a "power good" condition.

If the external regulator does not have a power-good output, the $\overline{PG}$ signal should be driven to a fixed level (just below the desired operating voltage) so that the $\overline{PG}$ pin voltage can be compared to V_{DDINT} by the internal startup logic. This can be accomplished with an external resistor divider from V_{DDEXT} or any other fixed stable voltage. A divider with impedance of 1M Ohm is sufficient to supply current to this $\overline{PG}$ input. To save even more current during hibernation, the EXT_WAKE signal may be used as the voltage source to the divider. EXT_WAKE is low during hibernation, but will go high before the V_{DDINT} voltage is applied by the external regulator. In all cases, care should be taken to account for the min and max values of V_{DDEXT} or V_{OH} for EXT_WAKE. The voltage applied to the $\overline{PG}$ pin is used as the threshold that is compared internally to the rising value of V_{DDINT} to signal the processor to start. The voltage at $\overline{PG}$ should be calculated such that the V_{DDINT} value has risen to the desired voltage range for the application.

A SYSTEM MMR ASSIGNMENTS

This appendix lists MMR addresses and register names for all system registers on the ADSP-BF50x processors. Table A-1 groups the registers by function/peripheral and indicates the section later in this chapter where individual registers for that group are listed. The tables in the later sections cross reference to individual register diagrams located in the chapter where that register is described. The diagrams show individual bit descriptions for each register.

Table A-1. Register Tables in This Chapter

Function/Peripheral
"System Reset and Interrupt Control Registers" on page A-4
"DMA/Memory DMA Control Registers" on page A-5
"Ports Registers" on page A-8
"Timer Registers" on page A-11
"Core Timer Registers" on page A-3
"Watchdog Timer Registers" on page A-15
"GP Counter Registers" on page A-15
"Dynamic Power Management Registers" on page A-17
"Processor-Specific Memory Registers" on page A-2
"PPI Registers" on page A-17
"SPI Controller Registers" on page A-18
"SPORT Controller Registers" on page A-19
"UART Controller Registers" on page A-23
"TWI Registers" on page A-25

Table A-1. Register Tables in This Chapter (Cont'd)

Function/Peripheral
"CAN Registers" on page A-26
"ACM Registers" on page A-42
"PWM Registers" on page A-44
"RSI Registers" on page A-46

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF FFFF.
- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.
- All system MMR space that is not defined in this appendix is reserved for internal use only.

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in Table A-2.

Table A-2. Processor-Specific Memory Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFE0 0004	DMEM_CONTROL	"L1 Data Memory Control Register" on page 2-5
0xFFE0 0300	DTEST_COMMAND	"Data Test Command Register" on page 2-6

Table A-2. Processor-Specific Memory Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 0A00	EBIU_AMGCTL	Asynchronous Memory Global Control Register
0XFFC0 0A04	EBIU_AMBCTL	Asynchronous Memory Bank Control Register
0XFFC0 0A20	EBIU_MODE	Asynchronous Memory Mode Control Register
0XFFC0 0A24	EBIU_FCTL	Asynchronous Memory Parameter Control Register
0XFFC0 328C	FLASH_CONTROL	Stacked flash control register
0XFFC0 3290	FLASH_CONTROL_SET	Stacked flash control set register
0XFFC0 3294	FLASH_CONTROL_CLEAR	Stacked flash control clear register

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in Table A-3.

Table A-3. Core Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFE0 3000	TCNTL	"Core Timer Control Register (TCNTL)" on page 11-5
0xFFE0 3004	TPERIOD	"Core Timer Period Register (TPERIOD)" on page 11-6
0xFFE0 3008	TSCALE	"Core Timer Scale Register (TSCALE)" on page 11-7
0xFFE0 300C	TCOUNT	"Core Timer Count Register (TCOUNT)" on page 11-5

System Reset and Interrupt Control Registers

System reset and interrupt control registers (0xFFC0 0100 – 0xFFC0 01FF) are listed in Table A-4.

Table A-4. System Reset and Interrupt Control Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0104	SYSCR	"System Reset Configuration (SYSCR) Register" on page 24-60
0xFFC0 010C	SIC_IMASK0	"System Interrupt Mask (SIC_IMASK) Register" on page 4-12
0xFFC0 014C	SIC_IMASK1	"System Interrupt Mask (SIC_IMASK) Register" on page 4-12
0xFFC0 0110	SIC_IAR0	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0114	SIC_IAR1	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0118	SIC_IAR2	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 011C	SIC_IAR3	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0150	SIC_IAR4	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0154	SIC_IAR5	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0158	SIC_IAR6	"System Interrupt Assignment (SIC_IAR) Register" on page 4-11
0xFFC0 0120	SIC_ISR0	"System Interrupt Status (SIC_ISR) Register" on page 4-12
0xFFC0 0160	SIC_ISR1	"System Interrupt Status (SIC_ISR) Register" on page 4-12

Table A-4. System Reset and Interrupt Control Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0124	SIC_IWR0	"System Interrupt Wakeup-Enable (SIC_IWR) Register" on page 4-12
0xFFC0 0164	SIC_IWR1	"System Interrupt Wakeup-Enable (SIC_IWR) Register" on page 4-12

DMA/Memory DMA Control Registers

DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in Table A-5.

Table A-5. DMA Traffic Control Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0B0C	DMA_TC_PER	"DMA_TC_PER Register" on page 7-92
0xFFC0 0B10	DMA_TC_CNT	"DMA_TC_CNT Register" on page 7-92

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table A-6 and Table A-7. Table A-6 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table A-7 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA channel 0 Y_MODIFY register is called DMA0_Y_MODIFY, and its address is 0xFFC0 0C1C. Likewise, the memory DMA stream 0 source current address register is called MDMA_S0_CURR_ADDR, and its address is 0xFFC0 0E64.

Table A-6. DMA Channel Base Addresses

DMA Channel Identifier	MMR Base Address	Register Prefix
0	0xFFC0 0C00	DMA0_
1	0xFFC0 0C40	DMA1_
2	0xFFC0 0C80	DMA2_
3	0xFFC0 0CC0	DMA3_
4	0xFFC0 0D00	DMA4_
5	0xFFC0 0D40	DMA5_
6	0xFFC0 0D80	DMA6_
7	0xFFC0 0DC0	DMA7_
8	0xFFC0 0E00	DMA8_
9	0xFFC0 0E40	DMA9_
10	0xFFC0 0E80	DMA10_
11	0xFFC0 0EC0	DMA11_
MemDMA stream 0 destination	0xFFC0 0F00	MDMA_D0_
MemDMA stream 0 source	0xFFC0 0F40	MDMA_S0_
MemDMA stream 1 destination	0xFFC0 0F80	MDMA_D1_
MemDMA stream 1 source	0xFFC0 0FC0	MDMA_S1_

Table A-7. DMA Register Suffix and Offset

Register Suffix	Offset From Base	For individual bits, see this diagram:
NEXT_DESC_PTR	0x00	“DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR)” on page 7-81
START_ADDR	0x04	“DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR)” on page 7-75
CONFIG	0x08	“DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)” on page 7-68

Table A-7. DMA Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	For individual bits, see this diagram:
X_COUNT	0x10	"DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)" on page 7-76
X_MODIFY	0x14	"DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)" on page 7-78
Y_COUNT	0x18	"DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)" on page 7-79
Y_MODIFY	0x1C	"DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY)" on page 7-80
CURR_DESC_PTR	0x20	"DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)" on page 7-83
CURR_ADDR	0x24	"DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)" on page 7-76
IRQ_STATUS	0x28	"DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)" on page 7-72
PERIPHERAL_MAP	0x2C	"DMA Peripheral Map Registers (DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP)" on page 7-67
CURR_X_COUNT	0x30	"DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT)" on page 7-77
CURR_Y_COUNT	0x38	"DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)" on page 7-79

Ports Registers

Ports registers (port F: 0xFFC0 0700 – 0xFFC0 07FF, port G: 0xFFC0 1500 – 0xFFC0 15FF, port H: 0xFFC0 1700 – 0xFFC0 17FF, pin control: 0xFFC0 3200 – 0xFFC0 32FF) are listed in Table A-8.

Table A-8. Ports Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0700	PORTFIO	"GPIO Data Registers" on page 9-31
0xFFC0 0704	PORTFIO_CLEAR	"GPIO Clear Registers" on page 9-32
0xFFC0 0708	PORTFIO_SET	"GPIO Set Registers" on page 9-32
0xFFC0 070C	PORTFIO_TOGGLE	"GPIO Toggle Registers" on page 9-33
0xFFC0 0710	PORTFIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-35
0xFFC0 0714	PORTFIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-38
0xFFC0 0718	PORTFIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-36
0xFFC0 071C	PORTFIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-40
0xFFC0 0720	PORTFIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-35
0xFFC0 0724	PORTFIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-39
0xFFC0 0728	PORTFIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-37
0xFFC0 072C	PORTFIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-41
0xFFC0 0730	PORTFIO_DIR	"GPIO Direction Registers" on page 9-30
0xFFC0 0734	PORTFIO_POLAR	"GPIO Polarity Registers" on page 9-33
0xFFC0 0738	PORTFIO_EDGE	"Interrupt Sensitivity Registers" on page 9-34

Table A-8. Ports Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 073C	PORTFIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-34
0xFFC0 0740	PORTFIO_INEN	"GPIO Input Enable Registers" on page 9-31
0xFFC0 1500	PORTGIO	"GPIO Data Registers" on page 9-31
0xFFC0 1504	PORTGIO_CLEAR	"GPIO Clear Registers" on page 9-32
0xFFC0 1508	PORTGIO_SET	"GPIO Set Registers" on page 9-32
0xFFC0 150C	PORTGIO_TOGGLE	"GPIO Toggle Registers" on page 9-33
0xFFC0 1510	PORTGIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-35
0xFFC0 1514	PORTGIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-38
0xFFC0 1518	PORTGIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-36
0xFFC0 151C	PORTGIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-40
0xFFC0 1520	PORTGIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-35
0xFFC0 1524	PORTGIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-39
0xFFC0 1528	PORTGIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-37
0xFFC0 152C	PORTGIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-41
0xFFC0 1530	PORTGIO_DIR	"GPIO Direction Registers" on page 9-30
0xFFC0 1534	PORTGIO_POLAR	"GPIO Polarity Registers" on page 9-33
0xFFC0 1538	PORTGIO_EDGE	"Interrupt Sensitivity Registers" on page 9-34
0xFFC0 153C	PORTGIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-34
0xFFC0 1540	PORTGIO_INEN	"GPIO Input Enable Registers" on page 9-31

