TMS320C6000 DSP
Ethernet Media Access Controller (EMAC)/ Management Data Input/Output (MDIO) Module
Reference Guide

Literature Number: SPRU628A
March 2004

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

This document discusses the Ethernet Media Access Controller (EMAC) and Physical layer (PHY) device Management Data Input/Output (MDIO) module in the digital signal processors (DSPs) of the TMS320C6000™ DSP family.

The EMAC controls the flow of packet data from the DSP to the PHY. The MDIO module controls PHY configuration and status monitoring.

Although the entire feature set of the EMAC and MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC and MDIO features.

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.

Related Documentation From Texas Instruments

The following documents describe the C6000™ devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the TMS320C6000™ CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6000 DSP Peripherals Overview Reference Guide (literature number SPRU190) describes the peripherals available on the TMS320C6000™ DSPs.

TMS320C6000 Technical Brief (literature number SPRU197) gives an introduction to the TMS320C62x™ and TMS320C67x™ DSPs, development tools, and third-party support.

TMS320C64x Technical Overview (SPRU395) gives an introduction to the TMS320C64x™ DSP and discusses the application areas that are enhanced by the TMS320C64x VelociTI™.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000™ DSPs and includes application program examples.

TMS320C6000 Code Composer Studio Tutorial (literature number SPRU301) introduces the Code Composer Studio™ integrated development environment and software tools.

Code Composer Studio Application Programming Interface Reference Guide (literature number SPRU321) describes the Code Composer Studio™ application programming interface (API), which allows you to program custom plug-ins for Code Composer.

TMS320C6x Peripheral Support Library Programmer's Reference (literature number SPRU273) describes the contents of the TMS320C6000™ peripheral support library of functions and macros. It lists functions and macros both by header file and alphabetically, provides a complete description of each, and gives code examples to show how they are used.

TMS320C6000 Chip Support Library API Reference Guide (literature number SPRU401) describes a set of application programming interfaces (APIs) used to configure and control the on-chip peripherals.

Trademarks

Contents

Figures

Tables

Chapter 1

Overview

This chapter provides an overview of the Ethernet Media Access Controller (EMAC) and Physical layer (PHY) device Management Data Input/Output (MDIO) module in the digital signal processors (DSPs) of the TMS320C6000™ DSP family. Included are the features of the EMAC and MDIO modules, an overview of their operation, how these modules connect to the outside world, and definitions of terms used within this document. Although the entire feature set of the EMAC and MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC and MDIO features.

The EMAC controls the flow of packet data from the DSP to the PHY. The MDIO module controls PHY configuration and status monitoring.

Both the EMAC and the MDIO modules interface to the DSP through a custom interface that allows efficient data transmission and reception. This custom interface is referred to as the EMAC control module, and is considered integral to the EMAC/MDIO peripheral. The control module is also used to control device reset, interrupts, and system priority.

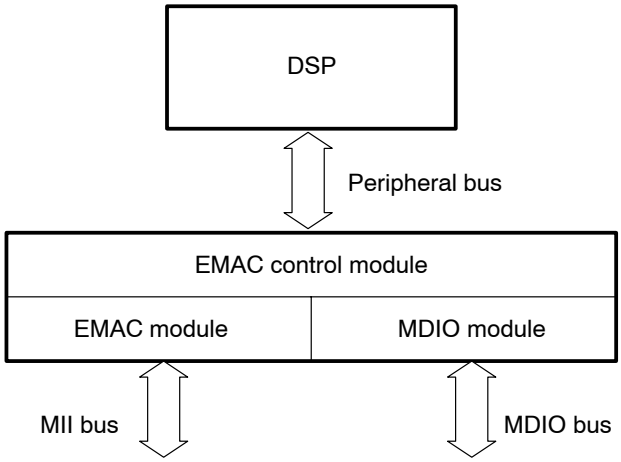
Topic **Page**

1.1 EMAC Control Module

The EMAC control module (Figure 1–1) is the main interface between the DSP core processor and the EMAC module and MDIO module. The EMAC control module contains the necessary components to allow the EMAC to make efficient use of DSP memory, plus it controls device reset, interrupts, and memory interface priority. The memory interface priority is used to balance the operation of the EMAC device with other memory transfer peripherals on the DSP. The EMAC control module includes the following features:

- Maps EMAC and MDIO registers into DSP configuration space.
- Controls EMAC and MDIO device reset and priority.
- Provides 4K byte local EMAC descriptor memory that allows the EMAC to operate on descriptors without affecting the DSP. (The descriptor memory holds enough information to transfer up to 256 Ethernet packets without DSP intervention.)
- Programmable interrupt logic allows the software driver to restrict the generation of back to back interrupts, allowing more work to be performed in a single call to the interrupt service routine.

Figure 1–1. EMAC Control Module Block Diagram



1.2 Ethernet Media Access Controller (EMAC) Module

Note:

The feature set of the EMAC module may vary between C6000 devices. Please see the device-specific datasheet for a listing of supported features.

The ethernet media access controller (EMAC) module provides an efficient interface between the DSP core processor and the networked community. The EMAC supports both 10Base-T (10Mbits/sec) and 100BaseTX (100Mbits/sec), in either half or full duplex, with hardware flow control and quality-of-service (QOS) support.

The basic feature set of the EMAC module is:

- EMAC acts as DMA master to either internal or external DSP memory space.
- Standard Media Independent Interface (MII) to physical layer device (PHY).
- Eight receive channels with VLAN tag discrimination for receive quality of service (QOS) support.
- Eight transmit channels with round-robin or fixed priority for transmit quality of service (QOS) support.
- Synchronous 10/100 Mbit operation.
- Ether-Stats and 802.3-Stats statistics gathering.
- Transmit CRC generation selectable on a per channel basis
- Broadcast frames selection for reception on a single channel.
- Multicast frames selection for reception on a single channel.
- Promiscuous receive mode frames selection for reception on a single channel (all frames, all good frames, short frames, error frames).
- Hardware flow control.

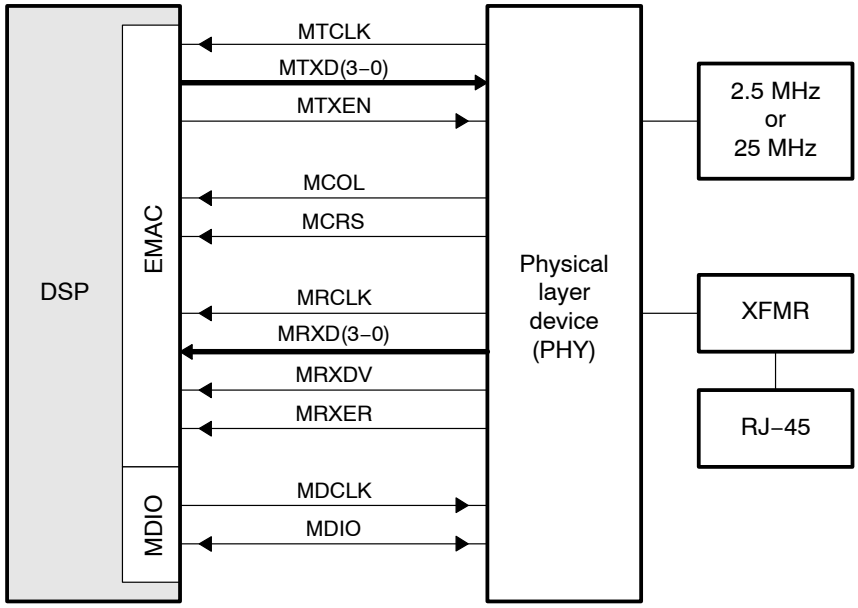
1.3 Management Data Input/Output (MDIO) Module

The management data input/output (MDIO) module implements the 802.3 serial management interface to interrogate and control Ethernet PHY(s) using a shared two-wire bus. Host software uses the MDIO module to configure the auto-negotiation parameters of each PHY attached to the EMAC, retrieve the negotiation results, and configure required parameters in the EMAC module for correct operation. The module is designed to allow almost transparent operation of the MDIO interface, with very little maintenance from the core processor.

1.4 System Level Connections

Figure 1–2 shows a DSP with integrated EMAC and MDIO interfaced in a typical system.

Figure 1–2. Typical Ethernet Configuration



The individual EMAC and MDIO signals are summarized in Table 1–1. For more information, refer to either the IEEE 802.3 standard or ISO/IEC 8802–3:2000(E).

The EMAC module does not include a transmit error (MTXER) pin. In the case of transmit error, CRC inversion is used to negate the validity of the transmitted frame.

Table 1−1. EMAC and MDIO Interface Signals

Signal Name	I/O	Description
MTCLK	I	Transmit clock (MTCLK). The transmit clock is a continuous clock that provides the timing reference for transmit operations. The MTXD and MTXEN signals are tied to this clock. The clock is generated by the PHY and is 2.5 MHz at 10Mb/s operation and 25 MHz at 100Mb/s operation.
MTXD(3−0)	O	Transmit data (MTXD). The transmit data pins are a collection of 4 data signals comprising 4 bits of data. MTDX0 is the least-significant bit (LSB). The signals are synchronized by MTCLK and valid only when MTXEN is asserted.
MTXEN	O	Transmit enable (MTXEN). The transmit enable signal indicates that the MTXD pins are generating nibble data for use by the PHY. It is driven synchronously to MTCLK.
MCOL	I	Collision detected (MCOL). The MCOL pin is asserted by the PHY when it detects a collision on the network. It remains asserted while the collision condition persists. This signal is not necessarily synchronous to MTCLK nor MRCLK. This pin is used in half-duplex operation only.
MCRS	I	Carrier sense (MCRS). The MCRS pin is asserted by the PHY when the network is not idle in either transmit or receive. The pin is deasserted when both transmit and receive are idle. This signal is not necessarily synchronous to MTCLK nor MRCLK. This pin is used in half-duplex operation only.
MRCLK	I	Receive clock (MRCLK). The receive clock is a continuous clock that provides the timing reference for receive operations. The MRXD, MRXDV, and MRXER signals are tied to this clock. The clock is generated by the PHY and is 2.5 MHz at 10Mb/s operation and 25 MHz at 100Mb/s operation.
MRXD(3−0)	I	Receive data (MRXD). The receive data pins are a collection of 4 data signals comprising 4 bits of data. MRDX0 is the least-significant bit (LSB). The signals are synchronized by MRCLK and valid only when MRXDV is asserted.
MRXDV	I	Receive data valid (MRXDV). The receive data valid signal indicates that the MRXD pins are generating nibble data for use by the EMAC. It is driven synchronously to MRCLK.
MRXER	I	Receive error (MRXER). The receive error signal is asserted for one or more MRCLK periods to indicate that an error was detected in the received frame. This is meaningful only during data transmission when MRXDV is active.
MDCLK	O	Management data clock (MDCLK). The MDIO data clock is sourced by the MDIO module on the DSP. It is used to synchronize MDIO data access operations done on the MDIO pin. The frequency of this clock is controlled by the CLKDIV bits in the MDIO control register (CONTROL).
MDIO	I/O	Management data input output (MDIO). The MDIO pin drives PHY management data into and out of the PHY by way of an access frame consisting of start of frame, read/write indication, PHY address, register address, and data bit cycles. The MDIO pin acts as an output for all but the data bit cycles at which time it is an input for read operations.

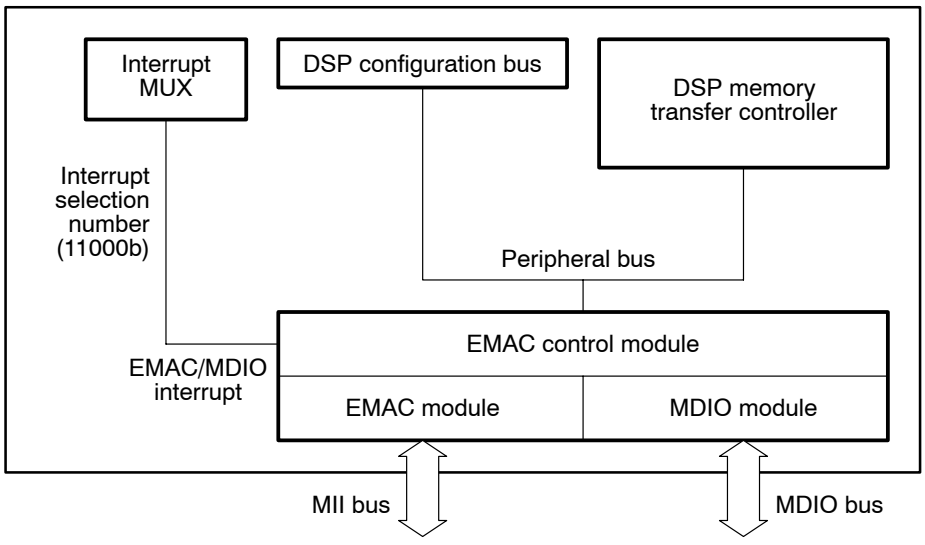
1.5 Architecture Overview

Figure 1–3 shows the three main functional modules of the EMAC/MDIO peripheral: EMAC control module, EMAC module, and MDIO module. The main interface between the EMAC control module and the DSP core is also shown. The following connections are made to the DSP:

- The peripheral bus connection from the EMAC control module allows the EMAC module to read and write both internal and external memory through the DSP's memory transfer controller (similar to an EDMA).
- The EMAC control module, EMAC, and MDIO all have control registers. These registers are memory mapped into DSP memory space via the DSP config bus. Along with these registers, the control module's internal RAM is mapped into this same range.
- The EMAC and MDIO interrupts are combined into a single interrupt within the control module. The interrupt from the control module then goes to the DSP's interrupt mux.

The EMAC and MDIO interrupts are combined within the control module, so only the control module interrupt needs to be monitored by the application software or device driver. The interrupt is mapped to a specific DSP interrupt through the use of the interrupt mux. The interrupt selection number of the combined EMAC/MDIO interrupt for use with the mux is 11000b.

Figure 1–3. EMAC and MDIO Block Diagram



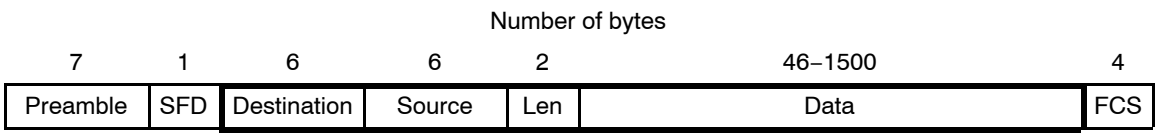
1.6 Definition of Terms

Table 1–2 lists the terms used throughout this document that relate to the operation of the EMAC or MDIO.