Table A-8. Ports Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1700	PORTHIO	"GPIO Data Registers" on page 9-31
0xFFC0 1704	PORTHIO_CLEAR	"GPIO Clear Registers" on page 9-32
0xFFC0 1708	PORTHIO_SET	"GPIO Set Registers" on page 9-32
0xFFC0 170C	PORTHIO_TOGGLE	"GPIO Toggle Registers" on page 9-33
0xFFC0 1710	PORTHIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-35
0xFFC0 1714	PORTHIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-38
0xFFC0 1718	PORTHIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-36
0xFFC0 171C	PORTHIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-40
0xFFC0 1720	PORTHIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-35
0xFFC0 1724	PORTHIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-39
0xFFC0 1728	PORTHIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-37
0xFFC0 172C	PORTHIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-41
0xFFC0 1730	PORTHIO_DIR	"GPIO Direction Registers" on page 9-30
0xFFC0 1734	PORTHIO_POLAR	"GPIO Polarity Registers" on page 9-33
0xFFC0 1738	PORTHIO_EDGE	"Interrupt Sensitivity Registers" on page 9-34
0xFFC0 173C	PORTHIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-34
0xFFC0 1740	PORTHIO_INEN	"GPIO Input Enable Registers" on page 9-31
0xFFC0 3200	PORTF_FER	"Function Enable Registers" on page 9-30
0xFFC0 3204	PORTG_FER	"Function Enable Registers" on page 9-30

Table A-8. Ports Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3208	PORTH_FER	"Function Enable Registers" on page 9-30
0xFFC0 3210	PORTF_MUX	"Port F Multiplexer Control Register" on page 9-27
0xFFC0 3214	PORTG_MUX	"Port F Multiplexer Control Register" on page 9-27
0xFFC0 3218	PORTH_MUX	"Port F Multiplexer Control Register" on page 9-27
0xFFC0 3240	PORTF_HYSTERESIS	"Port F Hysteresis Register" on page 9-24
0xFFC0 3244	PORTG_HYSTERESIS	"Port G Hysteresis Register" on page 9-25
0xFFC0 3248	PORTH_HYSTERESIS	"Port H Hysteresis Register" on page 9-25
0xFFC0 3280	NONGPIO_DRIVE	"Drive Strength Control" on page 9-26
0xFFC0 3288	NONGPIO_HYSTERESIS	"Non-GPIO Hysteresis Register" on page 9-26

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in Table A-9.

Table A-9. Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0600	TIMER0_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0604	TIMER0_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0608	TIMER0_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42

Table A-9. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 060C	TIMER0_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0610	TIMER1_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0614	TIMER1_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0618	TIMER1_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 061C	TIMER1_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0620	TIMER2_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0624	TIMER2_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0628	TIMER2_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 062C	TIMER2_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0630	TIMER3_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0634	TIMER3_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0638	TIMER3_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42

Table A-9. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 063C	TIMER3_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0640	TIMER4_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0644	TIMER4_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0648	TIMER4_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 064C	TIMER4_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0650	TIMER5_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0654	TIMER5_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0658	TIMER5_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 065C	TIMER5_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0660	TIMER6_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0664	TIMER6_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0668	TIMER6_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42

Table A-9. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 066C	TIMER6_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0670	TIMER7_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 10-40
0xFFC0 0674	TIMER7_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 10-41
0xFFC0 0678	TIMER7_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 067C	TIMER7_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 10-42
0xFFC0 0680	TIMER_ENABLE	"Timer Enable Register (TIMER_ENABLE)" on page 10-35
0xFFC0 0684	TIMER_DISABLE	"Timer Disable Register (TIMER_DISABLE)" on page 10-36
0xFFC0 0688	TIMER_STATUS	"Timer Status Register (TIMER_STATUS)" on page 10-38

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF) are listed in Table A-10.

Table A-10. Watchdog Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0200	WDOG_CTL	"Watchdog Control (WDOG_CTL) Register" on page 12-7
0xFFC0 0204	WDOG_CNT	"Watchdog Count (WDOG_CNT) Register" on page 12-5
0xFFC0 0208	WDOG_STAT	"Watchdog Status (WDOG_STAT) Register" on page 12-6

GP Counter Registers

GP Counter 0 registers (0xFFC0 3500 – 0xFFC0 351C) are listed in Table A-11, and GP Counter 1 registers (0xFFC0 3300 – 0xFFC0 331C) are listed in Table A-12.

Table A-11. GP Counter 0 Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3500	CNT0_CONFIG	"Counter Configuration Register (CNT_CONFIG)" on page 13-19
0xFFC0 3504	CNT0_IMASK	"Counter Interrupt Mask Register (CNT_IMASK)" on page 13-20
0xFFC0 3508	CNT0_STATUS	"Counter Status Register (CNT_STATUS)" on page 13-20
0xFFC0 350C	CNT0_COMMAND	"Counter Status Register (CNT_STATUS)" on page 13-20

Table A-11. GP Counter 0 Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3510	CNT0_DEBOUNCE	"Counter Debounce Register (CNT_DEBOUNCE)" on page 13-23
0xFFC0 3514	CNT0_COUNTER	"Counter Count Value Register (CNT_COUNTER)" on page 13-24
0xFFC0 3518	CNT0_MAX	"Counter Boundary Registers (CNT_MIN and CNT_MAX)" on page 13-25
0xFFC0 351C	CNT0_MIN	"Counter Boundary Registers (CNT_MIN and CNT_MAX)" on page 13-25

Table A-12. GP Counter 1 Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3300	CNT1_CONFIG	"Counter Configuration Register (CNT_CONFIG)" on page 13-19
0xFFC0 3304	CNT1_IMASK	"Counter Interrupt Mask Register (CNT_IMASK)" on page 13-20
0xFFC0 3308	CNT1_STATUS	"Counter Status Register (CNT_STATUS)" on page 13-20
0xFFC0 330C	CNT1_COMMAND	"Counter Status Register (CNT_STATUS)" on page 13-20
0xFFC0 3310	CNT1_DEBOUNCE	"Counter Debounce Register (CNT_DEBOUNCE)" on page 13-23
0xFFC0 33514	CNT1_COUNTER	"Counter Count Value Register (CNT_COUNTER)" on page 13-24
0xFFC0 3318	CNT1_MAX	"Counter Boundary Registers (CNT_MIN and CNT_MAX)" on page 13-25
0xFFC0 331C	CNT1_MIN	"Counter Boundary Registers (CNT_MIN and CNT_MAX)" on page 13-25

Dynamic Power Management Registers

Dynamic power management registers (0xFFC0 0000 – 0xFFC0 00FF) are listed in Table A-13.

Table A-13. Dynamic Power Management Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0000	PLL_CTL	"PLL Control Register" on page 8-22
0xFFC0 0004	PLL_DIV	"PLL Divide Register" on page 8-21
0xFFC0 0008	VR_CTL	"Voltage Regulator Control Register" on page 8-23
0xFFC0 000C	PLL_STAT	"PLL Status Register" on page 8-22
0xFFC0 0010	PLL_LOCKCNT	"PLL Lock Count Register" on page 8-23

PPI Registers

PPI registers (0xFFC0 1000 – 0xFFC0 10FF) are listed in Table A-14.

Table A-14. PPI Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1000	PPI_CONTROL	"PPI Control Register (PPI_CONTROL)" on page 20-25
0xFFC0 1004	PPI_STATUS	"PPI Status Register (PPI_STATUS)" on page 20-29
0xFFC0 1008	PPI_COUNT	"PPI Transfer Count Register (PPI_COUNT)" on page 20-32

Table A-14. PPI Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 100C	PPI_DELAY	"PPI Delay Count Register (PPI_DELAY)" on page 20-32
0xFFC0 1010	PPI_FRAME	"PPI Lines Per Frame Register (PPI_FRAME)" on page 20-33

SPI Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in Table A-15.

SPI1 controller registers (0xFFC0 3400 – 0xFFC0 34FF) are listed in Table A-16 on page A-19.

Table A-15. SPI0 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0500	SPI0_CTL	"SPI Control (SPI_CTL) Register" on page 18-35
0xFFC0 0504	SPI0_FLG	"SPI Flag (SPI_FLG) Register" on page 18-37
0xFFC0 0508	SPI0_STAT	"SPI Status (SPI_STAT) Register" on page 18-39
0xFFC0 050C	SPI0_TDBR	"SPI Transmit Data Buffer (SPI_TDBR) Register" on page 18-41
0xFFC0 0510	SPI0_RDBR	"SPI Receive Data Buffer (SPI_RDBR) Register" on page 18-42
0xFFC0 0514	SPI0_BAUD	"SPI Baud Rate (SPI_BAUD) Register" on page 18-34
0xFFC0 0518	SPI0_SHADOW	"SPI RDBR Shadow (SPI_SHADOW) Register" on page 18-43

Table A-16. SPI1 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3400	SPI1_CTL	"SPI Control (SPI_CTL) Register" on page 18-35
0xFFC0 3404	SPI1_FLG	"SPI Flag (SPI_FLG) Register" on page 18-37
0xFFC0 3408	SPI1_STAT	"SPI Status (SPI_STAT) Register" on page 18-39
0xFFC0 340C	SPI1_TDBR	"SPI Transmit Data Buffer (SPI_TDBR) Register" on page 18-41
0xFFC0 3410	SPI1_RDBR	"SPI Receive Data Buffer (SPI_RDBR) Register" on page 18-42
0xFFC0 3414	SPI1_BAUD	"SPI Baud Rate (SPI_BAUD) Register" on page 18-34
0xFFC0 3418	SPI1_SHADOW	"SPI RDBR Shadow (SPI_SHADOW) Register" on page 18-43

SPORT Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in Table A-17. SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-18 on page A-21.

Table A-17. SPORT0 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0800	SPORT0_TCR1	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 19-47
0xFFC0 0804	SPORT0_TCR2	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 19-47

Table A-17. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0808	SPORT0_TCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 19-63
0xFFC0 080C	SPORT0_TFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 19-64
0xFFC0 0810	SPORT0_TX	"SPORT Transmit Data (SPORT_TX) Register" on page 19-57
0xFFC0 0818	SPORT0_RX	"SPORT Receive Data (SPORT_RX) Register" on page 19-59
0xFFC0 0820	SPORT0_RCR1	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 19-52
0xFFC0 0824	SPORT0_RCR2	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 19-52
0xFFC0 0828	SPORT0_RCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 19-63
0xFFC0 082C	SPORT0_RFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 19-64
0xFFC0 0830	SPORT0_STAT	"SPORT Status (SPORT_STAT) Register" on page 19-62
0xFFC0 0834	SPORT0_CHNL	"SPORT Current Channel (SPORT_CHNL) Register" on page 19-66
0xFFC0 0838	SPORT0_MCMC1	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 19-65
0xFFC0 083C	SPORT0_MCMC2	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 19-65

Table A-17. SPORT0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0840	SPORT0_MTCS0	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 0844	SPORT0_MTCS1	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 0848	SPORT0_MTCS2	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 084C	SPORT0_MTCS3	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 0850	SPORT0_MRCS0	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 0854	SPORT0_MRCS1	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 0858	SPORT0_MRCS2	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 085C	SPORT0_MRCS3	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67

Table A-18. SPORT1 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0900	SPORT1_TCR1	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 19-47
0xFFC0 0904	SPORT1_TCR2	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 19-47
0xFFC0 0908	SPORT1_TCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 19-63

Table A-18. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 090C	SPORT1_TFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 19-64
0xFFC0 0910	SPORT1_TX	"SPORT Transmit Data (SPORT_TX) Register" on page 19-57
0xFFC0 0918	SPORT1_RX	"SPORT Receive Data (SPORT_RX) Register" on page 19-59
0xFFC0 0920	SPORT1_RCR1	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 19-52
0xFFC0 0924	SPORT1_RCR2	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 19-52
0xFFC0 0928	SPORT1_RCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 19-63
0xFFC0 092C	SPORT1_RFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 19-64
0xFFC0 0930	SPORT1_STAT	"SPORT Status (SPORT_STAT) Register" on page 19-62
0xFFC0 0934	SPORT1_CHNL	"SPORT Current Channel (SPORT_CHNL) Register" on page 19-66
0xFFC0 0938	SPORT1_MCMC1	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 19-65
0xFFC0 093C	SPORT1_MCMC2	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 19-65
0xFFC0 0940	SPORT1_MTCS0	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68

Table A-18. SPORT1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0944	SPORT1_MTCS1	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 0948	SPORT1_MTCS2	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 094C	SPORT1_MTCS3	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 19-68
0xFFC0 0950	SPORT1_MRCS0	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 0954	SPORT1_MRCS1	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 0958	SPORT1_MRCS2	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67
0xFFC0 095C	SPORT1_MRCS3	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 19-67

UART Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in Table A-19. UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in Table A-20.

Table A-19. UART0 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 0400	UART0_DLL	"UARTx_DLL and UARTx_DLH Registers" on page 15-43
0XFFC0 0404	UART0_DLH	"UARTx_DLL and UARTx_DLH Registers" on page 15-43
0XFFC0 0408	UART0_GCTL	"UARTx_GCTL Registers" on page 15-45

Table A-19. UART0 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 040C	UART0_LCR	"UARTx_LCR Registers" on page 15-28
0XFFC0 0410	UART0_MCR	"UARTx_MCR Registers" on page 15-31
0XFFC0 0414	UART0_LSR	"UARTx_LSR Registers" on page 15-33
0XFFC0 0418	UART0_MSR	"UARTx_MSR Registers" on page 15-36
0XFFC0 041C	UART0_SCR	"UARTx_SCR Registers" on page 15-44
0XFFC0 0420	UART0_IER_SET	"UARTx_IER_SET and UARTx_IER_CLEAR Registers" on page 15-39
0XFFC0 0424	UART0_IER_CLEAR	"UARTx_IER_SET and UARTx_IER_CLEAR Registers" on page 15-39
0XFFC0 0428	UART0_THR	"UARTx_THR Registers" on page 15-37
0XFFC0 042C	UART0_RBR	"UARTx_RBR Registers" on page 15-38

Table A-20. UART1 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2000	UART1_DLL	"UARTx_DLL and UARTx_DLH Registers" on page 15-43
0XFFC0 2004	UART1_DLH	"UARTx_DLL and UARTx_DLH Registers" on page 15-43
0XFFC0 2008	UART1_GCTL	"UARTx_GCTL Registers" on page 15-45
0XFFC0 200C	UART1_LCR	"UARTx_LCR Registers" on page 15-28
0XFFC0 2010	UART1_MCR	"UARTx_MCR Registers" on page 15-31
0XFFC0 2014	UART1_LSR	"UARTx_LSR Registers" on page 15-33
0XFFC0 2018	UART1_MSR	"UARTx_MSR Registers" on page 15-36
0XFFC0 201C	UART1_SCR	"UARTx_SCR Registers" on page 15-44
0XFFC0 2020	UART1_IER_SET	"UARTx_IER_SET and UARTx_IER_CLEAR Registers" on page 15-39

Table A-20. UART1 Controller Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2024	UART1_IER_CLEAR	"UARTx_IER_SET and UARTx_IER_CLEAR Registers" on page 15-39
0XFFC0 2028	UART1_THR	"UARTx_THR Registers" on page 15-37
0XFFC0 202C	UART1_RBR	"UARTx_RBR Registers" on page 15-38

TWI Registers

Two Wire Interface (TWI) registers (0xFFC0 1400 – 0xFFC0 14FF) are listed in Table A-21.

Table A-21. TWI Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1400	TWI_CLKDIV	"SCL Clock Divider Register (TWI_CLKDIV)" on page 16-25
0xFFC0 1404	TWI_CONTROL	"TWI CONTROL Register (TWI_CONTROL)" on page 16-24
0xFFC0 1408	TWI_SLAVE_CTL	"TWI Slave Mode Control Register (TWI_SLAVE_CTL)" on page 16-26
0xFFC0 140C	TWI_SLAVE_STAT	"TWI Slave Mode Status Register (TWI_SLAVE_STAT)" on page 16-28
0xFFC0 1410	TWI_SLAVE_ADDR	"TWI Slave Mode Address Register (TWI_SLAVE_ADDR)" on page 16-28
0xFFC0 1414	TWI_MASTER_CTL	"TWI Master Mode Control Register (TWI_MASTER_CTL)" on page 16-30
0xFFC0 1418	TWI_MASTER_STAT	"TWI Master Mode Status Register (TWI_MASTER_STAT)" on page 16-34
0xFFC0 141C	TWI_MASTER_ADDR	"TWI Master Mode Address Register (TWI_MASTER_ADDR)" on page 16-33

Table A-21. TWI Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1420	TWI_INT_STAT	"TWI Interrupt Status Register (TWI_INT_STAT)" on page 16-41
0xFFC0 1424	TWI_INT_MASK	"TWI Interrupt Mask Register (TWI_INT_MASK)" on page 16-40
0xFFC0 1428	TWI_FIFO_CTL	"TWI FIFO Control Register (TWI_FIFO_CTL)" on page 16-37
0xFFC0 142C	TWI_FIFO_STAT	"TWI FIFO Status Register (TWI_FIFO_STAT)" on page 16-39
0xFFC0 1480	TWI_XMT_DATA8	"TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)" on page 16-43
0xFFC0 1484	TWI_XMT_DATA16	"TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)" on page 16-44
0xFFC0 1488	TWI_RCV_DATA8	"TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)" on page 16-45
0xFFC0 148C	TWI_RCV_DATA16	"TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)" on page 16-46

CAN Registers

Controller Area Network (CAN) registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in Table A-22, Table A-23, Table A-24, and Table A-25.

Table A-22. CAN Mailbox Configuration 1 Registers (For Mailboxes 0-15)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2A00	CAN_MC1	Mailbox config reg 1
0XFFC0 2A04	CAN_MD1	Mailbox direction reg 1

Table A-22. CAN Mailbox Configuration 1 Registers (For Mailboxes 0-15) (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2A08	CAN_TRS1	Transmit Request Set reg 1
0XFFC0 2A0C	CAN_TRR1	Transmit Request Reset reg 1
0XFFC0 2A10	CAN_TA1	Transmit Acknowledge reg 1
0XFFC0 2A14	CAN_AA1	Transmit Abort Acknowledge reg 1
0XFFC0 2A18	CAN_RMP1	Receive Message Pending reg 1
0XFFC0 2A1C	CAN_RML1	Receive Message Lost reg 1
0XFFC0 2A20	CAN_MBTIF1	Mailbox Transmit Interrupt Flag reg 1
0XFFC0 2A24	CAN_MBRIF1	Mailbox Receive Interrupt Flag reg 1
0XFFC0 2A28	CAN_MBIM1	Mailbox Interrupt Mask reg 1
0XFFC0 2A2C	CAN_RFH1	Remote Frame Handling reg 1
0XFFC0 2A30	CAN_OPSS1	Overwrite Protection Single Shot Xmission reg 1

Table A-23. CAN Mailbox Configuration 2 Registers (For Mailboxes 16-31)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2A40	CAN_MC2	Mailbox config reg 2
0XFFC0 2A44	CAN_MD2	Mailbox direction reg 2
0XFFC0 2A48	CAN_TRS2	Transmit Request Set reg 2
0XFFC0 2A4C	CAN_TRR2	Transmit Request Reset reg 2
0XFFC0 2A50	CAN_TA2	Transmit Acknowledge reg 2
0XFFC0 2A54	CAN_AA2	Transmit Abort Acknowledge reg 2
0XFFC0 2A58	CAN_RMP2	Receive Message Pending reg 2
0XFFC0 2A5C	CAN_RML2	Receive Message Lost reg 2

Table A-23. CAN Mailbox Configuration 2 Registers (For Mailboxes 16-31) (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2A60	CAN_MBTIF2	Mailbox Transmit Interrupt Flag reg 2
0XFFC0 2A64	CAN_MBRIF2	Mailbox Receive Interrupt Flag reg 2
0XFFC0 2A68	CAN_MBIM2	Mailbox Interrupt Mask reg 2
0XFFC0 2A6C	CAN_RFH2	Remote Frame Handling reg 2
0XFFC0 2A70	CAN_OPSS2	Overwrite Protection Single Shot Xmission reg 2
0XFFC0 2A80	CAN_CLOCK	Bit Timing Configuration register 0
0XFFC0 2A84	CAN_TIMING	Bit Timing Configuration register 1
0XFFC0 2A88	CAN_DEBUG	Debug Register
0XFFC0 2A8C	CAN_STATUS	Global Status Register
0XFFC0 2A90	CAN_CEC	Error Counter Register
0XFFC0 2A94	CAN_GIS	Global Interrupt Status Register
0XFFC0 2A98	CAN_GIM	Global Interrupt Mask Register
0XFFC0 2A9C	CAN_GIF	Global Interrupt Flag Register
0XFFC0 2AA0	CAN_CONTROL	Master Control Register
0XFFC0 2AA4	CAN_INTR	Interrupt Pending Register
0XFFC0 2AAC	CAN_MBTD	Mailbox Temporary Disable Feature
0XFFC0 2AB0	CAN_EWR	Programmable Warning Level
0XFFC0 2AB4	CAN_ESR	Error Status Register
0XFFC0 2AC4	CAN_UCCNT	Universal Counter
0XFFC0 2AC8	CAN_UCRC	Universal Counter Reload/Capture Register
0XFFC0 2ACC	CAN_UCCNF	Universal Counter Configuration Register