Table 1–2. Terms and Definitions

Term	Definition
Descriptor	A small memory structure that describes a larger block of memory in terms of size, location, and state. Descriptors are used by the EMAC and application to describe the memory buffers used to hold Ethernet data.
Ethernet packet (packet)	The collection of bytes that represents the data portion of a single Ethernet frame on the wire. The format of an Ethernet frame is shown in Figure 1–4. The Ethernet packet is shown outlined in bold. The frame check sequence covers the 60 to 1514 bytes shown in the bolded region (defined to be the packet data). Note that the 4-byte FCS field may or may not be included as part of the packet data, depending on how the EMAC is configured.
Ethernet MAC address (MAC address)	A unique 6-byte address that identifies an Ethernet device on the network. In an Ethernet packet, a MAC address is used twice, first to identify the packet's destination and second to identify the packet's sender or source. An Ethernet MAC address is normally specified in hexadecimal, using dashes to separate bytes, for example : 08h–00h–28h–32h–17h–42h. The first three bytes normally designate the manufacturer of the device. However, when the first byte of the address is odd (LSB is 1), the address is a group address (broadcast or multicast). The second bit specifies whether the address is globally or locally administrated (not considered in this document).
Broadcast MAC address	A special Ethernet MAC address used to send data to all Ethernet devices on the local network. The broadcast address is FFh–FFh–FFh–FFh–FFh–FFh. The LSB of the first byte is odd qualifying it as a group address; however, its value is reserved for broadcast. It is classified separately by the EMAC.
Multicast MAC address	A class of MAC address used to send a packet to potentially more than one recipient. A group address is specified by setting the LSB of the first MAC address byte to 1. Thus 01h–02h–03h–04h–05h–06h is a valid multicast address. Typically, an Ethernet MAC looks for only certain multicast addresses on a network in order to reduce traffic load. The multicast address list of acceptable packets is specified by the application.

Figure 1–4. Ethernet Frame



Legend: SFD = Start Frame Delimiter; FCS = Frame Check Sequence (CRC)

Chapter 2

EMAC Module

This chapter discusses the architecture and basic function of the EMAC module. Although the entire feature set of the EMAC module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC features.

Topic **Page**

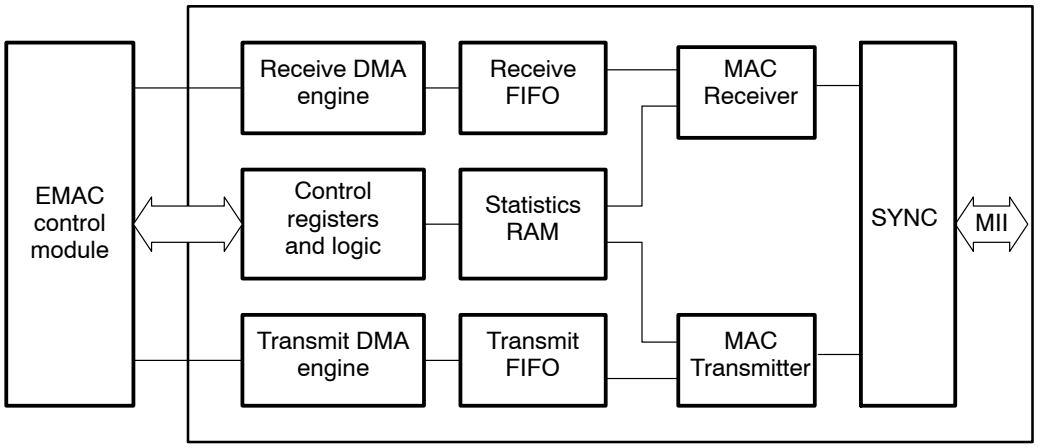
2.1 EMAC Module Components

The EMAC module (Figure 2–1) interfaces to the outside world through the Media Independent Interface (MII) interface and interfaces to the DSP core through the EMAC control module. The EMAC consists of the following logical components:

- Receive DMA engine
- Receive FIFO
- MAC receiver
- Transmit DMA engine
- Transmit FIFO
- MAC transmitter
- Statistics logic and RAM
- Control registers and logic

All logic is clocked synchronously with the CPUclk/4 peripheral clock except for the Ethernet MII synchronization logic.

Figure 2–1. EMAC Module Block Diagram



2.1.1 Receive DMA Engine

The receive DMA engine is the interface between the receive FIFO and the DSP core. It interfaces to the DSP through the bus arbiter in the EMAC control module.

2.1.2 Receive FIFO

The receive FIFO consists of three 64-byte FIFOs and associated control logic. The FIFO buffers received data in preparation for writing into packet buffers in DSP memory.

2.1.3 MAC Receiver

The MAC receiver detects and processes incoming network frames, deframes them, and puts them into the receive FIFO. The MAC receiver also detects errors and passes statistics to the statistics RAM.

2.1.4 Transmit DMA Engine

The transmit DMA engine is the interface between the transmit FIFO and the DSP core. It interfaces to the DSP through the bus arbiter in the EMAC control module.

2.1.5 Transmit FIFO

The transmit FIFO consists of three 64-byte FIFOs and associated control logic. The FIFO buffers data in preparation for transmission.

2.1.6 MAC Transmitter

The MAC transmitter formats frame data from the transmit FIFO and transmits the data using the CSMA/CD access protocol. Frame CRC can be automatically appended, if required. The MAC transmitter also detects transmission errors and passes statistics to the statistics RAM.

2.1.7 Statistics Logic and RAM

The Ethernet statistics are counted and stored in the statistics logic and FIFO RAM. This statistics RAM keeps track of 36 different Ethernet packet statistics.

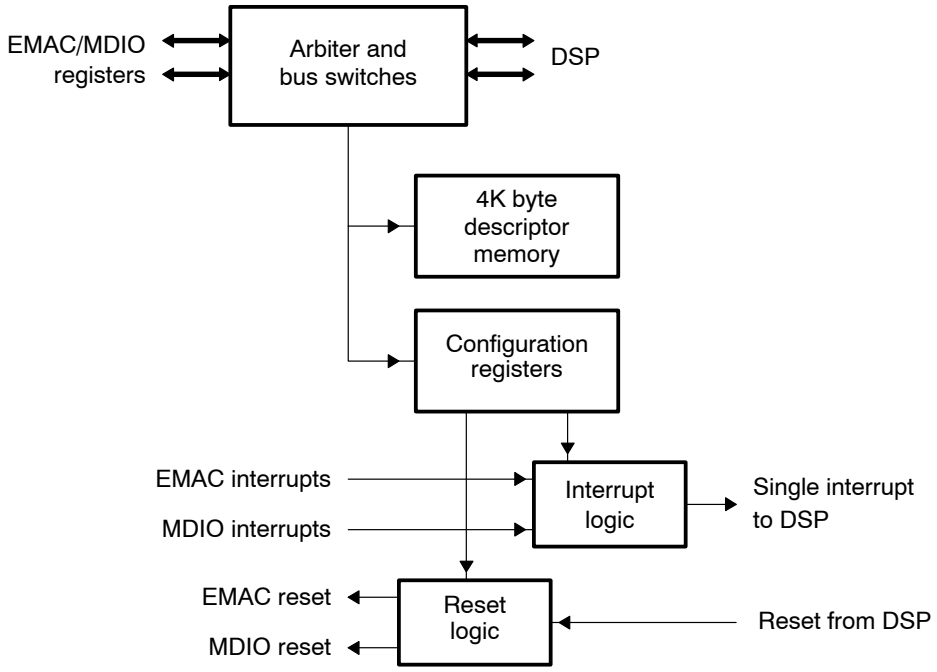
2.1.8 Control Registers and Logic

The EMAC is controlled by a set of memory-mapped registers. The control logic also signals transmit, receive, and status related interrupts to the DSP through the EMAC control module.

2.2 EMAC Control Module

The basic functions of the EMAC control module (Figure 2–2) are to interface the EMAC and MDIO modules to the DSP, and to provide for a local memory space to hold EMAC packet buffer descriptors. Local memory is used to help avoid contention to DSP memory spaces. Other functions include; the bus arbiter, transfer node priority control, reset control, and interrupt logic control.

Figure 2–2. EMAC Control Module Block Diagram



2.2.1 Internal Memory

The control module includes 4K bytes of internal memory. The internal memory block is essential for allowing the EMAC to operate more independently of the DSP. It also prevents memory underflow conditions when the EMAC issues read or write requests to descriptor memory. (Memory accesses to read or write the actual Ethernet packet data are protected by the EMACs internal FIFOs.)

A descriptor is a 16 byte memory structure which holds information about a single Ethernet packet buffer (that may contain a full or partial Ethernet packet). Thus with the 4K memory block provided for descriptor storage, the EMAC module can send and received up to a combined 256 packets before it needs to be serviced by application or driver software.

2.2.2 Bus Arbiter

The control module's bus arbiter operates transparently to the rest of the system. It is used for the following:

- ❏ Arbitrate between the DSP and EMAC buses for access to internal descriptor memory
- ❏ Arbitrate between internal EMAC buses for access to DSP system memory
- ❏ Map control module, EMAC module, and MDIO module registers into DSP memory space

2.2.3 Transfer Node Priority

The control module contains a register called EWTRCTRL that is used to set the priority of the transfer node used in issuing memory transfer requests to DSP system memory.

Although the EMAC has internal FIFOs to help alleviate memory transfer arbitration problems, the average transfer rate of data read and written by the EMAC to internal or external DSP memory must be at least that of the Ethernet wire rate. In addition, the internal FIFO system can not withstand a single memory latency event greater than the time it takes to fill or empty 2 internal 64 byte FIFOs.

For 100 Mb/s operation, these restrictions translate into the following rules:

- ❏ The short-term average, each 64 byte memory read/write request from the EMAC must be serviced in no more than 5.12 μs.
- ❏ Any single latency event in request servicing can be no longer than 10.24 μs.

The EMAC control module transfer control register (EWTRCTRL) is used to set the transfer node priority and the number of transfer requests that can be queued at any given time. The priority mechanism is identical to that used for cache and EDMA operations. It is important to have a balance between all peripherals. In most cases, the default priorities will not need adjustment. See section 5.1.1 for the description of EWTRCTRL.

2.2.4 Reset Control

The EMAC control module control register (EWCTL) can be used to individually reset either the EMAC or MDIO modules. Although a module reset is not part of normal system operation, there are some fatal error conditions (usually resulting from programming errors) that can only be corrected by resetting the module.

2.2.5 Interrupt Control

The EMAC control module combines multiple interrupt conditions generated by the EMAC and MDIO modules into a single interrupt signal that is mapped to a DSP interrupt via the DSP interrupt mux.

The control module uses two registers to control the interrupt signal to the DSP. First, the INTEN bit in EWCTL globally enables and disables the interrupt signal to the DSP. The INTEN bit is used to drive the interrupt line low during interrupt processing so that upon re-enabling the bit, the interrupt signal will rise if another interrupt condition exists thus creating a rising edge detectable by the DSP.

The EMAC control module interrupt timer count register (EWINTTCNT) is programmed with a value that counts down once EMAC/MDIO interrupts are enabled using EWCTL. The DSP interrupt signal is prevented from rising again until this count reaches zero.

The EWINTTCNT has no effect on interrupts once the count reaches zero, so there is no induced interrupt latency on random sporadic interrupts. However, the count will delay the issuance of a second interrupt immediately after a first. This protects the system from getting into an interrupt thrashing mode where the software interrupt service routine (ISR) completes processing just in time to get another interrupt. By postponing subsequent interrupts in a back-to-back condition, the software application or driver can get more work done in its ISR.

The EWINTTCNT reset value can be adjusted from within the ISR according to current system load, or simply set to a fixed value that assures a maximum number of interrupts per second.

The counter counts at a frequency of CPUclock/4; its default reset count is 0 (inactive), its maximum value is 1 FFFFh (131 071).

2.3 EMAC Module Operational Overview

The EMAC module operates independently of the DSP. It is configured and controlled by its register set mapped into DSP memory. Information about data packets is communicated by use of 16-byte descriptors that are placed in a 4K-byte block of RAM in the EMAC control module.

For transmit operations, each 16-byte descriptor describes a packet or packet fragment in DSP internal or external memory. For receive operations, each 16-byte descriptor represents a free packet buffer or buffer fragment. On both transmit and receive, an Ethernet packet is allowed to span one or more memory fragments, represented by one 16-byte descriptor per fragment. In typical operation, there is only one descriptor per receive buffer, but transmit packets may be fragmented, depending on the software architecture.

An interrupt is issued to the DSP whenever a transmit or receive operation has completed. However, it is not necessary for the DSP to service the interrupt while there are additional resources available. In other words, the EMAC continues to receive Ethernet packets until its receive descriptor list has been exhausted. On transmit operations, the transmit descriptors need only be serviced to recover their associated memory buffer. Thus, it is possible to delay servicing of the EMAC interrupt if there are real-time tasks to perform.

Eight channels are supplied for both transmit and receive operations. On transmit, the eight channels represent eight independent transmit queues. The EMAC can be configured to treat these channels as an equal priority “round-robin” queue, or as a set of eight fixed-priority queues. On receive, the eight channels represent eight independent receive queues with packet classification. Packets are classified based on the destination MAC address. Each of the eight channels is assigned its own MAC address. Also, specific types of frames can be sent to specific channels. For example; multicast, broadcast, or other (promiscuous, error, etc.), can each be received on a specific receive channel queue.

The EMAC keeps track of 36 different statistics, plus keeps the status of each individual packet in its corresponding packet descriptor.

2.3.1 Packet Buffer Descriptors

The buffer descriptor is a central part of the EMAC module and is how the application software describes Ethernet packets to be sent and empty buffers to be filled with incoming packet data.

The basic descriptor format is shown in Figure 2–3.

Figure 2–3. Basic Descriptor Format

Word Offset	Bit Fields	
	31 **16**	**15** **0**
0	Next Descriptor Pointer	
1	Buffer Pointer	
2	Buffer Offset	Buffer Length
3	Flags	Packet Length

Next Descriptor Pointer

The next descriptor pointer is used to create a single-linked list of descriptors. Each descriptor describes to a packet or a packet fragment. When a descriptor points to a single buffer packet or the first fragment of a packet, the start of packet (SOP) flag is set in the flags field. When a descriptor points to a single buffer packet or the last fragment of a packet, the end of packet (EOP) flag is set. When a packet is fragmented, each fragment must have its own descriptor and appear sequentially in the descriptor linked list.

Buffer Pointer

The buffer pointer points to the actual memory buffer that contains packet data during transmit operations, or is an empty buffer ready to receive packet data during receive operations.

Buffer Offset

The buffer offset is the offset from the start of the packet buffer to the first byte of valid data. This field only has meaning when the buffer descriptor points to a buffer that actually contains data.

Buffer Length

The buffer length is the actual number of valid packet data bytes stored in the buffer. If the buffer is empty and waiting to receive data, this field represents the size of the empty buffer.

Flags

The flags field contains more information about the buffer, such as, is it the first fragment in a packet (SOP), the last fragment in a packet (EOP), or contains an entire contiguous Ethernet packet (both SOP and EOP). The flags are described in sections 2.3.4 and 2.3.5.