Table A-24. CAN Mailbox Acceptance Mask Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2B00	CAN_AM00L	Mailbox 0 Low Acceptance Mask
0XFFC0 2B04	CAN_AM00H	Mailbox 0 High Acceptance Mask
0XFFC0 2B08	CAN_AM01L	Mailbox 1 Low Acceptance Mask
0XFFC0 2B0C	CAN_AM01H	Mailbox 1 High Acceptance Mask
0XFFC0 2B10	CAN_AM02L	Mailbox 2 Low Acceptance Mask
0XFFC0 2B14	CAN_AM02H	Mailbox 2 High Acceptance Mask
0XFFC0 2B18	CAN_AM03L	Mailbox 3 Low Acceptance Mask
0XFFC0 2B1C	CAN_AM03H	Mailbox 3 High Acceptance Mask
0XFFC0 2B20	CAN_AM04L	Mailbox 4 Low Acceptance Mask
0XFFC0 2B24	CAN_AM04H	Mailbox 4 High Acceptance Mask
0XFFC0 2B28	CAN_AM05L	Mailbox 5 Low Acceptance Mask
0XFFC0 2B2C	CAN_AM05H	Mailbox 5 High Acceptance Mask
0XFFC0 2B30	CAN_AM06L	Mailbox 6 Low Acceptance Mask
0XFFC0 2B34	CAN_AM06H	Mailbox 6 High Acceptance Mask
0XFFC0 2B38	CAN_AM07L	Mailbox 7 Low Acceptance Mask
0XFFC0 2B3C	CAN_AM07H	Mailbox 7 High Acceptance Mask
0XFFC0 2B40	CAN_AM08L	Mailbox 8 Low Acceptance Mask
0XFFC0 2B44	CAN_AM08H	Mailbox 8 High Acceptance Mask
0XFFC0 2B48	CAN_AM09L	Mailbox 9 Low Acceptance Mask
0XFFC0 2B4C	CAN_AM09H	Mailbox 9 High Acceptance Mask
0XFFC0 2B50	CAN_AM10L	Mailbox 10 Low Acceptance Mask
0XFFC0 2B54	CAN_AM10H	Mailbox 10 High Acceptance Mask
0XFFC0 2B58	CAN_AM11L	Mailbox 11 Low Acceptance Mask
0XFFC0 2B5C	CAN_AM11H	Mailbox 11 High Acceptance Mask
0XFFC0 2B60	CAN_AM12L	Mailbox 12 Low Acceptance Mask

Table A-24. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2B64	CAN_AM12H	Mailbox 12 High Acceptance Mask
0XFFC0 2B68	CAN_AM13L	Mailbox 13 Low Acceptance Mask
0XFFC0 2B6C	CAN_AM13H	Mailbox 13 High Acceptance Mask
0XFFC0 2B70	CAN_AM14L	Mailbox 14 Low Acceptance Mask
0XFFC0 2B74	CAN_AM14H	Mailbox 14 High Acceptance Mask
0XFFC0 2B78	CAN_AM15L	Mailbox 15 Low Acceptance Mask
0XFFC0 2B7C	CAN_AM15H	Mailbox 15 High Acceptance Mask
0XFFC0 2B80	CAN_AM16L	Mailbox 16 Low Acceptance Mask
0XFFC0 2B84	CAN_AM16H	Mailbox 16 High Acceptance Mask
0XFFC0 2B88	CAN_AM17L	Mailbox 17 Low Acceptance Mask
0XFFC0 2B8C	CAN_AM17H	Mailbox 17 High Acceptance Mask
0XFFC0 2B90	CAN_AM18L	Mailbox 18 Low Acceptance Mask
0XFFC0 2B94	CAN_AM18H	Mailbox 18 High Acceptance Mask
0XFFC0 2B98	CAN_AM19L	Mailbox 19 Low Acceptance Mask
0XFFC0 2B9C	CAN_AM19H	Mailbox 19 High Acceptance Mask
0XFFC0 2BA0	CAN_AM20L	Mailbox 20 Low Acceptance Mask
0XFFC0 2BA4	CAN_AM20H	Mailbox 20 High Acceptance Mask
0XFFC0 2BA8	CAN_AM21L	Mailbox 21 Low Acceptance Mask
0XFFC0 2BAC	CAN_AM21H	Mailbox 21 High Acceptance Mask
0XFFC0 2BB0	CAN_AM22L	Mailbox 22 Low Acceptance Mask
0XFFC0 2BB4	CAN_AM22H	Mailbox 22 High Acceptance Mask
0XFFC0 2BB8	CAN_AM23L	Mailbox 23 Low Acceptance Mask
0XFFC0 2BBC	CAN_AM23H	Mailbox 23 High Acceptance Mask
0XFFC0 2BC0	CAN_AM24L	Mailbox 24 Low Acceptance Mask
0XFFC0 2BC4	CAN_AM24H	Mailbox 24 High Acceptance Mask

Table A-24. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2BC8	CAN_AM25L	Mailbox 25 Low Acceptance Mask
0XFFC0 2BCC	CAN_AM25H	Mailbox 25 High Acceptance Mask
0XFFC0 2BD0	CAN_AM26L	Mailbox 26 Low Acceptance Mask
0XFFC0 2BD4	CAN_AM26H	Mailbox 26 High Acceptance Mask
0XFFC0 2BD8	CAN_AM27L	Mailbox 27 Low Acceptance Mask
0XFFC0 2BDC	CAN_AM27H	Mailbox 27 High Acceptance Mask
0XFFC0 2BE0	CAN_AM28L	Mailbox 28 Low Acceptance Mask
0XFFC0 2BE4	CAN_AM28H	Mailbox 28 High Acceptance Mask
0XFFC0 2BE8	CAN_AM29L	Mailbox 29 Low Acceptance Mask
0XFFC0 2BEC	CAN_AM29H	Mailbox 29 High Acceptance Mask
0XFFC0 2BF0	CAN_AM30L	Mailbox 30 Low Acceptance Mask
0XFFC0 2BF4	CAN_AM30H	Mailbox 30 High Acceptance Mask
0XFFC0 2BF8	CAN_AM31L	Mailbox 31 Low Acceptance Mask
0XFFC0 2BFC	CAN_AM31H	Mailbox 31 High Acceptance Mask