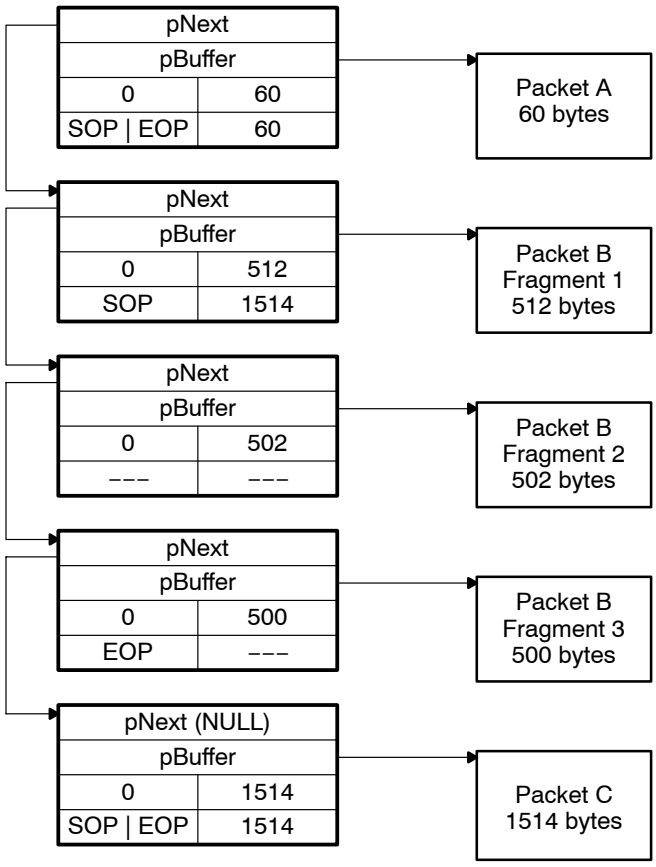
Packet Length

The packet length only has meaning for buffers that both contain data and are the start of a new packet (SOP). In the case of SOP descriptors, the packet length field contains the length of the entire Ethernet packet, regardless if it is contained in a single buffer or fragmented over several buffers.

Example

For example, consider three packets to be transmitted, Packet A is a single fragment (60 bytes), Packet B is fragmented over three buffers (1514 bytes), and Packet C is a single fragment (1514 bytes). The linked list of descriptors to describe these three packets is shown in Figure 2−4.

Figure 2–4. Typical Descriptor Linked List



2.3.2 Transmit and Receive Descriptor Queues

The EMAC module processes descriptors in linked list chains as discussed in section 2.3.1. The lists controlled by the EMAC are maintained by the application software though the use of the head descriptor pointer (HDP) registers. Since the EMAC supports eight channels for both transmit and receive, there are eight head descriptor pointer registers for both. They are designated as:

- TX*n*HDP – Transmit Channel *n* DMA Head Descriptor Pointer Register
- RX*n*HDP – Receive Channel *n* DMA Head Descriptor Pointer Register

After EMAC reset, and before enabling the EMAC for send or receive, all 16 head descriptor pointer registers must be initialized to zero.

There is a simple system that the EMAC uses to determine if a descriptor is currently owned by the EMAC or by the application software. There is a flag in the descriptor Flags field called OWNER. When this flag is set, the packet that is referenced by the descriptor is considered to be owned by the EMAC. Note that ownership is done on a packet based granularity, not on descriptor granularity, so only SOP descriptors make use of the OWNER flag. As packets are processed, the EMAC will patch the SOP descriptor of the corresponding packet and clear the OWNER flag. This is an indication that the EMAC has finished processing all descriptors up to and including the first with the EOP flag set indicating the end of the packet (note this may only be one descriptor with both the SOP and EOP flags set).

To first add a descriptor or a linked list of descriptors to an EMAC descriptor queue, the software application simply writes the pointer to the descriptor or first descriptor of a list to the corresponding HDP register. Note that the last descriptor in the list must have its “next” pointer cleared to zero. This is the only way the EMAC has of detecting the end of the list. So in the case where only a single descriptor is added, its “next descriptor” pointer must be initialized to zero.

The HDP register must never be written to a second time while a previous list is active. To add additional descriptors to a descriptor list already owned by the EMAC, the NULL “next” pointer of the last descriptor of the previous list is patched with a pointer to the first descriptor in the new list. The list of new descriptors to be appended to the existing list must itself be NULL terminated before the pointer patch is performed.

There is a potential race condition where the EMAC may read the “next” pointer of a descriptor as NULL in the instant before an application appends additional descriptors to the list by patching the pointer. This case is handled by the software application always examining the Flags field of all EOP packets, looking for a special flag called end of queue (EOQ). The EOQ flag is set by the EMAC on the last descriptor of a packet when the descriptor’s “next” pointer is NULL. This is the way the EMAC indicates to the software application that it believes it has reached the end of the list. When the software application sees the EOQ flag set, and there are more descriptors to process, the application may at that time submit the new list, or the portion of the appended list that was missed, by writing the new list pointer to the same HDP register that started the process.

This process applies when adding packets to a transmit list, and empty buffers to a receive list.

2.3.3 Transmit and Receive EMAC Interrupts

The EMAC processes descriptors in linked list chains as discussed in section 2.3.1, using the linked list queue mechanism discussed in section 2.3.2.

The EMAC synchronizes descriptor list processing though the use of interrupts to the software application. The interrupts are controlled by the application by using the interrupt masks, global interrupt enable, and the interrupt acknowledge register (INTACK).

Since the EMAC supports eight channels for both transmit and receive, there are eight INTACK registers for both. They are designated as:

- TX*n*INTACK – Transmit Channel *n* Interrupt Acknowledge Register
- RX*n*INTACK – Receive Channel *n* Interrupt Acknowledge Register

These registers serve two purposes. When read, they return the pointer to the last descriptor that the EMAC has processed. When written by the software application, the value represents the last descriptor processed by the software application. When these two values do not match, the interrupt is active.

Whether or not an active interrupt actually interrupts the DSP is determined by the system configuration. This is covered more in Chapter 4, *Software Operation*, but in general, the global interrupt for all EMAC and MDIO must be enabled in the EMAC control module, and it also must be mapped in the DSP interrupt MUX and enabled as a DSP interrupt. Given the system is configured properly, the interrupt for a specific receive or transmit channel executes under the previously described conditions when the corresponding interrupt is enabled in the EMAC using the RXINTMASKSET or TXINTMASKSET registers.

Whether or not the interrupt is enabled, the current state of the receive or transmit channel interrupt can be examined directly by the software application by reading the RXINTSTATRAW and TXINTSTATRAW registers.

Interrupts are acknowledged when the application software updates the value of TX*n*INTACK or RX*n*INTACK with a value that matches the internal value kept by the EMAC.

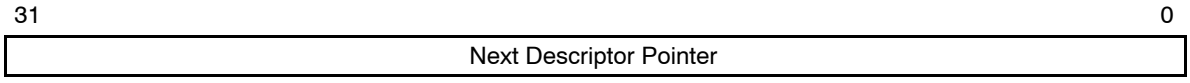
This mechanism ensures that the application software never misses an EMAC interrupt, since the interrupt and its acknowledgment are tied directly to the actual buffer descriptors processing done by each.

2.3.4 Transmit Buffer Descriptor Format

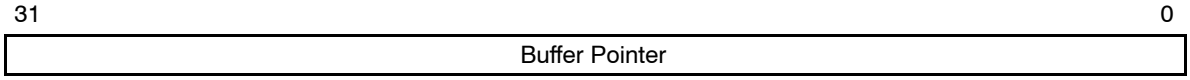
A transmit (TX) buffer descriptor (Figure 2−5) is a contiguous block of four 32-bit data words aligned on a 32-bit boundary that describes a packet or a packet fragment. Figure 2−6 shows the transmit descriptor described by a C structure.

Figure 2−5. Transmit Descriptor Format

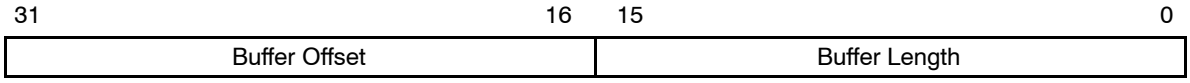
(a) Word 0



(b) Word 1



(c) Word 2



(d) Word 3

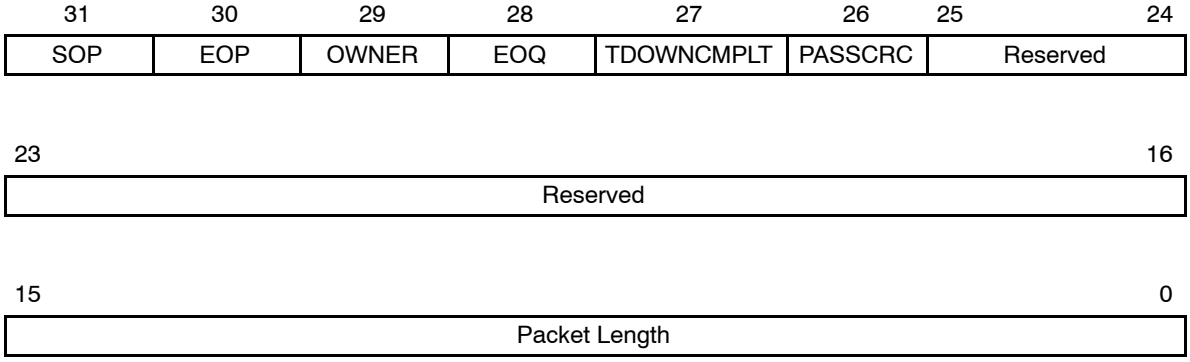


Figure 2−6. Transmit Descriptor in C Structure Format

```
/*
// EMAC Descriptor
//
// The following is the format of a single buffer descriptor
// on the EMAC.
*/
typedef struct _EMAC_Desc {
  struct _EMAC_Desc *pNext;     /* Pointer to next descriptor in chain */
  Uint8             *pBuffer;    /* Pointer to data buffer             */
  Uint32           BufOffLen;  /* Buffer Offset(MSW) and Length(LSW)   */
  Uint32           PktFlgLen;  /* Packet Flags(MSW) and Length(LSW)    */
} EMAC_Desc;

/* Packet Flags (some used for RX only) */
#define EMAC_DSC_FLAG_SOP               0x80000000u
#define EMAC_DSC_FLAG_EOP               0x40000000u
#define EMAC_DSC_FLAG_OWNER             0x20000000u
#define EMAC_DSC_FLAG_EOQ               0x10000000u
#define EMAC_DSC_FLAG_TDOWNCMPLT        0x08000000u
#define EMAC_DSC_FLAG_PASSCRC           0x04000000u
#define EMAC_DSC_FLAG_JABBER            0x02000000u
#define EMAC_DSC_FLAG_OVERSIZE          0x01000000u
#define EMAC_DSC_FLAG_FRAGMENT          0x00800000u
#define EMAC_DSC_FLAG_UNDERSIZED        0x00400000u
#define EMAC_DSC_FLAG_CONTROL           0x00200000u
#define EMAC_DSC_FLAG_OVERRUN           0x00100000u
#define EMAC_DSC_FLAG_CODEERROR         0x00080000u
#define EMAC_DSC_FLAG_ALIGNERROR        0x00040000u
#define EMAC_DSC_FLAG_CRCERROR          0x00020000u
#define EMAC_DSC_FLAG_NOMATCH           0x00010000u
```

Next Descriptor Pointer

This pointer points to the 32-bit word aligned memory address of the next buffer descriptor in the transmit queue. This pointer is used to create a linked list of buffer descriptors. If the value of this pointer is zero, then the current buffer is the last buffer in the queue. The software application must set this value prior to adding the descriptor to the active transmit list. This pointer is not altered by the EMAC.

The value of pNext should never be altered once the descriptor in an active transmit queue, unless its current value is NULL. If the pNext pointer is in initially NULL, and more packets need to be queued for transmit, the software application may alter this pointer to point to a newly appended descriptor. The EMAC will use the new pointer value and proceed to the next descriptor unless the pNext value has already been read. In this latter case, the transmitter will halt on the transmit channel in question, and the software application may restart it at that time. The software can detect this case by checking for an end of queue (EOQ) condition flag on the updated packet descriptor when it is returned by the EMAC.

Buffer Pointer

The buffer pointer is the byte-aligned memory address of the memory buffer associated with the buffer descriptor. The software application must set this value prior to adding the descriptor to the active transmit list. This pointer is not altered by the EMAC.

Buffer Offset

This 16-bit field indicates how many unused bytes are at the start of the buffer. For example, a value of 0000h indicates that no unused bytes are at the start of the buffer and that valid data begins on the first byte of the buffer, while a value of 000Fh indicates that the first 15 bytes of the buffer are to be ignored by the EMAC and that valid buffer data starts on byte 16 of the buffer. The software application must set this value prior to adding the descriptor to the active transmit list. This field is not altered by the EMAC.

Note that this value is only checked on the first descriptor of a given packet (where the start of packet (SOP) flag is set). It can not be used to specify the offset of subsequent packet fragments. Also, since the buffer pointer may point to any byte-aligned address, this field may be entirely superfluous, depending on the device driver architecture.

The range of legal values for this field is 0 to (Buffer Length – 1).

Buffer Length

This 16-bit field indicates how many valid data bytes are in the buffer. On single fragment packets, this value is also the total length of the packet data to be transmitted. If the buffer offset field is used, the offset bytes are not counted as part of this length. This length counts only valid data bytes. The software application must set this value prior to adding the descriptor to the active transmit list. This field is not altered by the EMAC.

Packet Length

This 16-bit field specifies the number of data bytes in the entire packet. Any leading buffer offset bytes are not included. The sum of the buffer length fields of each of the packet's fragments (if more than one) must be equal to the packet length. The software application must set this value prior to adding the descriptor to the active transmit list. This field is not altered by the EMAC. This value is only checked on the first descriptor of a given packet (where the start of packet (SOP) flag is set).

Start of Packet (SOP) Flag

When set, this flag indicates that the descriptor points to a packet buffer that is the start of a new packet. In the case of a single fragment packet, both the SOP and end of packet (EOP) flags are set. Otherwise; the descriptor pointing to the last packet buffer for the packet sets the EOP flag. This bit is set by the software application and is not altered by the EMAC.

End of Packet (EOP) Flag

When set, this flag indicates that the descriptor points to a packet buffer that is last for a given packet. In the case of a single fragment packet, both the start of packet (SOP) and EOP flags are set. Otherwise; the descriptor pointing to the last packet buffer for the packet sets the EOP flag. This bit is set by the software application and is not altered by the EMAC.

Ownership (OWNER) Flag

When set, this flag indicates that all the descriptors for the given packet (from SOP to EOP) are currently owned by the EMAC. This flag is set by the software application on the SOP packet descriptor before adding the descriptor to the transmit descriptor queue. For a single fragment packet, the SOP, EOP, and OWNER flags are all set. The OWNER flag is cleared by the EMAC once it is finished with all the descriptors for the given packet. Note that this flag is valid on SOP descriptors only.

End of Queue (EOQ) Flag

When set, this flag indicates that the descriptor in question was the last descriptor in the transmit queue for a given transmit channel, and that the transmitter has halted. This flag is initially cleared by the software application prior to adding the descriptor to the transmit queue. This bit is set by the EMAC when the EMAC identifies that a descriptor is the last for a given packet (the EOP flag is set), and there are no more descriptors in the transmit list (next descriptor pointer is NULL).

The software application can use this bit to detect when the EMAC transmitter for the corresponding channel has halted. This is useful when the application appends additional packet descriptors to a transmit queue list that is already owned by the EMAC. Note that this flag is valid on EOP descriptors only.

Teardown Complete (TDOWNCMPLT) Flag

This flag is used when a transmit queue is being torn down, or aborted, instead of allowing it to be transmitted. This would happen under device driver reset or shutdown conditions. The EMAC sets this bit in the SOP descriptor of each packet as it is aborted from transmission.

Note that this flag is valid on SOP descriptors only. Also note that only the first packet in an unsent list has the TDOWNCMPLT flag set. Subsequent descriptors are not even processed by the EMAC.

Pass CRC (PASSCRC) Flag

This flag is set by the software application in the SOP packet descriptor before it adds the descriptor to the transmit queue. Setting this bit indicates to the EMAC that the 4 byte Ethernet CRC is already present in the packet data, and that the EMAC should not generate its own version of the CRC.

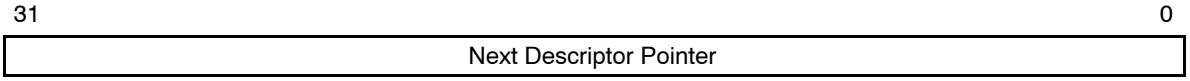
When the CRC flag is cleared, the EMAC generates and appends the 4-byte CRC. The buffer length and packet length fields do not include the CRC bytes. When the CRC flag is set, the 4-byte CRC is supplied by the software application and is already appended to the end of the packet data. The buffer length and packet length fields include the CRC bytes, as they are part of the valid packet data. Note that this flag is valid on SOP descriptors only.

2.3.5 Receive Buffer Descriptor Format

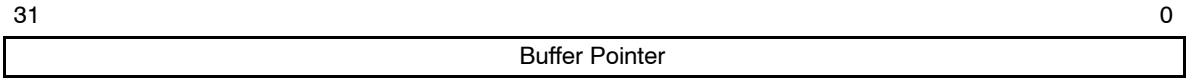
A receive (RX) buffer descriptor (Figure 2−7) is a contiguous block of four 32-bit data words aligned on a 32-bit boundary. Figure 2−8 shows the receive descriptor described by a C structure.

Figure 2−7. Receive Descriptor Format

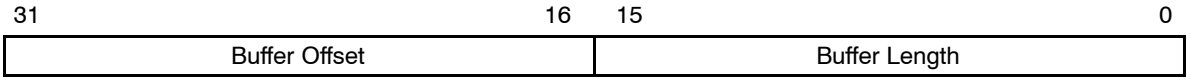
(a) Word 0



(b) Word 1



(c) Word 2



(d) Word 3

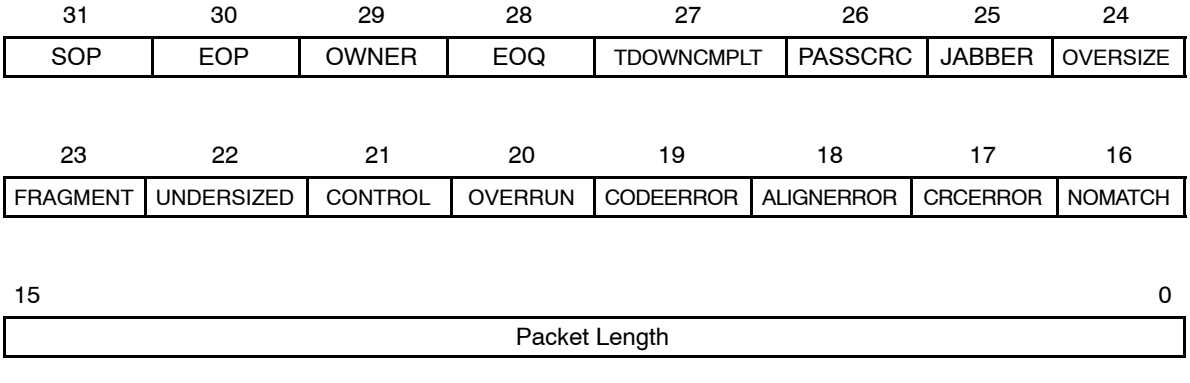


Figure 2−8. Receive Descriptor in C Structure Format

```
/*
// EMAC Descriptor
//
// The following is the format of a single buffer descriptor
// on the EMAC.
*/
typedef struct _EMAC_Desc {
  struct _EMAC_Desc *pNext;     /* Pointer to next descriptor in chain */
  Uint8             *pBuffer;    /* Pointer to data buffer             */
  Uint32           BufOffLen;  /* Buffer Offset(MSW) and Length(LSW)   */
  Uint32           PktFlgLen;  /* Packet Flags(MSW) and Length(LSW)    */
} EMAC_Desc;

/* Packet Flags (some used for RX only) */
#define EMAC_DSC_FLAG_SOP               0x80000000u
#define EMAC_DSC_FLAG_EOP               0x40000000u
#define EMAC_DSC_FLAG_OWNER             0x20000000u
#define EMAC_DSC_FLAG_EOQ               0x10000000u
#define EMAC_DSC_FLAG_TDOWNCMPLT        0x08000000u
#define EMAC_DSC_FLAG_PASSCRC           0x04000000u
#define EMAC_DSC_FLAG_JABBER            0x02000000u
#define EMAC_DSC_FLAG_OVERSIZE          0x01000000u
#define EMAC_DSC_FLAG_FRAGMENT          0x00800000u
#define EMAC_DSC_FLAG_UNDERSIZED        0x00400000u
#define EMAC_DSC_FLAG_CONTROL           0x00200000u
#define EMAC_DSC_FLAG_OVERRUN           0x00100000u
#define EMAC_DSC_FLAG_CODEERROR         0x00080000u
#define EMAC_DSC_FLAG_ALIGNERROR        0x00040000u
#define EMAC_DSC_FLAG_CRCERROR          0x00020000u
#define EMAC_DSC_FLAG_NOMATCH           0x00010000u
```

Next Descriptor Pointer

This pointer points to the 32-bit word aligned memory address of the next buffer descriptor in the receive queue. This pointer is used to create a linked list of buffer descriptors. If the value of this pointer is zero, then the current buffer is the last buffer in the queue. The software application must set this value prior to adding the descriptor to the active receive list. This pointer is not altered by the EMAC.

The value of pNext should never be altered once the descriptor in an active receive queue, unless its current value is NULL. If the pNext pointer is in initially NULL, and more empty buffers can be added to the pool, the software application may alter this pointer to point to a newly appended descriptor. The EMAC will use the new pointer value and proceed to the next descriptor unless the pNext value has already been read. In this latter case, the receiver will halt the receive channel in question, and the software application may restart it at that time. The software can detect this case by checking for an end of queue (EOQ) condition flag on the updated packet descriptor when it is returned by the EMAC.

Buffer Pointer

The buffer pointer is the byte-aligned memory address of the memory buffer associated with the buffer descriptor. The software application must set this value prior to adding the descriptor to the active receive list. This pointer is not altered by the EMAC.

Buffer Offset

This 16-bit field must be initialized to zero by the software application before adding the descriptor to a receive queue.

Whether or not this field is updated depends on the setting of the RXBUFFEROFFSET register. When the offset register is set to a non-zero value, the received packet is written to the packet buffer at an offset given by the value of the register, and this value is also written to the buffer offset field of the descriptor.

When a packet is fragmented over multiple buffers because it does not fit in the first buffer supplied, the buffer offset only applies to the first buffer in the list, which is where the start of packet (SOP) flag is set in the corresponding buffer descriptor. In other words, the buffer offset field is only updated by the EMAC on SOP descriptors.

The range of legal values for the BUFFEROFFSET register is 0 to (Buffer Length – 1) for the smallest value of buffer length for all descriptors in the list.

Buffer Length

This 16-bit field is used for two purposes:

1) Before the descriptor is first placed on the receive queue by the application software, the buffer length field is first initialized by the software to be the physical size of the empty data buffer pointed to by the buffer pointer field.

2) After the empty buffer has been processed by the EMAC and filled with received data bytes, the buffer length field is updated by the EMAC to reflect the actual number of valid data bytes written to the buffer.

Packet Length

This 16-bit field specifies the number of data bytes in the entire packet. This value is initialized to zero by the software application for empty packet buffers. The value is filled in by the EMAC on the first buffer used for a given packet. This is signified by the EMAC setting a start of packet (SOP) flag. The packet length is set by the EMAC on all SOP buffer descriptors.

Start of Packet (SOP) Flag

When set, this flag indicates that the descriptor points to a packet buffer that is the start of a new packet. In the case of a single fragment packet, both the SOP and end of packet (EOP) flags are set. Otherwise; the descriptor pointing to the last packet buffer for the packet has the EOP flag set. This flag is initially cleared by the software application before adding the descriptor to the receive queue. This bit is set by the EMAC on SOP descriptors.

End of Packet (EOP) Flag

When set, this flag indicates that the descriptor points to a packet buffer that is last for a given packet. In the case of a single fragment packet, both the start of packet (SOP) and EOP flags are set. Otherwise; the descriptor pointing to the last packet buffer for the packet has the EOP flag set. This flag is initially cleared by the software application before adding the descriptor to the receive queue. This bit is set by the EMAC on EOP descriptors.

Ownership (OWNER) Flag

When set, this flag indicates that the descriptor is currently owned by the EMAC. This flag is set by the software application before adding the descriptor to the receive descriptor queue. This flag is cleared by the EMAC once it is finished with a given set of descriptors, associated with a received packet. The flag is updated by the EMAC on SOP descriptor only. So when the application identifies that the OWNER flag is cleared on an SOP descriptor, it may assume that all descriptors up to and including the first with the EOP flag set have been released by the EMAC. (Note that in the case of single buffer packets, the same descriptor will have both the SOP and EOP flags set.)

End of Queue (EOQ) Flag

When set, this flag indicates that the descriptor in question was the last descriptor in the receive queue for a given receive channel, and that the corresponding receiver channel has halted. This flag is initially cleared by the software application prior to adding the descriptor to the receive queue. This bit is set by the EMAC when the EMAC identifies that a descriptor is the last for a given packet received (also sets the EOP flag), and there are no more descriptors in the receive list (next descriptor pointer is NULL).

The software application can use this bit to detect when the EMAC receiver for the corresponding channel has halted. This is useful when the application appends additional free buffer descriptors to an active receive queue. Note that this flag is valid on EOP descriptors only.

Teardown Complete (TDOWNCMPLT) Flag

This flag is used when a receive queue is being torn down, or aborted, instead of being filled with received data. This would happen under device driver reset or shutdown conditions. The EMAC sets this bit in the descriptor of the first free buffer when the tear down occurs. No additional queue processing is performed.

Pass CRC (PASSCRC) Flag

This flag is set by the EMAC in the SOP buffer descriptor if the received packet includes the 4-byte CRC. This flag should be cleared by the software application before submitting the descriptor to the receive queue.

Jabber Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet is a jabber frame and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

Oversize Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet is an oversized frame and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

Fragment Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet is only a packet fragment and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

Undersized Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet is undersized and was not discarded because the RXCSFEN bit was set in the RXMBPENABLE register.

Control Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet is a jabber frame and was not discarded because the RXCMFEN bit was set in the RXMBPENABLE register.

Overrun Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet was aborted due to a receive overrun.

Code Error (CODEERROR) Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet contained a code error and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

Alignment Error (ALIGNERROR) Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet contained an alignment error and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

CRC Error (CRCERROR) Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet contained a CRC error and was not discarded because the RXCEFEN bit was set in the RXMBPENABLE register.

No Match (NOMATCH) Flag

This flag is set by the EMAC in the SOP buffer descriptor, if the received packet did not pass any of the EMACs address match criteria and was not discarded because the RXCAFEN bit was set in the RXMBPENABLE register. Although the packet is a valid Ethernet data packet, it was only received because the EMAC is in promiscuous mode.

2.4 Media Independent Interface (MII)

The following sections discuss operation of the Media Independent Interface (MII) in 10 Mbps and 100 Mbps mode. An IEEE 802.3 compliant Ethernet MAC controls the interface.

2.4.1 Data Reception

2.4.1.1 Receive Control

Data received from the PHY is interpreted and output to the EMAC receive FIFO. Interpretation involves detection and removal of the preamble and start of frame delimiter, extraction of the address and frame length, data handling, error checking and reporting, cyclic redundancy checking (CRC), and statistics control signal generation. Address detection and frame filtering is performed outside the MII interface.

2.4.1.2 Receive Inter-Frame Interval

The 802.3 required inter-packet gap (IPG) is 24 MII clocks (96-bit times). However, the EMAC can tolerate a reduced IPG (2 MII clocks or 8-bit times) with a correct preamble and start frame delimiter. This interval between frames must comprise (in the following order):

1) An Inter-Packet Gap (IPG).
2) A seven octet preamble (all octets 55h).
3) A one octet start frame delimiter (5Dh).

2.4.1.3 Receive Flow Control

When enabled and triggered, receive flow control is initiated to limit the EMAC from further frame reception. Two forms of receive flow control are implemented, collision based for half-duplex mode, and IEEE 802.3X pause frames for full-duplex mode. In either case, receive flow control prevents frame reception by issuing the flow control appropriate for the current mode of operation. Receive flow control prevents reception of frames on the EMAC until all of the triggering conditions clear, at which time frames may again be received by the EMAC.

Receive flow control is enabled by the RXFLOWEN bit in the MACCONTROL register. The EMAC is configured for collision or IEEE 802.3X flow control via the FULLDUPLEX bit in the MACCONTROL register. Receive flow control is triggered when the number of free buffers in any enabled receive channel (RX*n*FREEBUFFER) is less than or equal to the channel flow control threshold register (RX*n*FLOWTHRESH) value. Receive flow control is independent of receive QOS, except that both use the free buffer values.

2.4.1.4 Collision-Based Receive Flow Control

Collision-based receive flow control provides a means of preventing frame reception when the EMAC is operating in half-duplex mode (FULLDUPLEX bit is cleared in MACCONTROL register). When receive flow control is enabled and triggered, the EMAC generates collisions for received frames. The jam sequence transmitted is the twelve byte sequence C3.C3.C3.C3.C3.C3.C3.C3.C3.C3.C3.C3h. The jam sequence begins no later than the source address starts to be received. Note that these forced collisions are not limited to a maximum of 16 consecutive collisions, and are independent of the normal back-off algorithm.