Table A-25. CAN Mailbox Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2C00	CAN_MB00_DATA0	Mailbox 0 Data Word 0 [15:0] Register
0XFFC0 2C04	CAN_MB00_DATA1	Mailbox 0 Data Word 1 [31:16] Register
0XFFC0 2C08	CAN_MB00_DATA2	Mailbox 0 Data Word 2 [47:32] Register
0XFFC0 2C0C	CAN_MB00_DATA3	Mailbox 0 Data Word 3 [63:48] Register
0XFFC0 2C10	CAN_MB00_LENGTH	Mailbox 0 Data Length Code Register
0XFFC0 2C14	CAN_MB00_TIMESTAMP	Mailbox 0 Time Stamp Value Register
0XFFC0 2C18	CAN_MB00_ID0	Mailbox 0 Identifier Low Register
0XFFC0 2C1C	CAN_MB00_ID1	Mailbox 0 Identifier High Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2C20	CAN_MB01_DATA0	Mailbox 1 Data Word 0 [15:0] Register
0XFFC0 2C24	CAN_MB01_DATA1	Mailbox 1 Data Word 1 [31:16] Register
0XFFC0 2C28	CAN_MB01_DATA2	Mailbox 1 Data Word 2 [47:32] Register
0XFFC0 2C2C	CAN_MB01_DATA3	Mailbox 1 Data Word 3 [63:48] Register
0XFFC0 2C30	CAN_MB01_LENGTH	Mailbox 1 Data Length Code Register
0XFFC0 2C34	CAN_MB01_TIMESTAMP	Mailbox 1 Time Stamp Value Register
0XFFC0 2C38	CAN_MB01_ID0	Mailbox 1 Identifier Low Register
0XFFC0 2C3C	CAN_MB01_ID1	Mailbox 1 Identifier High Register
0XFFC0 2C40	CAN_MB02_DATA0	Mailbox 2 Data Word 0 [15:0] Register
0XFFC0 2C44	CAN_MB02_DATA1	Mailbox 2 Data Word 1 [31:16] Register
0XFFC0 2C48	CAN_MB02_DATA2	Mailbox 2 Data Word 2 [47:32] Register
0XFFC0 2C4C	CAN_MB02_DATA3	Mailbox 2 Data Word 3 [63:48] Register
0XFFC0 2C50	CAN_MB02_LENGTH	Mailbox 2 Data Length Code Register
0XFFC0 2C54	CAN_MB02_TIMESTAMP	Mailbox 2 Time Stamp Value Register
0XFFC0 2C58	CAN_MB02_ID0	Mailbox 2 Identifier Low Register
0XFFC0 2C5C	CAN_MB02_ID1	Mailbox 2 Identifier High Register
0XFFC0 2C60	CAN_MB03_DATA0	Mailbox 3 Data Word 0 [15:0] Register
0XFFC0 2C64	CAN_MB03_DATA1	Mailbox 3 Data Word 1 [31:16] Register
0XFFC0 2C68	CAN_MB03_DATA2	Mailbox 3 Data Word 2 [47:32] Register
0XFFC0 2C6C	CAN_MB03_DATA3	Mailbox 3 Data Word 3 [63:48] Register
0XFFC0 2C70	CAN_MB03_LENGTH	Mailbox 3 Data Length Code Register
0XFFC0 2C74	CAN_MB03_TIMESTAMP	Mailbox 3 Time Stamp Value Register
0XFFC0 2C78	CAN_MB03_ID0	Mailbox 3 Identifier Low Register
0XFFC0 2C7C	CAN_MB03_ID1	Mailbox 3 Identifier High Register
0XFFC0 2C80	CAN_MB04_DATA0	Mailbox 4 Data Word 0 [15:0] Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2C84	CAN_MB04_DATA1	Mailbox 4 Data Word 1 [31:16] Register
0XFFC0 2C88	CAN_MB04_DATA2	Mailbox 4 Data Word 2 [47:32] Register
0XFFC0 2C8C	CAN_MB04_DATA3	Mailbox 4 Data Word 3 [63:48] Register
0XFFC0 2C90	CAN_MB04_LENGTH	Mailbox 4 Data Length Code Register
0XFFC0 2C94	CAN_MB04_TIMESTAMP	Mailbox 4 Time Stamp Value Register
0XFFC0 2C98	CAN_MB04_ID0	Mailbox 4 Identifier Low Register
0XFFC0 2C9C	CAN_MB04_ID1	Mailbox 4 Identifier High Register
0XFFC0 2CA0	CAN_MB05_DATA0	Mailbox 5 Data Word 0 [15:0] Register
0XFFC0 2CA4	CAN_MB05_DATA1	Mailbox 5 Data Word 1 [31:16] Register
0XFFC0 2CA8	CAN_MB05_DATA2	Mailbox 5 Data Word 2 [47:32] Register
0XFFC0 2CAC	CAN_MB05_DATA3	Mailbox 5 Data Word 3 [63:48] Register
0XFFC0 2CB0	CAN_MB05_LENGTH	Mailbox 5 Data Length Code Register
0XFFC0 2CB4	CAN_MB05_TIMESTAMP	Mailbox 5 Time Stamp Value Register
0XFFC0 2CB8	CAN_MB05_ID0	Mailbox 5 Identifier Low Register
0XFFC0 2CBC	CAN_MB05_ID1	Mailbox 5 Identifier High Register
0XFFC0 2CC0	CAN_MB06_DATA0	Mailbox 6 Data Word 0 [15:0] Register
0XFFC0 2CC4	CAN_MB06_DATA1	Mailbox 6 Data Word 1 [31:16] Register
0XFFC0 2CC8	CAN_MB06_DATA2	Mailbox 6 Data Word 2 [47:32] Register
0XFFC0 2CCC	CAN_MB06_DATA3	Mailbox 6 Data Word 3 [63:48] Register
0XFFC0 2CD0	CAN_MB06_LENGTH	Mailbox 6 Data Length Code Register
0XFFC0 2CD4	CAN_MB06_TIMESTAMP	Mailbox 6 Time Stamp Value Register
0XFFC0 2CD8	CAN_MB06_ID0	Mailbox 6 Identifier Low Register
0XFFC0 2CDC	CAN_MB06_ID1	Mailbox 6 Identifier High Register
0XFFC0 2CE0	CAN_MB07_DATA0	Mailbox 7 Data Word 0 [15:0] Register
0XFFC0 2CE4	CAN_MB07_DATA1	Mailbox 7 Data Word 1 [31:16] Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2CE8	CAN_MB07_DATA2	Mailbox 7 Data Word 2 [47:32] Register
0XFFC0 2CEC	CAN_MB07_DATA3	Mailbox 7 Data Word 3 [63:48] Register
0XFFC0 2CF0	CAN_MB07_LENGTH	Mailbox 7 Data Length Code Register
0XFFC0 2CF4	CAN_MB07_TIMESTAMP	Mailbox 7 Time Stamp Value Register
0XFFC0 2CF8	CAN_MB07_ID0	Mailbox 7 Identifier Low Register
0XFFC0 2CFC	CAN_MB07_ID1	Mailbox 7 Identifier High Register
0XFFC0 2D00	CAN_MB08_DATA0	Mailbox 8 Data Word 0 [15:0] Register
0XFFC0 2D04	CAN_MB08_DATA1	Mailbox 8 Data Word 1 [31:16] Register
0XFFC0 2D08	CAN_MB08_DATA2	Mailbox 8 Data Word 2 [47:32] Register
0XFFC0 2D0C	CAN_MB08_DATA3	Mailbox 8 Data Word 3 [63:48] Register
0XFFC0 2D10	CAN_MB08_LENGTH	Mailbox 8 Data Length Code Register
0XFFC0 2D14	CAN_MB08_TIMESTAMP	Mailbox 8 Time Stamp Value Register
0XFFC0 2D18	CAN_MB08_ID0	Mailbox 8 Identifier Low Register
0XFFC0 2D1C	CAN_MB08_ID1	Mailbox 8 Identifier High Register
0XFFC0 2D20	CAN_MB09_DATA0	Mailbox 9 Data Word 0 [15:0] Register
0XFFC0 2D24	CAN_MB09_DATA1	Mailbox 9 Data Word 1 [31:16] Register
0XFFC0 2D28	CAN_MB09_DATA2	Mailbox 9 Data Word 2 [47:32] Register
0XFFC0 2D2C	CAN_MB09_DATA3	Mailbox 9 Data Word 3 [63:48] Register
0XFFC0 2D30	CAN_MB09_LENGTH	Mailbox 9 Data Length Code Register
0XFFC0 2D34	CAN_MB09_TIMESTAMP	Mailbox 9 Time Stamp Value Register
0XFFC0 2D38	CAN_MB09_ID0	Mailbox 9 Identifier Low Register
0XFFC0 2D3C	CAN_MB09_ID1	Mailbox 9 Identifier High Register
0XFFC0 2D40	CAN_MB10_DATA0	Mailbox 10 Data Word 0 [15:0] Register
0XFFC0 2D44	CAN_MB10_DATA1	Mailbox 10 Data Word 1 [31:16] Register
0XFFC0 2D48	CAN_MB10_DATA2	Mailbox 10 Data Word 2 [47:32] Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2D4C	CAN_MB10_DATA3	Mailbox 10 Data Word 3 [63:48] Register
0XFFC0 2D50	CAN_MB10_LENGTH	Mailbox 10 Data Length Code Register
0XFFC0 2D54	CAN_MB10_TIMESTAMP	Mailbox 10 Time Stamp Value Register
0XFFC0 2D58	CAN_MB10_ID0	Mailbox 10 Identifier Low Register
0XFFC0 2D5C	CAN_MB10_ID1	Mailbox 10 Identifier High Register
0XFFC0 2D60	CAN_MB11_DATA0	Mailbox 11 Data Word 0 [15:0] Register
0XFFC0 2D64	CAN_MB11_DATA1	Mailbox 11 Data Word 1 [31:16] Register
0XFFC0 2D68	CAN_MB11_DATA2	Mailbox 11 Data Word 2 [47:32] Register
0XFFC0 2D6C	CAN_MB11_DATA3	Mailbox 11 Data Word 3 [63:48] Register
0XFFC0 2D70	CAN_MB11_LENGTH	Mailbox 11 Data Length Code Register
0XFFC0 2D74	CAN_MB11_TIMESTAMP	Mailbox 11 Time Stamp Value Register
0XFFC0 2D78	CAN_MB11_ID0	Mailbox 11 Identifier Low Register
0XFFC0 2D7C	CAN_MB11_ID1	Mailbox 11 Identifier High Register
0XFFC0 2D80	CAN_MB12_DATA0	Mailbox 12 Data Word 0 [15:0] Register
0XFFC0 2D84	CAN_MB12_DATA1	Mailbox 12 Data Word 1 [31:16] Register
0XFFC0 2D88	CAN_MB12_DATA2	Mailbox 12 Data Word 2 [47:32] Register
0XFFC0 2D8C	CAN_MB12_DATA3	Mailbox 12 Data Word 3 [63:48] Register
0XFFC0 2D90	CAN_MB12_LENGTH	Mailbox 12 Data Length Code Register
0XFFC0 2D94	CAN_MB12_TIMESTAMP	Mailbox 12 Time Stamp Value Register
0XFFC0 2D98	CAN_MB12_ID0	Mailbox 12 Identifier Low Register
0XFFC0 2D9C	CAN_MB12_ID1	Mailbox 12 Identifier High Register
0XFFC0 2DA0	CAN_MB13_DATA0	Mailbox 13 Data Word 0 [15:0] Register
0XFFC0 2DA4	CAN_MB13_DATA1	Mailbox 13 Data Word 1 [31:16] Register
0XFFC0 2DA8	CAN_MB13_DATA2	Mailbox 13 Data Word 2 [47:32] Register
0XFFC0 2DAC	CAN_MB13_DATA3	Mailbox 13 Data Word 3 [63:48] Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2DB0	CAN_MB13_LENGTH	Mailbox 13 Data Length Code Register
0XFFC0 2DB4	CAN_MB13_TIMESTAMP	Mailbox 13 Time Stamp Value Register
0XFFC0 2DB8	CAN_MB13_ID0	Mailbox 13 Identifier Low Register
0XFFC0 2DBC	CAN_MB13_ID1	Mailbox 13 Identifier High Register
0XFFC0 2DC0	CAN_MB14_DATA0	Mailbox 14 Data Word 0 [15:0] Register
0XFFC0 2DC4	CAN_MB14_DATA1	Mailbox 14 Data Word 1 [31:16] Register
0XFFC0 2DC8	CAN_MB14_DATA2	Mailbox 14 Data Word 2 [47:32] Register
0XFFC0 2DCC	CAN_MB14_DATA3	Mailbox 14 Data Word 3 [63:48] Register
0XFFC0 2DD0	CAN_MB14_LENGTH	Mailbox 14 Data Length Code Register
0XFFC0 2DD4	CAN_MB14_TIMESTAMP	Mailbox 14 Time Stamp Value Register
0XFFC0 2DD8	CAN_MB14_ID0	Mailbox 14 Identifier Low Register
0XFFC0 2DDC	CAN_MB14_ID1	Mailbox 14 Identifier High Register
0XFFC0 2DE0	CAN_MB15_DATA0	Mailbox 15 Data Word 0 [15:0] Register
0XFFC0 2DE4	CAN_MB15_DATA1	Mailbox 15 Data Word 1 [31:16] Register
0XFFC0 2DE8	CAN_MB15_DATA2	Mailbox 15 Data Word 2 [47:32] Register
0XFFC0 2DEC	CAN_MB15_DATA3	Mailbox 15 Data Word 3 [63:48] Register
0XFFC0 2DF0	CAN_MB15_LENGTH	Mailbox 15 Data Length Code Register
0XFFC0 2DF4	CAN_MB15_TIMESTAMP	Mailbox 15 Time Stamp Value Register
0XFFC0 2DF8	CAN_MB15_ID0	Mailbox 15 Identifier Low Register
0XFFC0 2DFC	CAN_MB15_ID1	Mailbox 15 Identifier High Register
0XFFC0 2E00	CAN_MB16_DATA0	Mailbox 16 Data Word 0 [15:0] Register
0XFFC0 2E04	CAN_MB16_DATA1	Mailbox 16 Data Word 1 [31:16] Register
0XFFC0 2E08	CAN_MB16_DATA2	Mailbox 16 Data Word 2 [47:32] Register
0XFFC0 2E0C	CAN_MB16_DATA3	Mailbox 16 Data Word 3 [63:48] Register
0XFFC0 2E10	CAN_MB16_LENGTH	Mailbox 16 Data Length Code Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2E14	CAN_MB16_TIMESTAMP	Mailbox 16 Time Stamp Value Register
0XFFC0 2E18	CAN_MB16_ID0	Mailbox 16 Identifier Low Register
0XFFC0 2E1C	CAN_MB16_ID1	Mailbox 16 Identifier High Register
0XFFC0 2E20	CAN_MB17_DATA0	Mailbox 17 Data Word 0 [15:0] Register
0XFFC0 2E24	CAN_MB17_DATA1	Mailbox 17 Data Word 1 [31:16] Register
0XFFC0 2E28	CAN_MB17_DATA2	Mailbox 17 Data Word 2 [47:32] Register
0XFFC0 2E2C	CAN_MB17_DATA3	Mailbox 17 Data Word 3 [63:48] Register
0XFFC0 2E30	CAN_MB17_LENGTH	Mailbox 17 Data Length Code Register
0XFFC0 2E34	CAN_MB17_TIMESTAMP	Mailbox 17 Time Stamp Value Register
0XFFC0 2E38	CAN_MB17_ID0	Mailbox 17 Identifier Low Register
0XFFC0 2E3C	CAN_MB17_ID1	Mailbox 17 Identifier High Register
0XFFC0 2E40	CAN_MB18_DATA0	Mailbox 18 Data Word 0 [15:0] Register
0XFFC0 2E44	CAN_MB18_DATA1	Mailbox 18 Data Word 1 [31:16] Register
0XFFC0 2E48	CAN_MB18_DATA2	Mailbox 18 Data Word 2 [47:32] Register
0XFFC0 2E4C	CAN_MB18_DATA3	Mailbox 18 Data Word 3 [63:48] Register
0XFFC0 2E50	CAN_MB18_LENGTH	Mailbox 18 Data Length Code Register
0XFFC0 2E54	CAN_MB18_TIMESTAMP	Mailbox 18 Time Stamp Value Register
0XFFC0 2E58	CAN_MB18_ID0	Mailbox 18 Identifier Low Register
0XFFC0 2E5C	CAN_MB18_ID1	Mailbox 18 Identifier High Register
0XFFC0 2E60	CAN_MB19_DATA0	Mailbox 19 Data Word 0 [15:0] Register
0XFFC0 2E64	CAN_MB19_DATA1	Mailbox 19 Data Word 1 [31:16] Register
0XFFC0 2E68	CAN_MB19_DATA2	Mailbox 19 Data Word 2 [47:32] Register
0XFFC0 2E6C	CAN_MB19_DATA3	Mailbox 19 Data Word 3 [63:48] Register
0XFFC0 2E70	CAN_MB19_LENGTH	Mailbox 19 Data Length Code Register
0XFFC0 2E74	CAN_MB19_TIMESTAMP	Mailbox 19 Time Stamp Value Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2E78	CAN_MB19_ID0	Mailbox 19 Identifier Low Register
0XFFC0 2E7C	CAN_MB19_ID1	Mailbox 19 Identifier High Register
0XFFC0 2E80	CAN_MB20_DATA0	Mailbox 20 Data Word 0 [15:0] Register
0XFFC0 2E84	CAN_MB20_DATA1	Mailbox 20 Data Word 1 [31:16] Register
0XFFC0 2E88	CAN_MB20_DATA2	Mailbox 20 Data Word 2 [47:32] Register
0XFFC0 2E8C	CAN_MB20_DATA3	Mailbox 20 Data Word 3 [63:48] Register
0XFFC0 2E90	CAN_MB20_LENGTH	Mailbox 20 Data Length Code Register
0XFFC0 2E94	CAN_MB20_TIMESTAMP	Mailbox 20 Time Stamp Value Register
0XFFC0 2E98	CAN_MB20_ID0	Mailbox 20 Identifier Low Register
0XFFC0 2E9C	CAN_MB20_ID1	Mailbox 20 Identifier High Register
0XFFC0 2EA0	CAN_MB21_DATA0	Mailbox 21 Data Word 0 [15:0] Register
0XFFC0 2EA4	CAN_MB21_DATA1	Mailbox 21 Data Word 1 [31:16] Register
0XFFC0 2EA8	CAN_MB21_DATA2	Mailbox 21 Data Word 2 [47:32] Register
0XFFC0 2EAC	CAN_MB21_DATA3	Mailbox 21 Data Word 3 [63:48] Register
0XFFC0 2EB0	CAN_MB21_LENGTH	Mailbox 21 Data Length Code Register
0XFFC0 2EB4	CAN_MB21_TIMESTAMP	Mailbox 21 Time Stamp Value Register
0XFFC0 2EB8	CAN_MB21_ID0	Mailbox 21 Identifier Low Register
0XFFC0 2EBC	CAN_MB21_ID1	Mailbox 21 Identifier High Register
0XFFC0 2EC0	CAN_MB22_DATA0	Mailbox 22 Data Word 0 [15:0] Register
0XFFC0 2EC4	CAN_MB22_DATA1	Mailbox 22 Data Word 1 [31:16] Register
0XFFC0 2EC8	CAN_MB22_DATA2	Mailbox 22 Data Word 2 [47:32] Register
0XFFC0 2ECC	CAN_MB22_DATA3	Mailbox 22 Data Word 3 [63:48] Register
0XFFC0 2ED0	CAN_MB22_LENGTH	Mailbox 22 Data Length Code Register
0XFFC0 2ED4	CAN_MB22_TIMESTAMP	Mailbox 22 Time Stamp Value Register
0XFFC0 2ED8	CAN_MB22_ID0	Mailbox 22 Identifier Low Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2EDC	CAN_MB22_ID1	Mailbox 22 Identifier High Register
0XFFC0 2EE0	CAN_MB23_DATA0	Mailbox 23 Data Word 0 [15:0] Register
0XFFC0 2EE4	CAN_MB23_DATA1	Mailbox 23 Data Word 1 [31:16] Register
0XFFC0 2EE8	CAN_MB23_DATA2	Mailbox 23 Data Word 2 [47:32] Register
0XFFC0 2EEC	CAN_MB23_DATA3	Mailbox 23 Data Word 3 [63:48] Register
0XFFC0 2EF0	CAN_MB23_LENGTH	Mailbox 23 Data Length Code Register
0XFFC0 2EF4	CAN_MB23_TIMESTAMP	Mailbox 23 Time Stamp Value Register
0XFFC0 2EF8	CAN_MB23_ID0	Mailbox 23 Identifier Low Register
0XFFC0 2EFC	CAN_MB23_ID1	Mailbox 23 Identifier High Register
0XFFC0 2F00	CAN_MB24_DATA0	Mailbox 24 Data Word 0 [15:0] Register
0XFFC0 2F04	CAN_MB24_DATA1	Mailbox 24 Data Word 1 [31:16] Register
0XFFC0 2F08	CAN_MB24_DATA2	Mailbox 24 Data Word 2 [47:32] Register
0XFFC0 2F0C	CAN_MB24_DATA3	Mailbox 24 Data Word 3 [63:48] Register
0XFFC0 2F10	CAN_MB24_LENGTH	Mailbox 24 Data Length Code Register
0XFFC0 2F14	CAN_MB24_TIMESTAMP	Mailbox 24 Time Stamp Value Register
0XFFC0 2F18	CAN_MB24_ID0	Mailbox 24 Identifier Low Register
0XFFC0 2F1C	CAN_MB24_ID1	Mailbox 24 Identifier High Register
0XFFC0 2F20	CAN_MB25_DATA0	Mailbox 25 Data Word 0 [15:0] Register
0XFFC0 2F24	CAN_MB25_DATA1	Mailbox 25 Data Word 1 [31:16] Register
0XFFC0 2F28	CAN_MB25_DATA2	Mailbox 25 Data Word 2 [47:32] Register
0XFFC0 2F2C	CAN_MB25_DATA3	Mailbox 25 Data Word 3 [63:48] Register
0XFFC0 2F30	CAN_MB25_LENGTH	Mailbox 25 Data Length Code Register
0XFFC0 2F34	CAN_MB25_TIMESTAMP	Mailbox 25 Time Stamp Value Register
0XFFC0 2F38	CAN_MB25_ID0	Mailbox 25 Identifier Low Register
0XFFC0 2F3C	CAN_MB25_ID1	Mailbox 25 Identifier High Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2F40	CAN_MB26_DATA0	Mailbox 26 Data Word 0 [15:0] Register
0XFFC0 2F44	CAN_MB26_DATA1	Mailbox 26 Data Word 1 [31:16] Register
0XFFC0 2F48	CAN_MB26_DATA2	Mailbox 26 Data Word 2 [47:32] Register
0XFFC0 2F4C	CAN_MB26_DATA3	Mailbox 26 Data Word 3 [63:48] Register
0XFFC0 2F50	CAN_MB26_LENGTH	Mailbox 26 Data Length Code Register
0XFFC0 2F54	CAN_MB26_TIMESTAMP	Mailbox 26 Time Stamp Value Register
0XFFC0 2F58	CAN_MB26_ID0	Mailbox 26 Identifier Low Register
0XFFC0 2F5C	CAN_MB26_ID1	Mailbox 26 Identifier High Register
0XFFC0 2F60	CAN_MB27_DATA0	Mailbox 27 Data Word 0 [15:0] Register
0XFFC0 2F64	CAN_MB27_DATA1	Mailbox 27 Data Word 1 [31:16] Register
0XFFC0 2F68	CAN_MB27_DATA2	Mailbox 27 Data Word 2 [47:32] Register
0XFFC0 2F6C	CAN_MB27_DATA3	Mailbox 27 Data Word 3 [63:48] Register
0XFFC0 2F70	CAN_MB27_LENGTH	Mailbox 27 Data Length Code Register
0XFFC0 2F74	CAN_MB27_TIMESTAMP	Mailbox 27 Time Stamp Value Register
0XFFC0 2F78	CAN_MB27_ID0	Mailbox 27 Identifier Low Register
0XFFC0 2F7C	CAN_MB27_ID1	Mailbox 27 Identifier High Register
0XFFC0 2F80	CAN_MB28_DATA0	Mailbox 28 Data Word 0 [15:0] Register
0XFFC0 2F84	CAN_MB28_DATA1	Mailbox 28 Data Word 1 [31:16] Register
0XFFC0 2F88	CAN_MB28_DATA2	Mailbox 28 Data Word 2 [47:32] Register
0XFFC0 2F8C	CAN_MB28_DATA3	Mailbox 28 Data Word 3 [63:48] Register
0XFFC0 2F90	CAN_MB28_LENGTH	Mailbox 28 Data Length Code Register
0XFFC0 2F94	CAN_MB28_TIMESTAMP	Mailbox 28 Time Stamp Value Register
0XFFC0 2F98	CAN_MB28_ID0	Mailbox 28 Identifier Low Register
0XFFC0 2F9C	CAN_MB28_ID1	Mailbox 28 Identifier High Register
0XFFC0 2FA0	CAN_MB29_DATA0	Mailbox 29 Data Word 0 [15:0] Register