Receive flow control does not depend on the value of the incoming frame destination address. A collision is generated for any incoming packet, regardless of the destination address, if any EMAC enabled channel's free buffer register value is less than or equal to the channel's flow threshold value.

2.4.1.5 IEEE 802.3X Based Receive Flow Control

IEEE 802.3x based receive flow control provides a means of preventing frame reception when the EMAC is operating in full-duplex mode (FULLDUPLEX bit is set in MACCONTROL register). When receive flow control is enabled and triggered, the EMAC transmits a pause frame to request that the sending station stop transmitting for the period indicated within the transmitted pause frame.

The EMAC transmits a pause frame to the reserved multicast address at the first available opportunity (immediately if currently idle, or following the completion of the frame currently being transmitted). The pause frame contains the maximum possible value for the pause time (FFFFh). The EMAC counts the receive pause frame time (decrements FF00h to 0) and retransmits an outgoing pause frame, if the count reaches zero. When the flow control request is removed, the EMAC transmits a pause frame with a zero pause time to cancel the pause request.

Note that transmitted pause frames are only a request to the other end station to stop transmitting. Frames that are received during the pause interval are received normally (provided the receive FIFO is not full).

Pause frames are transmitted if enabled and triggered, regardless of whether or not the EMAC is observing the pause time period from an incoming pause frame.

The EMAC transmits pause frames as described:

- The 48-bit reserved multicast destination address 01.80.C2.00.00.01h.

- The 48-bit source address equal to the EMAC channel 0 address, regardless of whether channel 0 is enabled for reception or not (MACADDRL0).
- The 16-bit length/type field containing the value 88.08h.
- The 16-bit pause opcode equal to 00.01h.
- The 16-bit pause time value of FF.FFh. A pause-quantum is 512 bit-times. Pause frames sent to cancel a pause request have a pause time value of 00.00h.
- Zero padding to 64-byte data length (EMAC transmits only 64-byte pause frames).
- The 32-bit frame-check sequence (CRC word).

All quantities are hexadecimal and are transmitted most-significant-byte first. The least-significant-bit (LSB) is transferred first in each byte.

If the RXFLOWEN bit, in the MACCONTROL register, is cleared to 0 while the pause time is nonzero, then the pause time is cleared to zero and a zero count pause frame is sent.

2.4.2 Data Transmission

The EMAC passes data to the PHY from the transmit FIFO (when enabled). Data is synchronized to the transmit clock rate. Transmission begins when there are 128 bytes, or a complete packet, in the FIFO in both 10 Mbps and 100 Mbps mode. The smallest frame that can be sent is two bytes of data with four bytes of CRC (6-byte frame).

2.4.2.1 Transmit Control

A jam sequence is output if a collision is detected on a transmit packet. If the collision was late (after the first 64 bytes have been transmitted), the collision is ignored. If the collision is not late, the controller will back off before retrying the frame transmission. When operating in full-duplex mode, the carrier sense (CRS) and collision-sensing modes are disabled.

2.4.2.2 CRC Insertion

The MAC generates and appends a 32-bit Ethernet CRC onto the transmitted data, if the SOP buffer descriptor PASSCRC flag is cleared. For the EMAC-generated CRC case, a CRC (or placeholder) at the end of the data is allowed but not required. The buffer byte count value should not include the CRC bytes, if they are present.

If the SOP buffer descriptor PASSCRC flag is set, then the last four bytes of the transmit data are transmitted as the frame CRC. The four CRC data bytes should be the last four bytes of the frame and should be included in the buffer byte count value. The MAC performs no error checking on the outgoing CRC.

2.4.2.3 MTXER

The MII_MTXER signal is not used. If an underflow condition occurs on a transmitted frame, the frame CRC is inverted to indicate the error to the network.

2.4.2.4 Adaptive Performance Optimization (APO)

The EMAC incorporates adaptive performance optimization (APO) logic that may be enabled by setting the TXPACE bit in the MACCONTROL register. Transmission pacing to enhance performance is enabled when the TXPACE bit is set. Adaptive performance pacing introduces delays into the normal transmission of frames, delaying transmission attempts between stations, reducing the probability of collisions occurring during heavy traffic (as indicated by frame deferrals and collisions); thereby, increasing the chance of successful transmission.

When a frame is deferred, suffers a single collision, multiple collisions, or excessive collisions, the pacing counter is loaded with an initial value of 31. When a frame is transmitted successfully (without experiencing a deferral, single collision, multiple collision, or excessive collision), the pacing counter is decremented by 1, down to 0.

With pacing enabled, a new frame is permitted to immediately (after one IPG) attempt transmission only if the pacing counter is zero. If the pacing counter is non-zero, the frame is delayed by the pacing delay of approximately four interpacket gap delays. APO only affects the IPG preceding the first attempt at transmitting a frame; APO does not affect the back-off algorithm for retransmitted frames.

2.4.2.5 Interpacket-Gap (IPG) Enforcement

The measurement reference for the IPG of 96 bit times is changed depending on frame traffic conditions. If a frame is successfully transmitted without collision and MCRS is deasserted within approximately 48 bit times of MTXEN being deasserted, then 96 bit times is measured from MTXEN. If the frame suffered a collision or MCRS is not deasserted until more than approximately 48 bit times after MTXEN is deasserted, then 96 bit times (approximately, but not less) is measured from MCRS.

2.4.2.6 Back Off

The EMAC implements the 802.3 binary exponential back-off algorithm.

2.4.2.7 Transmit Flow Control

Incoming pause frames are acted upon, when enabled, to prevent the EMAC from transmitting any further frames. Incoming pause frames are only acted upon when the FULLDUPLEX and TXFLOWEN bits in the MACCONTROL register are set. Pause frames are not acted upon in half-duplex mode. Pause frame action is taken if enabled, but normally the frame is filtered and not transferred to memory. MAC control frames are transferred to memory, if the RXCMFEN bit in the RXMBPENABLE register is set. The TXFLOWEN and FULLDUPLEX bits affect whether or not MAC control frames are acted upon, but they have no affect upon whether or not MAC control frames are transferred to memory or filtered.

Pause frames are a subset of MAC control frames with an opcode field of 0001h. Incoming pause frames are only acted upon by the EMAC if:

- ❏ TXFLOWEN bit is set in MACCONTROL register
- ❏ The frame's length is 64 to RXMAXLEN bytes inclusive
- ❏ The frame contains no CRC error or align/code errors

The pause time value from valid frames is extracted from the two bytes following the opcode. The pause time is loaded into the EMAC's transmit pause timer and the transmit pause time period begins.

If a valid pause frame is received during the transmit pause time period of a previous transmit pause frame then:

- ❏ If the destination address is not equal to the reserved multicast address or the address in the MACADDRH, MACADDRM, and MACADDRL*n* registers, then the transmit pause timer immediately expires, or
- ❏ If the new pause time value is 0, then the transmit pause timer immediately expires, else
- ❏ The EMAC transmit pause timer immediately is set to the new pause frame pause time value. (Any remaining pause time from the previous pause frame is discarded).

If the TXFLOWEN bit in MACCONTROL is cleared, then the pause-timer immediately expires.

The EMAC does not start the transmission of a new data frame any sooner than 512-bit times after a pause frame with a non-zero pause time has finished being received (MRXDV going inactive). No transmission begins until the pause timer has expired (the EMAC may transmit pause frames in order to initiate outgoing flow control). Any frame already in transmission when a pause frame is received is completed and unaffected.

Incoming pause frames consist of:

- A 48-bit destination address equal to one of the following:
  - the reserved multicast destination address 01.80.C2.00.00.01h
  - Any EMAC unicast 48-bit address (MACADDRH, MACADDRM, MACADDRLn). Pause frames are accepted, regardless of whether the channel is enabled or not.
- The 48-bit source address of the transmitting device.
- The 16-bit length/type field containing the value 88.08h.
- The 16-bit pause opcode equal to 00.01h.
- The 16-bit pause_time. A pause-quantum is 512 bit-times.
- Padding to 64-byte data length.
- The 32-bit frame-check sequence (CRC word).

All quantities are hexadecimal and are transmitted most-significant-byte first. The least-significant-bit (LSB) is transferred first in each byte.

The padding is required to make up the frame to a minimum of 64 bytes. The standard allows pause frames longer than 64 bytes to be discarded or interpreted as valid pause frames. The EMAC recognizes any pause frame between 64 bytes and RXMAXLEN bytes in length.

2.4.2.8 Speed, Duplex, and Pause Frame Support

The MAC can operate at 10 Mbps or 100 Mbps, in half-duplex or full-duplex mode, and with or without pause frame support as configured by the host.

2.5 Packet Receive Operation

2.5.1 Receive DMA Host Configuration

To configure the receive DMA for operation the host must perform:

- ❏ Write the receive base address to the MACADDRH, MACADDRM, and MACADDRL*n* registers.
- ❏ Initialize the RX*n*HDP registers to zero.
- ❏ Write the MACHASH1 and MACHASH2 registers, if multicast addressing is desired.
- ❏ Initialize the RX*n*FREEBUFFER, RX*n*FLOWTHRESH, and RXFILTERLOWTHRESH registers, if flow control is to be enabled.
- ❏ Enable the desired receive interrupts using the RXINTMASKSET and RXINTMASKCLEAR registers.
- ❏ Set the appropriate configuration bits in the MACCONTROL register.
- ❏ Write the RXBUFFEROFFSET register value (typically zero).
- ❏ Setup the receive channel(s) buffer descriptors and initialize RX*n*HDP registers.
- ❏ Enable the receive DMA controller by setting the RXEN bit in the RXCONTROL register.
- ❏ Configure and enable the receive operation, as desired, in the RXMBPENABLE register and by using the RXUNICASTSET and RXUNICASTCLEAR registers.

2.5.2 Receive Channel Enabling

Each of the eight receive channels has an enable bit (RXCH*n*EN) that is controlled using the RXUNICASTSET and RXUNICASTCLEAR registers. The RXCH*n*EN bits determine whether the given channel is enabled (when set to 1) to receive frames with a matching destination address. The receive channels must be enabled starting with channel 0 and moving up without skipping channels. If only a single channel is to be used, then only channel 0 should be enabled. If two receive channels are used, then channels 0 and 1 should be enabled. If three channels are used, then channels 0 through 2 should be enabled, etc.

The BROADEN and MULTEN bits in the RXMBPENABLE register determine if broadcast and multicast frames, respectively, are enabled or filtered. If broadcast and multicast frames are enabled, then they are copied to only a single channel selected by the BROADCH and MULTCH bits of RXMBPENABLE. The PROMCH bits select the promiscuous channel to receive frames selected by the RXCMFEN, RXCSFEN, RXCEFEN, and RXCAFEN bits. These four bits allow reception of MAC control frames, short frames, error frames, and all frames (promiscuous), respectively.

2.5.3 Receive Channel Addressing

The 48-bit address for each receive channel is determined by concatenating the 32-bit MACADDRH register value, the 8-bit MACADDRM register value, and the 8-bit MACADDRL*n* register value. All eight MACADDRL*n* registers should be initialized, because pause frames are acted upon regardless of whether a channel is enabled or disabled. If only one channel is to be enabled, it is permissible to replicate the MAC address of that channel across multiple MACADDRL*n* registers to avoid consuming multiple MAC addresses.

Incoming group addresses (multicast) are hashed into an index in the hash table. If the indexed bit is set, the frame is copied to the selected channel memory (MULTCH) when multicast frames are enabled by setting the MULTEN bit in the RXMBPENABLE register. The multicast hash bits are set in the MACHASH1 and MACHASH2 registers.

2.5.4 Hardware Receive QOS Support

Hardware receive quality of service (QOS) is supported, when enabled, by the Tag Protocol Identifier format and the associated Tag Control Information (TCI) format priority field. When the incoming frame length/type value is equal to 81.00h, the EMAC recognizes the frame as an Ethernet Encoded Tag Protocol Type. The two octets immediately following the protocol type contain the 16-bit TCI field. Bits 15–13 of the TCI field contain the received frames priority (0 to 7). The received frame is a low-priority frame, if the priority value is 0 to 3; the received frame is a high-priority frame, if the priority value is 4 to 7. All frames that have a length/type field value not equal to 81.00h are low-priority frames.

Received frames that contain priority information are determined by the EMAC as:

- A 48-bit (6-octet) destination address equal to:
  - The destination station's individual address (MACADDRH, MACADDRM, and MACADDRL*n* registers).
  - The destination station's multicast address (MACHASH1 and MACHASH2 registers).
  - The broadcast address of all ones.

- A 48-byte (6-octet) source address.
- The 16-bit (2-octet) length/type field containing the value 81.00h.
- The 16-bit (2-octet) TCI field with the priority field in the upper 3 bits.
- Data octets
- The 4-octet CRC.

The RXFILTERLOWTHRESH and the RX*n*FREEBUFFER registers are used in conjunction with the priority information to implement receive hardware QOS. Low-priority frames are filtered, if the number of free buffers (RX*n*FREEBUFFER) for the frame channel is less than or equal to the filter low threshold (RXFILTERLOWTHRESH) value. Hardware QOS is enabled by the RXQOSEN bit in the RXMBPENABLE register.

2.5.5 Host Free Buffer Tracking

The host must track free buffers for each enabled channel (including unicast, multicast, broadcast, and promiscuous), if receive QOS or receive flow control is used. Disabled channel free buffer values are don't cares. During initialization, the host should write the number of free buffers for each enabled channel to the appropriate RX*n*FREEBUFFER register. The EMAC decrements the appropriate channel's free buffer value for each buffer used. When the host reclaims the frame buffers, the host should write the channel free buffer register with the number of reclaimed buffers (write to increment). There are a maximum of 65 535 free buffers available. The RX*n*FREEBUFFER registers only need to be updated by the host if receive QOS or flow control is used.

2.5.6 Receive Channel Teardown

The host commands a receive channel teardown by writing the channel number to the RXTEARDOWN register. When a teardown command is issued to an enabled receive channel, the following occurs:

- Any current frame in reception completes normally.
- The TDOWNCMPLT flag is set in the next buffer descriptor in the chain, if there is one.
- The channel head descriptor pointer is cleared to 0.
- A receive interrupt for the channel is issued to the host.
- The corresponding RX*n*INTACK register contains the value FFFF FFFCh.
- The host should acknowledge a teardown interrupt with an FFFF FFFCh acknowledge value.

Channel teardown may be commanded on any channel at any time. The host is informed of the teardown completion by the set teardown complete buffer descriptor bit. The EMAC does not clear any channel enables due to a teardown command. A teardown command to an inactive channel issues an interrupt that software should acknowledge with an FFFF FFFCh acknowledge value to RX*n*INTACK (note that there is no buffer descriptor in this case). Software may read RX*n*INTACK to determine if the interrupt was due to a commanded teardown. The read value is FFFF FFFCh, if the interrupt was due to a teardown command.