Table A-25. CAN Mailbox Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 2FA4	CAN_MB29_DATA1	Mailbox 29 Data Word 1 [31:16] Register
0XFFC0 2FA8	CAN_MB29_DATA2	Mailbox 29 Data Word 2 [47:32] Register
0XFFC0 2FAC	CAN_MB29_DATA3	Mailbox 29 Data Word 3 [63:48] Register
0XFFC0 2FB0	CAN_MB29_LENGTH	Mailbox 29 Data Length Code Register
0XFFC0 2FB4	CAN_MB29_TIMESTAMP	Mailbox 29 Time Stamp Value Register
0XFFC0 2FB8	CAN_MB29_ID0	Mailbox 29 Identifier Low Register
0XFFC0 2FBC	CAN_MB29_ID1	Mailbox 29 Identifier High Register
0XFFC0 2FC0	CAN_MB30_DATA0	Mailbox 30 Data Word 0 [15:0] Register
0XFFC0 2FC4	CAN_MB30_DATA1	Mailbox 30 Data Word 1 [31:16] Register
0XFFC0 2FC8	CAN_MB30_DATA2	Mailbox 30 Data Word 2 [47:32] Register
0XFFC0 2FCC	CAN_MB30_DATA3	Mailbox 30 Data Word 3 [63:48] Register
0XFFC0 2FD0	CAN_MB30_LENGTH	Mailbox 30 Data Length Code Register
0XFFC0 2FD4	CAN_MB30_TIMESTAMP	Mailbox 30 Time Stamp Value Register
0XFFC0 2FD8	CAN_MB30_ID0	Mailbox 30 Identifier Low Register
0XFFC0 2FDC	CAN_MB30_ID1	Mailbox 30 Identifier High Register
0XFFC0 2FE0	CAN_MB31_DATA0	Mailbox 31 Data Word 0 [15:0] Register
0XFFC0 2FE4	CAN_MB31_DATA1	Mailbox 31 Data Word 1 [31:16] Register
0XFFC0 2FE8	CAN_MB31_DATA2	Mailbox 31 Data Word 2 [47:32] Register
0XFFC0 2FEC	CAN_MB31_DATA3	Mailbox 31 Data Word 3 [63:48] Register
0XFFC0 2FF0	CAN_MB31_LENGTH	Mailbox 31 Data Length Code Register
0XFFC0 2FF4	CAN_MB31_TIMESTAMP	Mailbox 31 Time Stamp Value Register
0XFFC0 2FF8	CAN_MB31_ID0	Mailbox 31 Identifier Low Register
0XFFC0 2FFC	CAN_MB31_ID1	Mailbox 31 Identifier High Register