2.5.7 Receive Frame Classification

Received frames are proper (good) frames, if they are between 64 and RXMAXLEN in length (inclusive) and contain no errors (code/align/CRC).

Received frames are long frames, if their frame count exceeds the value in the RXMAXLEN register. The RXMAXLEN register reset (default) value is 5EEh (1518). Long received frames are either oversized or jabber frames. Long frames with no errors are oversized frames; long frames with CRC, code, or alignment errors are jabber frames.

Received frames are short frames, if their frame count is less than 64 bytes. Short frames that address match and contain no errors are undersized frames; short frames with CRC, code, or alignment errors are fragment frames. If the frame count is less than or equal to 20, then the frame CRC is passed regardless of whether the RXPASSCRC bit is set or cleared in the RXMBPENABLE register.

A received long packet always contains RXMAXLEN number of bytes transferred to memory (if the RXCEFEN bit is set in RXMBPENABLE) regardless of the value of the RXPASSCRC bit. An example with RXMAXLEN set to 1518:

- If the frame length is 1518, then the packet is not a long packet and there are 1514 or 1518 bytes transferred to memory depending on the value of the RXPASSCRC bit.
- If the frame length is 1519, there are 1518 bytes transferred to memory regardless of the the RXPASSCRC bit value. The last three bytes are the first three CRC bytes.
- If the frame length is 1520, there are 1518 bytes transferred to memory regardless of the RXPASSCRC bit value. The last two bytes are the first two CRC bytes.
- If the frame length is 1521, there are 1518 bytes transferred to memory regardless of the RXPASSCRC bit value. The last byte is the first CRC byte.

- If the frame length is 1522, there are 1518 bytes transferred to memory. The last byte is the last data byte.

2.5.8 Promiscuous Receive Mode

When the promiscuous receive mode is enabled, nonaddress matching frames that would normally be filtered are transferred to the promiscuous channel and address matching frames that would normally be filtered are transferred to the address match channel. A frame is considered to be an address matching frame only if it is enabled to be received on a unicast, multicast, or broadcast channel. Frames received to disabled unicast, multicast, or broadcast channels are considered nonaddress matching. A single channel is selected as the promiscuous channel by the PROMCH bits in the RXMBPENABLE register. The promiscuous receive mode is enabled by the RXCMFEN, RXCEFEN, RXCSFEN, and RXCAFEN bits in RXMBPENABLE. Table 2–1 shows the effects of the promiscuous enable bits. Proper frames are frames that are between 64 and RXMAXLEN bytes in length inclusive and contain no errors (code/align/CRC).

Table 2–1. Receive Frame Treatment Summary

	RXMBPENABLE Bits				
ADDR MATCH	RXCAFEN	RXCEFEN	RXCMFEN	RXCSFEN	Frame Treatment
0	0	0	0	0	No frames transferred
0	0	0	0	1	Undersized frames (not fragments) transferred to promiscuous channel.
0	0	0	1	0	Control frames (without errors) transferred to promiscuous channel.
0	0	0	1	1	Control/undersize frames (without errors) transferred to promiscuous channel
0	0	1	0	0	All frames with errors transferred to promiscuous channel (jabber/fragment/CRC/code/align).
0	0	1	0	1	All frames with errors and undersize frames transferred to promiscuous channel.
0	0	1	1	0	All frames with errors and control frames transferred to promiscuous channel.

Table 2–1. Receive Frame Treatment Summary (Continued)

ADDR MATCH	RXMBPENABLE Bits				Frame Treatment
	RXCAFEN	RXCEFEN	RXCMFEN	RXCSFEN	
0	0	1	1	1	All frames with errors, control frames, and short frames transferred to promiscuous channel.
0	1	0	0	0	Proper frames transferred to promiscuous channel. All other frames filtered.
0	1	0	0	1	Proper/undersized frames (no errors) transferred to promiscuous channel. All others filtered.
0	1	0	1	0	Proper/control frames (no errors) transferred to promiscuous channel. All others filtered.
0	1	0	1	1	Proper/undersized/control frames (no errors) transferred to promiscuous channel. All error frames filtered (oversize/jabber/fragment/code/align/CRC).
0	1	1	0	0	Proper/oversize/jabber/fragment/code/align/ CRC frames transferred to promiscuous channel. Control and undersized frames with no errors filtered.
0	1	1	0	1	Proper/undersized/fragment/oversize/jabber /code/align/CRC frames transferred to promiscuous channel. Control frames with no errors filtered.
0	1	1	1	0	Proper/fragment/control/oversize/jabber/ code/align/CRC frames transferred to promiscuous channel. Undersized frames filtered.
0	1	1	1	1	All non-address matching frames with and without errors transferred to promiscuous channel.
1	X	0	0	0	Proper frames transferred to address match channel. All others filtered.
1	X	0	0	1	Proper/undersized frames transferred to address match channel. All others filtered

Table 2–1. Receive Frame Treatment Summary (Continued)

ADDR MATCH	RXMBPENABLE Bits				Frame Treatment
	RXCAFEN	RXCEFEN	RXCMFEN	RXCSFEN	
1	X	0	1	0	Proper/control frames (no errors) transferred to address match channel. All others frames filtered.
1	X	0	1	1	Proper/control/undersized frames transferred to address match channel. All other frames filtered.
1	X	1	0	0	Proper/oversize/jabber/fragment/code/align/ CRC frames transferred to address match channel. Undersized/control frames (no errors) filtered.
1	X	1	0	1	Proper/oversize/jabber/fragment/undersized /code/align/CRC frames transferred to address match channel. Control frames (no errors) filtered.
1	X	1	1	0	Proper/oversize/jabber/fragment/control/cod e/align/CRC frames transferred to address match channel. Undersized frames (no errors) filtered.
1	X	1	1	1	All address matching frames with and without errors transferred to the address match channel

2.5.9 Receive Overrun

The types of receive overrun are:

- ❑ FIFO start of frame overrun (FIFO_SOF)
- ❑ FIFO middle of frame overrun (FIFO_MOF)
- ❑ DMA start of frame overrun (DMA_SOF)
- ❑ DMA middle of frame overrun (DMA_MOF)

The statistics counters used to track the types of receive overrun are:

- ❑ Receive Start of Frame Overruns Register (RXSOFOVERRUNS)
- ❑ Receive Middle of Frame Overruns Register (RXMOFOVERRUNS)
- ❑ Receive DMA Overruns Register (RXDMAOVERRUNS)

Start of frame overruns have no resources available when frame reception begins. Start of frame overruns increment the appropriate overrun statistic(s) and the frame is filtered.

Middle of frame overruns have the resources to start the frame reception, but run out of resources during frame reception. In normal operation, a frame that overruns after starting the frame reception is filtered and the appropriate statistic(s) are incremented; however, the RXCEFEN bit in the RXMBPENABLE register affects overrun frame treatment. Table 2–2 shows how the overrun condition is handled for the middle of frame overrun.

Table 2–2. Middle of Frame Overrun Treatment

ADDR MATCH	RXCEFEN	Middle of Frame Overrun Treatment
0	0	Overrun frame filtered.
0	1	As much frame data as possible is transferred to the promiscuous channel until overrun. The appropriate overrun statistic(s) is incremented and the OVERRUN and NOMATCH flags are set in the SOP buffer descriptor. Note that the RXMAXLEN number of bytes cannot be reached for an overrun to occur (it would be truncated and be a jabber or oversize).
1	0	Overrun frame filtered with the appropriate overrun statistic(s) incremented.
1	1	As much frame data as possible is transferred to the address match channel until overrun. The appropriate overrun statistic(s) is incremented and the OVERRUN flag is set in the SOP buffer descriptor. Note that the RXMAXLEN number of bytes cannot be reached for an overrun to occur (it would be truncated).

2.6 Packet Transmit Operation

The transmit DMA is an eight channel interface. Priority between the eight queues may be either fixed or round robin as selected by TXPTYPE bit in the MACCONTROL register. If the priority type is fixed, then channel 7 has the highest priority and channel 0 has the lowest priority. Round robin priority proceeds from channel 0 to channel 7.

2.6.1 Transmit DMA Host Configuration

To configure the transmit DMA for operation the host must perform:

- ❑ Write the base address to the MACADDRH, MACADDRM, and MACADDRL*n* registers (used for pause frames on transmit).
- ❑ Initialize the TX*n*HDP registers to zero.
- ❑ Enable the desired transmit interrupts using the TXINTMASKSET and TXINTMASKCLEAR registers.
- ❑ Set the appropriate configuration bits in the MACCONTROL register.
- ❑ Enable the transmit DMA controller by setting the TXEN bit in the TXCONTROL register.
- ❑ Setup the transmit buffer descriptors and write the appropriate TX*n*HDP registers with the pointer to the first descriptor to start transmit operations.

2.6.2 Transmit Channel Teardown

The host commands a transmit channel teardown by writing the channel number to the TXTEARDOWN register. When a teardown command is issued to an enabled transmit channel, the following occurs:

- ❑ Any current frame in transmission completes normally.
- ❑ The TDOWNCMPLT flag is set in the next SOP buffer descriptor in the chain, if there is one.
- ❑ The channel head descriptor pointer is cleared to 0.
- ❑ An interrupt is issued to inform the host of the channel teardown.
- ❑ The corresponding TX*n*INTACK register contains the value FFFF FFFCh.
- ❑ The host should acknowledge a teardown interrupt with an FFFF FFFCh acknowledge value.

Channel teardown may be commanded on any channel at any time. The host is informed of the teardown completion by the set teardown complete buffer descriptor bit. The EMAC does not clear any channel enables due to a teardown command. A teardown command to an inactive channel issues an interrupt that software should acknowledge with an FFFF FFFCh acknowledge value to TX*n*INTACK (note that there is no buffer descriptor in this case). Software may read TX*n*INTACK to determine if the interrupt was due to a commanded teardown. The read value is FFFF FFFCh, if the interrupt was due to a teardown command.

2.7 EMAC Module Interrupts

The EMAC control module combines different interrupt signals from both the EMAC and MDIO modules and generates a single interrupt signal that is wired to the DSP interrupt mux. Once this interrupt is generated, the reason for the interrupt can be read from the MACINVECTOR register. MACINVECTOR combines the following interrupt signals: TXPEND*n*, RXPEND*n*, STATPEND, HOSTPEND, LINKINT, and USERINT.

The LINKINT and USERINT interrupt bits are associated with MDIO operation and are explained in Chapter 3, *MDIO Module*.

2.7.1 Transmit and Receive Interrupts

When the EMAC completes the reception or transmission of a frame, the EMAC writes the appropriate interrupt acknowledgement register (RX*n*INTACK or TX*n*INTACK) with the address of the last buffer descriptor used for the transfer of the frame data. The appropriate host interrupt pending (RXPEND*n* or TXPEND*n*), is then issued if enabled by the interrupt mask. On an interrupt from the EMAC, the host processes the buffer chain. When the host completes the buffer chain processing, the host writes the address of the last processed buffer descriptor to the appropriate interrupt acknowledge register. The host write does not actually change the register value. The data written by the host (buffer descriptor address of the last processed buffer) is compared to the data in the register written by the EMAC (address of last buffer descriptor used by the EMAC). If the two values are not equal, the channel pending interrupt signal remains asserted; if the two values are equal, the pending interrupt is cleared. The actual memory value is changed only by the EMAC. The value that the EMAC is expecting can be found by reading RX*n*INTACK or TX*n*INTACK.

If the two values are not equal, the EMAC has received or transmitted more frame(s) since the interrupt was issued, or the host did not process all buffers in the chain that were available to be processed when the interrupt was issued, so the pending interrupt remains asserted.

2.7.2 Statistics Interrupt

The statistics interrupt (STATPEND) is issued if enabled when any statistics value is greater than or equal to 8000 0000h (if enabled by the STATINT bit in the MACINTMASKSET register). The statistics interrupt is removed by writing to decrement any statistics value greater than 8000 0000h. So long as the most-significant-bit of any statistics value is set, the interrupt remains asserted.

2.7.3 Host Error Interrupt

The host error interrupt (HOSTPEND) is issued under any of several error conditions, dealing with the handling of buffer descriptors. The failure of the software application to supply properly formatted descriptors results in this error. The error bit can only be cleared by resetting the EMAC module.

2.7.4 Proper Interrupt Processing

All the interrupts signaled from the EMAC module are level driven, so if they remain active, their level remains constant; the DSP core requires edge-triggered interrupts. In order to properly convert the level-driven interrupt signal to an edge-triggered signal, the application software must make use of the interrupt control logic contained in the EMAC control module.

Section 2.2.5, *Interrupt Control*, discusses the EWCTL register contained in the EMAC control module. For safe interrupt processing, upon entry to the ISR, the software application should disable interrupts using the EWCTL register, and then reenable them upon leaving the ISR. If any interrupt signals are active at that time, this creates another rising edge on the interrupt signal going to the DSP interrupt mux, thus triggering another interrupt. The EWCTL register also uses the EWINTTCNT register to implement interrupt pacing.

2.8 Receive and Transmit Latency

The transmit and receive FIFOs each contains three 64-byte cells. The EMAC begins transmission of a packet on the wire after two FIFO cells, or a complete packet, are available in the FIFO. Transmit underrun cannot occur for packet sizes of 128 bytes or less. For larger packet sizes, transmit underrun can occur if the memory latency is greater than 5.12 μs in 100 Mbit mode (51.2 μs in 10 Mbit mode). The memory latency is the time required to transmit the next 64-byte cell into the FIFO. The latency time includes any required buffer descriptor reads for the cell data.

Receive overrun is prevented if the receive memory cell latency is 5.12 μs maximum. The latency time includes any required buffer descriptor reads for the cell data. Latency to DSP internal and external RAM can be controlled through the use of the transfer node priority setting in the EMAC control module. Latency to descriptor RAM is low because RAM is local to the EMAC, as it is part of the EMAC control module.

Chapter 3

MDIO Module

This chapter discusses the architecture and basic function of the MDIO module. Although the entire feature set of the MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported MDIO features.

Topic **Page**

3.1 MDIO Introduction

The management data input/output (MDIO) module is used to manage up to 32 physical layer (PHY) devices connected to the ethernet media access controller (EMAC). The MDIO module is designed to allow almost transparent operation of the MDIO interface with little maintenance from the DSP.