ACM Registers

ADC controller module (ACM) registers (0xFFC0 3100 – 0xFFC0 31FF) are listed in Table A-26.

Table A-26. AMC Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3100	ACM_CTL	ACM Control Register
0XFFC0 3104	ACM_TC0	ACM Timing Configuration 0 Register
0XFFC0 3108	ACM_TC1	ACM Timing Configuration 1 Register
0XFFC0 310C	ACM_STAT	ACM Status Register
0XFFC0 3110	ACM_ES	ACM Event Status Register
0XFFC0 3114	ACM_IMSK	ACM Interrupt Mask Register
0XFFC0 3118	ACM_MS	ACM Missed Event Status Register
0XFFC0 311C	ACM_EMSK	ACM Missed Event Interrupt Mask Register
0XFFC0 3120	ACM_ER0	ACM Event 0 Control Register
0XFFC0 3124	ACM_ER1	ACM Event 1 Control Register
0XFFC0 3128	ACM_ER2	ACM Event 2 Control Register
0XFFC0 312C	ACM_ER3	ACM Event 3 Control Register
0XFFC0 3130	ACM_ER4	ACM Event 4 Control Register
0XFFC0 3134	ACM_ER5	ACM Event 5 Control Register
0XFFC0 3138	ACM_ER6	ACM Event 6 Control Register
0XFFC0 313C	ACM_ER7	ACM Event 7 Control Register
0XFFC0 3140	ACM_ER8	ACM Event 8 Control Register
0XFFC0 3144	ACM_ER9	ACM Event 9 Control Register
0XFFC0 3148	ACM_ER10	ACM Event 10 Control Register
0XFFC0 314C	ACM_ER11	ACM Event 11 Control Register
0XFFC0 3150	ACM_ER12	ACM Event 12 Control Register

Table A-26. AMC Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3154	ACM_ER13	ACM Event 13 Control Register
0XFFC0 3158	ACM_ER14	ACM Event 14 Control Register
0XFFC0 315C	ACM_ER15	ACM Event 15 Control Register
0XFFC0 3180	ACM_ET0	ACM Event 0 Time Register
0XFFC0 3184	ACM_ET1	ACM Event 1 Time Register
0XFFC0 3188	ACM_ET2	ACM Event 2 Time Register
0XFFC0 318C	ACM_ET3	ACM Event 3 Time Register
0XFFC0 3190	ACM_ET4	ACM Event 4 Time Register
0XFFC0 3194	ACM_ET5	ACM Event 5 Time Register
0XFFC0 3198	ACM_ET6	ACM Event 6 Time Register
0XFFC0 319C	ACM_ET7	ACM Event 7 Time Register
0XFFC0 31A0	ACM_ET8	ACM Event 8 Time Register
0XFFC0 31A4	ACM_ET9	ACM Event 9 Time Register
0XFFC0 31A8	ACM_ET10	ACM Event 10 Time Register
0XFFC0 31AC	ACM_ET11	ACM Event 11 Time Register
0XFFC0 31B0	ACM_ET12	ACM Event 12 Time Register
0XFFC0 31B4	ACM_ET13	ACM Event 13 Time Register
0XFFC0 31B8	ACM_ET14	ACM Event 14 Time Register
0XFFC0 31BC	ACM_ET15	ACM Event 15 Time Register
0XFFC0 31C0	ACM_TMR0	ACM Timer 0 Registers
0XFFC0 31C4	ACM_TMR1	ACM Timer 1 Registers

PWM Registers

Pulsewidth modulator (PWM0 and PWM1) registers (0xFFC0 3700 – 0xFFC0 37FF) are listed in Table A-27 and Table A-28.

Table A-27. PWM0 Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3700	PWM0_CTRL	PWM0 Control Register
0XFFC0 3704	PWM0_STAT	PWM0 Status Register
0XFFC0 3708	PWM0_TM	PWM0 Period Register
0XFFC0 370C	PWM0_DT	PWM0 Dead Time Register
0XFFC0 3710	PWM0_GATE	PWM0 Chopping Control
0XFFC0 3714	PWM0_CHA	PWM0 Channel A Duty Control
0XFFC0 3718	PWM0_CHB	PWM0 Channel B Duty Control
0XFFC0 371C	PWM0_CHC	PWM0 Channel C Duty Control
0XFFC0 3720	PWM0_SEG	PWM0 Crossover and Output Enable
0XFFC0 3724	PWM0_SYNCWT	PWM0 Sync pulse width control
0XFFC0 3728	PWM0_CHAL	PWM0 Channel AL Duty Control (SR mode only)
0XFFC0 372C	PWM0_CHBL	PWM0 Channel BL Duty Control (SR mode only)
0XFFC0 3730	PWM0_CHCL	PWM0 Channel CL Duty Control (SR mode only)
0XFFC0 3734	PWM0_LSI	Low Side Invert (SR mode only)
0XFFC0 3738	PWM0_STAT2	PWM0 Status Register

Table A-28. PWM1 Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
Motor Control PWM1 Registers (0xFFC03000 - 0xFFC030FF)		
0XFFC0 3000	PWM1_CTRL	PWM1 Control Register
0XFFC0 3004	PWM1_STAT	PWM1 Status Register
0XFFC0 3008	PWM1_TM	PWM1 Period Register
0XFFC0 300C	PWM1_DT	PWM1 Dead Time Register
0XFFC0 3010	PWM1_GATE	PWM1 Chopping Control
0XFFC0 3014	PWM1_CHA	PWM1 Channel A Duty Control
0XFFC0 3018	PWM1_CHB	PWM1 Channel B Duty Control
0XFFC0 301C	PWM1_CHC	PWM1 Channel C Duty Control
0XFFC0 3020	PWM1_SEG	PWM1 Crossover and Output Enable
0XFFC0 3024	PWM1_SYNCWT	PWM1 Sync pulse width control
0XFFC0 3028	PWM1_CHAL	PWM1 Channel AL Duty Control (SR mode only)
0XFFC0 302C	PWM1_CHBL	PWM1 Channel BL Duty Control (SR mode only)
0XFFC0 3030	PWM1_CHCL	PWM1 Channel CL Duty Control (SR mode only)
0XFFC0 3034	PWM1_LSI	Low Side Invert (SR mode only)
0XFFC0 3038	PWM1_STAT2	PWM1 Status Register

RSI Registers

Removable storage interface (RSI) registers (0xFFC0 3800 – 0xFFC0 3CFF) are listed in Table A-29.

Table A-29. RSI Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3800	RSI_PWR_CONTROL	RSI Power Control Register
0XFFC0 3804	RSI_CLK_CONTROL	RSI Clock Control Register
0XFFC0 3808	RSI_ARGUMENT	RSI Argument Register
0XFFC0 380C	RSI_COMMAND	RSI Command Register
0XFFC0 3810	RSI_RESP_CMD	RSI Response Command Register
0XFFC0 3814	RSI_RESPONSE0	RSI Response Register
0XFFC0 3818	RSI_RESPONSE1	RSI Response Register
0XFFC0 381C	RSI_RESPONSE2	RSI Response Register
0XFFC0 3820	RSI_RESPONSE3	RSI Response Register
0XFFC0 3824	RSI_DATA_TIMER	RSI Data Timer Register
0XFFC0 3828	RSI_DATA_LGTH	RSI Data Length Register
0XFFC0 382C	RSI_DATA_CONTROL	RSI Data Control Register
0XFFC0 3830	RSI_DATA_CNT	RSI Data Counter Register
0XFFC0 3834	RSI_STATUS	RSI Status Register
0XFFC0 3838	RSI_STATUSCL	RSI Status Clear Register
0XFFC0 383C	RSI_MASK0	RSI Interrupt 0 Mask Register
0XFFC0 3840	RSI_MASK1	RSI Interrupt 1 Mask Register
0XFFC0 3848	RSI_FIFO_CNT	RSI FIFO Counter Register
0XFFC0 384C	RSI_CEATA_CONTROL	RSI CEATA Register
0XFFC0 3880	RSI_FIFO	RSI Data FIFO Register
0XFFC0 38C0	RSI_ESTAT	RSI Exception Status Register

Table A-29. RSI Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 38C4	RSI_EMASK	RSI Exception Mask Register
0XFFC0 38C8	RSI_CONFIG	RSI Configuration Register
0XFFC0 38CC	RSI_RD_WAIT_EN	RSI Read Wait Enable Register
0XFFC0 38D0	RSI_PID0	RSI Peripheral ID Register 0
0XFFC0 38D4	RSI_PID1	RSI Peripheral ID Register 1
0XFFC0 38D8	RSI_PID2	RSI Peripheral ID Register 2
0XFFC0 38DC	RSI_PID3	RSI Peripheral ID Register 3

ACM Registers

The ADC controller module (ACM) registers (0xFFC0 3100 – 0xFFC0 31FF) are listed in Table A-30.

Table A-30. ADC Controller Module (ACM) Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3100	ACM_CTL	ACM Control Register on page 22-29
0XFFC0 3104	ACM_TC0	ACM Timing Configuration 0 on page 22-36
0XFFC0 3108	ACM_TC1	ACM Timing Configuration1 on page 22-36
0XFFC0 310C	ACM_STAT	ACM Status Register on page 22-30
0XFFC0 3110	ACM_ES	ACM Event Status Register on page 22-31
0XFFC0 3114	ACM_IMSK	ACM Interrupt Mask Register on page 22-32
0XFFC0 3118	ACM_MS	ACM Missed Event Status on page 22-33
0XFFC0 311C	ACM_EMSK	ACM Event Missed Interrupt Mask on page 22-34
0XFFC0 3120	ACM_ER0	ACM Event0 Control Register on page 22-35

Table A-30. ADC Controller Module (ACM) Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 3124	ACM_ER1	ACM Event1 Control Register on page 22-35
0XFFC0 3128	ACM_ER2	ACM Event2 Control Register on page 22-35
0XFFC0 312C	ACM_ER3	ACM Event3 Control Register on page 22-35
0XFFC0 3130	ACM_ER4	ACM Event4 Control Register on page 22-35
0XFFC0 3134	ACM_ER5	ACM Event5 Control Register on page 22-35
0XFFC0 3138	ACM_ER6	ACM Event6 Control Register on page 22-35
0XFFC0 313C	ACM_ER7	ACM Event7 Control Register on page 22-35
0XFFC0 3140	ACM_ER8	ACM Event8 Control Register on page 22-35
0XFFC0 3144	ACM_ER9	ACM Event9 Control Register on page 22-35
0XFFC0 3148	ACM_ER10	ACM Event10 Control Register on page 22-35
0XFFC0 314C	ACM_ER11	ACM Event11 Control Register on page 22-35
0XFFC0 3150	ACM_ER12	ACM Event12 Control Register on page 22-35
0XFFC0 3154	ACM_ER13	ACM Event13 Control Register on page 22-35
0XFFC0 3158	ACM_ER14	ACM Event14 Control Register on page 22-35
0XFFC0 315C	ACM_ER15	ACM Event15 Control Register on page 22-35
0XFFC0 3180	ACM_ET0	ACM Event0 Time Register on page 22-36
0XFFC0 3184	ACM_ET1	ACM Event1 Time Register on page 22-36
0XFFC0 3188	ACM_ET2	ACM Event2 Time Register on page 22-36
0XFFC0 318C	ACM_ET3	ACM Event3 Time Register on page 22-36
0XFFC0 3190	ACM_ET4	ACM Event4 Time Register on page 22-36
0XFFC0 3194	ACM_ET5	ACM Event5 Time Register on page 22-36
0XFFC0 3198	ACM_ET6	ACM Event6 Time Register on page 22-36
0XFFC0 319C	ACM_ET7	ACM Event7 Time Register on page 22-36
0XFFC0 31A0	ACM_ET8	ACM Event8 Time Register on page 22-36
0XFFC0 31A4	ACM_ET9	ACM Event9 Time Register on page 22-36

Table A-30. ADC Controller Module (ACM) Registers (Cont'd)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0XFFC0 31A8	ACM_ET10	ACM Event10 Time Register on page 22-36
0XFFC0 31AC	ACM_ET11	ACM Event11 Time Register on page 22-36
0XFFC0 31B0	ACM_ET12	ACM Event12 Time Register on page 22-36
0XFFC0 31B4	ACM_ET13	ACM Event13 Time Register on page 22-36
0XFFC0 31B8	ACM_ET14	ACM Event14 Time Register on page 22-36
0XFFC0 31BC	ACM_ET15	ACM Event15 Time Register on page 22-36

B TEST FEATURES

This appendix discusses the test features of the ADSP-BF50x processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a boundary-scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table B-1)
- A TAP controller that controls all sequencing of events through the test registers
- An instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table B-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the

TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure B-1 shows the state diagram for the TAP controller.

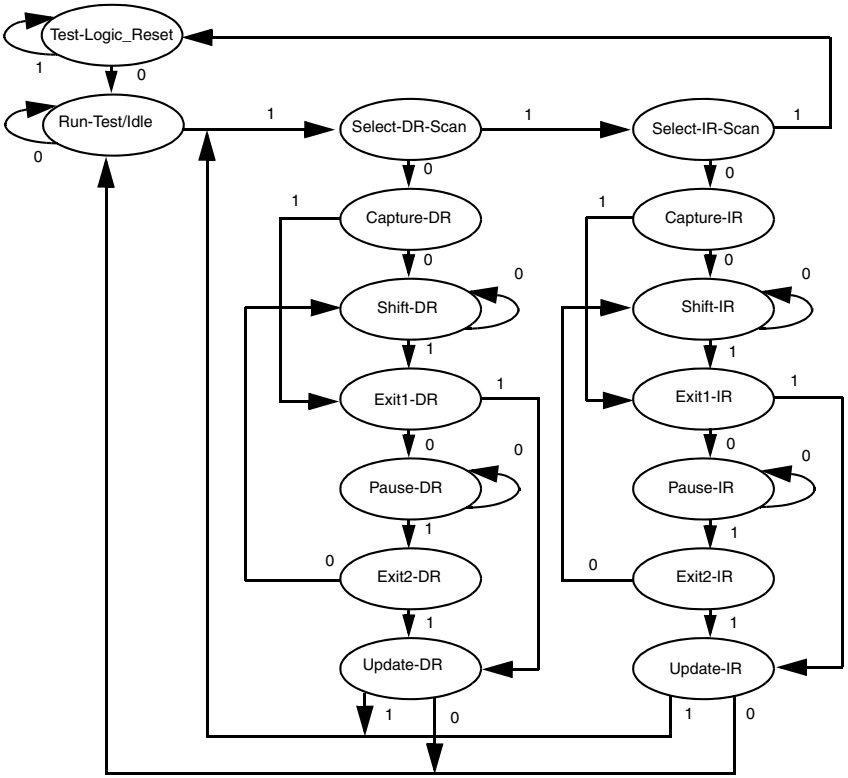


Figure B-1. TAP Controller State Diagram

Note:

- The TAP controller enters the test-logic-reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the test-logic-reset state when TRST is asynchronously asserted.
- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table B-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table B-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure B-2 shows the instruction bit scan ordering for the paths shown in Table B-2.

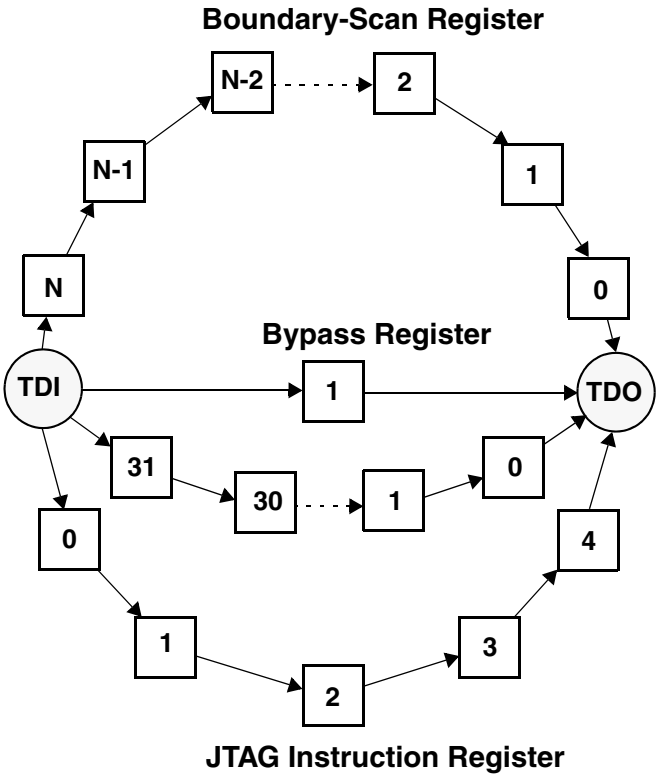


Figure B-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The EXTEST instruction selects the boundary-scan register to be connected between the TDI and TDO pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

Boundary-Scan Register

The boundary-scan register is selected by the EXTEST and SAMPLE/PRELOAD instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.

I INDEX

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z