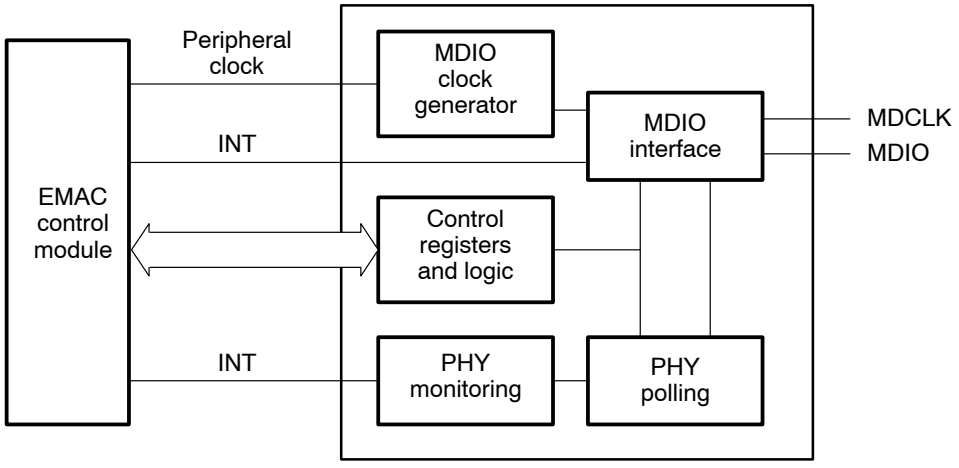
The MDIO module continuously polls all 32 MDIO addresses in order to enumerate all PHY devices in the system. Once a PHY device has been selected by the DSP, the MDIO module reads the PHY status register to monitor the PHY link state. Link change events are stored in the MDIO module, which can interrupt the DSP. This storing of the events allows the DSP to poll the link status of the PHY device without continuously performing MDIO module accesses. However, when the DSP must access the MDIO module for configuration and negotiation, the MDIO module performs the MDIO read or write operation independent of the DSP. This independent operation allows the DSP to poll for completion or interrupting the DSP once the operation has completed.

3.2 MDIO Module Components

The MDIO module (Figure 3−1) interfaces to the outside world through two MDIO pins (MDCLK and MDIO), and to the DSP core through the EMAC control module. The MDIO module consists of the following logical components:

- MDIO clock generator
- Global PHY detection and link state monitoring
- Active PHY monitoring
- PHY register user access

Figure 3−1. MDIO Module Block Diagram



3.2.1 MDIO Clock Generator

The MDIO clock generator controls the MDIO clock based from a divide-down of the peripheral clock (CPUclk/4) in the EMAC control module. The MDIO clock is specified to run up to 2.5 MHz, although typical operation would be 1.0 MHz. Since the peripheral clock frequency is variable (CPU/4), the application software or driver controls the divide-down amount.

3.2.2 Global PHY Detection and Link State Monitoring

The MDIO module continuously polls all 32 MDIO addresses in order to enumerate all PHY devices in the system. The module tracks whether or not a PHY on a particular address has responded, and whether or not the PHY currently has a link. Using this information allows the software application to quickly determine which MDIO address a PHY is using and when more than one PHY is used in a system, quickly switch between PHYs based on their current link status.

3.2.3 Active PHY Monitoring

Once a PHY candidate has been selected for use, the MDIO module transparently monitors its link state by reading the PHY status register. Link change events are stored on the MDIO device and can optionally interrupt the DSP. This allows the DSP to poll the link status of the device without continuously performing costly MDIO accesses. Up to two PHY devices can be actively monitored at any given time.

3.2.4 PHY Register User Access

When the DSP must access MDIO for configuration and negotiation, the PHY access module performs the actual MDIO read or write operation independent of the DSP. This allows the DSP to poll for completion or receive an interrupt when the read or write operation has been performed. There are two user access registers (USERACCESS0 and USERACCESS1), allowing the software to submit up to two access requests simultaneously. (The requests are processed sequentially.)

3.3 MDIO Module Operational Overview

The MDIO module implements the 802.3 serial management interface to simultaneously interrogate and control up to two ethernet PHYs simultaneously using a shared two-wire bus. It separately performs autodetection and records the current link status of up to 32 PHYs, polling all 32 MDIO addresses.

Host software uses the MDIO module to configure the autonegotiation parameters of the primary PHY attached to the EMAC, retrieve the negotiation results, and configure required parameters in the EMAC.

Up to two ethernet PHYs can be directly controlled and queried. The Media Independent Interface (MII) addresses of these two PHY devices are specified in the PHYADDR bits of the USERPHYSEL*n* register. The module can be programmed to trigger a DSP interrupt on a PHY link change event, by setting the LINKINTENB bit in USERPHYSEL*n*.

Reads and writes to registers in these PHY devices is performed using the USERACCESS*n* register.

The MDIO module powers up in an idle state until specifically enabled by setting the ENABLE bit in the CONTROL register. At this time, the MDIO clock divider and preamble mode selection is also configured. The MDIO preamble can be disabled when none of the connected PHYs require it (it is enabled by default).

Once the MDIO module is enabled, the MDIO interface state machine continuously polls the PHY link status (by reading the Generic Status Register) of all possible 32 PHY addresses and records the results in the ALIVE and LINK registers. The corresponding bit for each PHY (0–31) is set in the ALIVE register, if the PHY responded to the read request; the bit is set in the LINK register, if the PHY responded and also is currently linked. In addition, any PHY register read transactions initiated by the application software using the USERACCESS*n* register also causes the ALIVE register to be updated.

The link status of two of the 32 possible PHY addresses is tracked through the use of the USERPHYSEL*n* register. A change in the link status of the two PHYs being monitored sets the appropriate bit in the LINKINTRAW and LINKINTMASKED registers, if enabled by the LINKINTENB bit in USERPHYSEL*n*.

While the MDIO module is enabled, the host can issue a read or write transaction over the MII management interface using the DATA, PHYADR, REGADR, and WRITE bits in the USERACCESS*n* register. When the application sets the GO bit in USERACCESS*n*, the MDIO module begins the transaction without any further intervention from the DSP. Upon completion, the MDIO module clears the GO bit and sets the MAC*n* bit in the USERINTRAW register corresponding to USERACCESS*n* used. The corresponding MAC*n* bit in the USERINTMASKED register may also be set, depending on the mask setting configured in the USERINTMASKSET and USERINTMASKCLEAR registers. A round-robin arbitration scheme is used to schedule transactions that may be queued using both USERACCESS0 and USERACCESS1. The application software must check the status of the GO bit in USERACCESS*n* before initiating a new transaction, to ensure that the previous transaction has completed. The application software can use the ACK bit in USERACCESS*n* to determine the status of a read transaction.

3.3.1 Initializing the MDIO Module

The following steps are performed by the application software or device driver to initialize the MDIO device:

1) Configure the PREAMBLE and CLKDIV bits in the CONTROL register.

2) Enable the MDIO module by setting the ENABLE bit in the CONTROL register.

3) The ALIVE register can be read after a delay to determine which PHYs responded, and the LINK register can determine which of those (if any) already have a link.

4) Setup the appropriate PHY addresses in the USERPHYSEL*n* register(s), and set the LINKINTENB bit to enable a link change event interrupt is desirable.

5) If an interrupt on general MDIO register access is desired, set the MAC*n* bit in the USERINTMASKSET register for the USERACCESS*n* register to be used. If only one PHY is to be used, one of the USERACCESS*n* registers can be setup to trigger a completion interrupt and the other register is not setup.

3.3.2 Writing Data to a PHY Register

The MDIO module includes a user access register (USERACCESS*n*) to directly access a specified PHY device. To write a PHY register, perform the following:

1) Check to ensure that the GO bit in the USERACCESS*n* register is cleared.

2) Write to the GO, WRITE, REGADR, PHYADR, and DATA bits in USERACCESS*n* corresponding to the PHY and PHY register you wish to write to.

3) The write operation to the PHY is scheduled and completed by the MDIO module. Completion of the write operation can be determined by polling the GO bit in USERACCESS*n* for a 0.

4) Completion of the operation sets the MAC*n* bit in the USERINTRAW register for the USERACCESS*n* used. If interrupts have been enabled on this bit using the USERINTMASKSET register, then the bit is also set in the USERINTMASKED register and an interrupt is triggered on the DSP.

3.3.3 Reading Data From a PHY Register

The MDIO module includes a user access register (USERACCESS*n*) to directly access a specified PHY device. To read a PHY register, perform the following:

1) Check to ensure that the GO bit in the USERACCESS*n* register is cleared.

2) Write to the GO, REGADR, and PHYADR bits in USERACCESS*n* corresponding to the PHY and PHY register you wish to read from.

3) The read data value is available in the data bits of USERACCESS*n* after the module completes the read operation on the serial bus. Completion of the read operation can be determined by polling the GO and ACK bits in USERACCESS*n*. Once the GO bit has cleared, the ACK bit is set on a successful read.

4) Completion of the operation sets the MAC*n* bit in the USERINTRAW register for the USERACCESS*n* used. If interrupts have been enabled on this bit using the USERINTMASKSET register, then the bit is also set in the USERINTMASKED register and an interrupt is triggered on the DSP.

3.4 MDIO Module Interrupts

The EMAC control module combines different interrupt signals from both the EMAC and MDIO modules and generates a single interrupt signal that is wired to the DSP interrupt mux. Once this interrupt is generated, the reason for the interrupt can be read from the MACINVECTOR register in the EMAC. MACINVECTOR combines the following interrupt signals: TXPEND*n*, RXPEND*n*, STATPEND, HOSTPEND, LINKINT, and USERINT.

The TXPEND*n*, RXPEND*n*, STATPEND, and HOSTPEND interrupt bits are associated with EMAC operation and are explained in Chapter 2, *EMAC Module*.

3.4.1 Link Change Interrupt

The MDIO module asserts a link change interrupt (LINKINT) if there is a change in the link state of the PHY corresponding to the address in the PHYADDR bits in the USERPHYSEL*n* register and the LINKINTENB bit is also set in USERPHYSEL*n*. This interrupt event is also captured in the MAC*n* bits of the LINKINTRAW register. MAC0 and MAC1 correspond to USERPHYSEL0 and USERPHYSEL1, respectively.

When the interrupt is enabled and generated, the corresponding MAC*n* bit is also set in the LINKINTMASKED register. The interrupt is cleared writing back the same bit to LINKINTMASKED (write to clear).

3.4.2 User Access Completion Interrupt

When the GO bit in one of the USERACCESS*n* registers transitions from 1 to 0 (indicating completion of a user access) and the MAC*n* bit in the USERINTMASKSET register corresponding to USERACCESS0 or USERACCESS1 is set, a user access completion interrupt (USERINT) is asserted. This interrupt event is also captured in the MAC*n* bits of the USERINTRAW register. MAC0 and MAC1 correspond to USERACCESS0 and USERACCESS1, respectively.

When the interrupt is enabled and generated, the corresponding MAC*n* bit is also set in the USERINTMASKED register. The interrupt is cleared writing back the same bit to USERINTMASKED (write to clear).

3.4.3 Proper Interrupt Processing

All the interrupts signaled from the MDIO module are level driven, so if they remain active, their level remains constant; the DSP core requires edge-triggered interrupts. In order to properly convert the level-driven interrupt signal to an edge-triggered signal, the application software must make use of the interrupt control logic contained in the EMAC control module.

Section 2.2.5, *Interrupt Control*, discusses the EWCTL register contained in the EMAC control module. For safe interrupt processing, upon entry to the ISR, the software application should disable interrupts using the EWCTL register, and then reenable them upon leaving the ISR. If any interrupt signals are active at that time, this creates another rising edge on the interrupt signal going to the DSP interrupt mux, thus triggering another interrupt. The EWCTL register also uses the EWINTTCNT register to implement interrupt pacing.

Chapter 4

Software Operation

This chapter discusses the software interface used to operate the EMAC and MDIO modules. It describes in detail how to initialize and maintain Ethernet operation in a software application or device driver.

The example code excerpts are taken from EMAC and MDIO Chip Support Library (CSL) functions. The full source code that includes these examples can be found in the CSL library source.

There are may different approaches in structuring an Ethernet software applications or device driver. It is important to keep in mind that this chapter documents the particular approach to programming the EMAC used by CSL, and it is not intended to show the only possible methodology.

This chapter is broken down into the EMAC control module, EMAC module, and MDIO module. The operation of these modules is described in Chapters 2 and 3. This chapter illustrates one software application approach based on that information.

Topic **Page**

4.1 Module Function Overview

This section summarizes the function of each module.

4.1.1 EMAC Control Module

The EMAC control module is used for global reset control, global interrupt enable, and to pace back to back interrupts using an interrupt retrigger count based on the peripheral clock (CPUclk/4). There is also a register to configure the priority of EMAC global memory accesses at the DSP's transfer controller, and a 4K block of RAM local to the EMAC that is used to hold packet buffer descriptors.

Note that although the EMAC control module and the EMAC module have slightly different functions, they are not distinguished from each other in the CSL API. The CSL HAL and function libraries refer to both as EMAC. Also, in practice, the type of maintenance performed on the EMAC control module is more commonly conducted from the EMAC module software (as opposed to the MDIO module).

4.1.2 EMAC Module

The EMAC module is used to send and receive Ethernet packets. This is done by maintaining up to 8 transmit and receive descriptor queues. The EMAC module configuration must also be kept up-to-date based on PHY negotiation results returned from the MDIO module.

4.1.3 MDIO Module

The MDIO module is used to initially configure the external PHY device, monitor the PHY, and relay any changes back to the software controlling the EMAC module. The MDIO module software can be a simple implementation to maintain one specific PHY or can maintain the status of multiple PHYs and autoselect the best PHY for use at any given time.

4.2 Target Environment

For the purposes of this example code in this chapter, some assumptions are made about the target environment. These assumptions are based on most commonly used configuration of the device (and actually go beyond the base functionality of a device driver). The desired feature set of the target environment is listed below. This is not intended to represent all the potential features of the EMAC system, but only those most commonly used in an application.

- The EMAC module uses a DSP interrupt for servicing transmit, receive, and EMAC status.
- The MDIO module uses a half-second polling loop to update PHY selection and status monitoring.
- There can be one or more PHYs connected to the DSP (although only one is in use at any given time).
- There is a single receive channel for unicast, broadcast, multicast, and promiscuous packets.
- The driver will not receive any type error packets.
- There are eight transmit channels. These can be placed in round-robin or fixed-priority mode. The mode is selected at run time.

4.3 EMAC Control Module Operation

The EMAC control module is used to reset the EMAC and MDIO modules, set-up the memory access priority at the DSP transfer controller, and control device interrupts. The EMAC control module registers are considered part of the EMAC module, and its initialization is combined with that of the EMAC module.

4.3.1 Initialization

The initialization of the EMAC control module consists of two parts:

- Configuration of the interrupt on the DSP.
- Initialization of the EMAC control module:
  - Resetting the EMAC and MDIO modules (using EWCTL)
  - Configuring the transfer node priority (using EWTRCTRL)
  - Setting the interrupt pace count (using EWINTTCNT)
  - Initializing the EMAC and MDIO modules.
  - Enabling interrupts in the EMAC control module (using EWCTL)

The code to perform these actions may appear as in Figure 4–1.

The process of mapping the EMAC interrupts to one of the DSP's interrupts is done using the DSP interrupt mux. For details, see Interrupt Selector and External Interrupts in the *TMS320C6000 DSP Peripherals Overview Reference Guide* (SPRU190). The interrupt mux code for the EMAC is `11000b`.

Once the interrupt is mapped to a DSP interrupt, general masking and unmasking of the interrupt (to control reentrancy) should be done at the DSP level by manipulating the DSP interrupt enable mask. The EMAC control module control register (EWCTL) should only be used to enable and disable interrupts from within the EMAC interrupt service routine (ISR). This is because disabling and reenabling the interrupt in EWCTL also resets the interrupt pace counter.

4.3.2 Monitoring

There is little monitoring that needs to be done on the EMAC control module. The EMAC driver uses the EMAC control module internal RAM for its packet buffer descriptors, and EWCTL and EMAC control module interrupt timer count register (EWINTTCNT) are used to control interrupts and interrupt pacing from within the EMAC ISR.

In the event of a fatal error condition, EWCTL can also be used to reset the EMAC and/or MDIO module.

Figure 4−1. EMAC Control Module Initialization Code

```
Uint32 tmpval;

/*
// Globally disable EMAC/MDIO interrupts in wrapper and put both
// EMAC and MDIO modules into reset
*/
EMAC_RSET( EWCTL, EMAC_FMKS( EWCTL, INTEN, DISABLE ) |
                  EMAC_FMKS( EWCTL, EMACRST, YES ) |
                  EMAC_FMKS( EWCTL, MDIORST, YES ) );

/* Wait about 100 cycles */
for( i=0; i<5; i++ )
    tmpval = EMAC_RGET( EWCTL );

/* Leave EMAC/MDIO interrupts disabled and take both
   EMAC and MDIO modules out of reset */
EMAC_RSET( EWCTL, EMAC_FMKS( EWCTL, INTEN, DISABLE ) |
                    EMAC_FMKS( EWCTL, EMACRST, NO ) |
                  EMAC_FMKS( EWCTL, MDIORST, NO ) );

/* Wait about 100 cycles */
for( i=0; i<5; i++ )
    tmpval = EMAC_RGET( EWCTL );

/* Set EMAC Priority to "1", allocation reqs "3" */
EMAC_RSET( EWTRCTRL, 0x13 );

/* Set Interrupt Timer Count (CPUclk/4) */
EMAC_RSET( EWINTTCNT, 1500 );

/*
// Initialize MDIO and EMAC Module
*/
```

[Discussed Later in this document]

```
/* Enable global interrupt in wrapper */
EMAC_FSETS( EWCTL, INTEN, ENABLE );
```

4.4 MDIO Module Operation

The MDIO module is used to configure and monitor one or more PHY devices that are connected to the EMAC module.

The software described is written to be a stand-alone module that acts as a slave to the EMAC software. After being initially configured, the MDIO software is entirely autonomous. Changes in PHYs or PHY link state are communicated back to the EMAC module as a return value from the MDIO event processor. The EMAC module can then retrieve the current MDIO state by calling a status function.

This section is not intended to be a primer on PHYs nor PHY control registers. It is intended to document the operation of the MDIO hardware module. It is assumed you have some knowledge of PHY operation. See your PHY device documentation for more information on PHY control registers.

4.4.1 Initialization

Other than initializing the software state machine (that is beyond the scope of this document), all that needs to be done for the MDIO module is to enable the MDIO engine and to configure the clock divider. To set the clock divider, supply an MDIO clock of 1 MHz. Since the base clock used is the peripheral clock (CPUclk/4), the divider can be set to 150 for a 600 MHz device, with the slower MDIO clocks for slower CPU frequencies being perfectly acceptable.

Both the state machine enable and the MDIO clock divider are both controlled through the MDIO control register (CONTROL). If none of the potentially connected PHYs require the access preamble, the PREAMBLE bit can also be set in CONTROL to speed up PHY register access. The code for this may appear as in Figure 4–2.

Figure 4–2. MDIO Module Initialization Code

```
#define PCLK 150

...

/* Enable MDIO and setup divider */
MDIO_RSET( CONTROL, MDIO_FMKS(CONTROL,ENABLE,YES) |
                    MDIO_FMK(CONTROL,CLKDIV,PCLK) );
```

If the MDIO module is to operate on an interrupt basis, the interrupts can be enabled at this time using the USERINTMASKSET register for register access and the USERPHYSEL*n* register if a target PHY is already known.

However, to run the software state machine, a real-time-based timer event is required. For this example, the entire MDIO software engine is powered off a 0.5-second timer. Also, the software autoselects a PHY to use so that the PHY address on the MDIO bus does not have to be specified at run time.

4.4.2 Selecting and Configuring a PHY

Once the MDIO state machine has been enabled, it starts polling all 32 PHY addresses on the MDIO bus, looking for active PHYs. Since this can take up to 50 us to read one register, it can be some time before the MDIO state machine provides an accurate representation of all the PHYs available. Also, a PHY can take up to 3 seconds to negotiate a link. Thus, it is advisable to run the MDIO software off a time-based event rather than polling.

4.4.2.1 PHY Search

The code in Figure 4–3 is run when the software state machine is in its initialization state. It reads the MDIO PHY alive indication register (ALIVE) to get a representation of the PHYs that are currently present on the MDIO bus. Over time, the value of this register can change. Thus, the software must reread the ALIVE register whenever it needs to find a new PHY.

If the corresponding bit is set in the ALIVE register, this code attempts to initialize the PHY based on the input configuration. If the configuration was successful, the PHY search halts while the software state machine waits for a link indication on the target PHY.

Figure 4–3. PHY Search Code

```
// Try the next PHY if anything but a MDIOINIT condition
ltmp1 = MDIO_RGET( ALIVE );
for( tmp1=0; tmp1<32; tmp1++ )
{
    if( ltmp1 & (1<<pd->phyAddr) )
    {
        if( EMIMDIO_initPHY( pd, pd->phyAddr ) )
              break;
    }

    if( ++pd->phyAddr == 32 )
        pd->phyAddr = 0;
}
```

4.4.2.2 Initial PHY Configuration

The code in Figure 4−3 calls a software function named EMIMDIO_initPHY(). This function initializes the PHY and the software state machine. An edited portion of the code is shown in Figure 4−4. The basic process in PHY initial configuration is:

1) Write to the control register all *other* active PHY devices (determined by reading the ALIVE register) to isolate them from the MII bus. Although multiple PHYs can share the MDIO bus, they can not share the MII bus.

2) Write to the control register on the target PHY to reset. Wait and verify that the reset completes. This verifies that the PHY is truly alive.

3) Read the PHY's capabilities from the PHY status register. Select auto-negotiation or fix a PHY configuration based on the PHY's ability and the your preference.

4) Begin waiting for negotiation to complete, or a link condition.

Figure 4–4. PHY Initial Configuration Code

```
Uint16              tmp1,tmp2;
Uint32              ltmp1;
uint                i;

/* Shutdown all other PHYs */
ltmp1 = MDIO_RGET( ALIVE );
for( i=0; ltmp1; i++,ltmp1>>=1 )
{
       if( (ltmp1 & 1) && (i != phyAddr) )
       {
              PHYREG_write( PHYREG_CONTROL, i, PHYREG_CONTROL_ISOLATE |
                                               PHYREG_CONTROL_POWERDOWN );
              PHYREG_wait();
       }
}

/* Reset the PHY we plan to use */
PHYREG_write( PHYREG_CONTROL, phyAddr, PHYREG_CONTROL_RESET );
PHYREG_wait();

/* Wait for reset to go low (but not forever) */
for( i=0; i<5000; i++ )
{
       PHYREG_read( PHYREG_CONTROL, phyAddr );
       PHYREG_waitResults( tmp1 );
       if( !(tmp1 & PHYREG_CONTROL_RESET) )
         break;
}
if( i == 5000 )
       return(0);

/* Read the STATUS reg to check autonegotiation capability */
PHYREG_read( PHYREG_STATUS, phyAddr );
PHYREG_waitResults( tmp1 );

/* See if we auto-neg or not */
if( (pd->ModeFlags & EMIMDIO_MODEFLG_AUTONEG) &&
                                  (tmp1 & PHYREG_STATUS_AUTOCAPABLE) )
{
       /* We will use NWAY */
       /* We then "wait" for negotiation to complete */
}
else
{
       /* We will use a fixed configuration */
       /* We then "wait" for a link indication */
}

return(1);
```

4.4.3 Negotiation Results and Link Indication

Once a PHY has been configured and is either awaiting negotiation or link status, the same state machine checks the status at any given point. The negotiation wait state simply waits for the PHY negotiation to complete. Once this is done, the results of the negotiation are saved and the software state machine enters the link wait state.

The link wait software state just waits for a good link indication from the PHY. This is done by reading the PHY control register. Note that at all times, the MDIO hardware is polling the link state of all PHY devices. The current link state is stored in the MDIO PHY link status register (LINK). The software process for establishing links is:

1) Verify a good link by both reading the PHY status register and by examining the LINK register.

2) Setup to monitor the target PHY using the USERPHYSEL*n* register. This enables tracking of any link state changes using the LINKINTRAW register. Even when polling, it is not possible to miss a link change event.

3) Clear any previously pending LINKINTRAW bit. No race condition, since link would have to go down and comeback up between these two operations. Since it takes thousands of CPU cycles to read the PHY, it can not happen.

4) Begin periodic polling of the LINKINTRAW and LINK registers to look for further link changes. There is no need to access the PHY directly from this point forward.

5) On a timeout, begin using the ALIVE register to select a PHY candidate.

The code in Figure 4–5 performs this operation using USERPHYSEL0.

Figure 4−5. Link Indication Code

```
/* Read the STATUS reg to check for "link" */
PHYREG_read( PHYREG_STATUS, pd->phyAddr );
PHYREG_waitResults( tmp1 );
if( !(tmp1 & PHYREG_STATUS_LINKSTATUS) )
    goto CheckTimeout;

/* Make sure we're linked in the MDIO module as well */
ltmp1 = MDIO_RGET( LINK );
if( !(ltmp1&(1<<pd->phyAddr)) )
    goto CheckTimeout;

/* Start monitoring this PHY */
MDIO_RSET( USERPHYSEL0, pd->phyAddr );

/* Clear the link change flag so we can detect a "re-link" later */
MDIO_RSET( LINKINTRAW, 1 );
```

4.4.4 Monitoring (Event Processing)

The MDIO software module from which the code examples are taken is written such that a central event function handles all parts of the PHY operation. This event function is called every half second. When in the “linked” software state, the only operation to be performed is to check the status of the LINKINTRAW register for link status changes. When the LINKINTRAW register indicates a change of status or the LINK register indicates no current link, the following operations are performed:

1) If using autonegotiation and the link is currently down, then restart negotiation; otherwise, reread negotiation results.

2) Wait for negotiation results when appropriate, or just wait for link.

3) (Execute the same code as in section 4.4.3).

The code in Figure 4−6 performs this operation. Most of the actions taken on a link change event is executed by code from section 4.4.3.

Figure 4−6. Link Status Monitoring Code

```
/*
// Here we check for a "link-change" status indication or a link
// down indication.
*/
ltmp1 = MDIO_RGET( LINKINTRAW ) & 1;
MDIO_RSET( LINKINTRAW, ltmp1 );
if( ltmp1 || !(MDIO_RGET(LINK)&(1<<pd->phyAddr)) )
{
       /*
       // There has been a change in link (or it is down)
       // If we do not auto-neg, then we just wait for a new link
       // Otherwise, we enter NWAYSTART or NWAYWAIT
       */

       /* If not NWAY, just wait for link */
       if( !(pd->ModeFlags & EMIMDIO_MODEFLG_NWAYACTIVE) )
              pd->phyState = PHYSTATE_LINKWAIT;
       else
       {
              /* Handle NWAY condition */

              /* First see if link is really down */
              PHYREG_read( PHYREG_STATUS, pd->phyAddr );
              PHYREG_wait();
              PHYREG_read( PHYREG_STATUS, pd->phyAddr );
              PHYREG_waitResults( tmp1 );
              if( !(tmp1 & PHYREG_STATUS_LINKSTATUS) )
              {
                     /* No Link - restart NWAY */
                     pd->phyState = PHYSTATE_NWAYSTART;

                     PHYREG_write( PHYREG_CONTROL, pd->phyAddr,
                                   PHYREG_CONTROL_AUTONEGEN |
                                   PHYREG_CONTROL_AUTORESTART );
                     PHYREG_wait();
              }
              else
              {
              /* We have a Link - re-read NWAY params  */
              pd->phyState = PHYSTATE_NWAYWAIT;
              }
       }
}
```

4.4.5 MDIO Register Access

All of the routines previously described use the MDIO module to access PHY control registers. This is done by using the USERACCESS*n* register. This register access process is described in sections 3.3.2 and 3.3.3. The software functions that implement the access process are four macros:

- PHYREG_read(regadr, phyadr) – start the process of reading a PHY register
- PHYREG_write(regadr, phyadr, data) – start the process of writing a PHY register
- PHYREG_wait() – synchronize operation (make sure read/write is idle)
- PHYREG_waitResults(results) – wait for read to complete and return data read

Note that it is not necessary for a wait after a write operation, as long as the status is checked before every operation to make sure the MDIO hardware is idle. An alternative approach is to call PHYREG_wait() after every write, and PHYREG_waitResults() after every read, then the hardware can be assumed to be idle when starting a new operation.

The macros are defined in Figure 4–7 (USERACCESS0 is assumed).

Note that the ACK bit is not checked on PHY register reads (does not follow the procedure outlined in section 3.3.3). Since the ALIVE register is used to initially select a PHY, it is assumed that the PHY is acknowledging read operations. It is possible that a PHY could become inactive at a future point in time. An example of this would be a PHY than can have its MDIO addresses changed while the system is running. Not very likely, but this condition can be tested by periodically checking the PHY state in the ALIVE register.

Figure 4−7. MDIO Register Access Macros

```
#define PHYREG_read(regadr, phyadr)                                          \
             MDIO_RSET( USERACCESS0,                                         \
                    MDIO_FMK(USERACCESS0,GO,1u)             |                \
                    MDIO_FMK(USERACCESS0,REGADR,regadr)     |                \
                    MDIO_FMK(USERACCESS0,PHYADR,phyadr)     )

#define PHYREG_write(regadr, phyadr, data)                                   \
             MDIO_RSET( USERACCESS0,                                         \
                    MDIO_FMK(USERACCESS0,GO,1u)             |                \
                    MDIO_FMK(USERACCESS0,WRITE,1)           |                \
                    MDIO_FMK(USERACCESS0,REGADR,regadr)     |                \
                    MDIO_FMK(USERACCESS0,PHYADR,phyadr)     |                \
                    MDIO_FMK(USERACCESS0,DATA, data)        )

#define PHYREG_wait()                                                        \
             while( MDIO_FGET(USERACCESS0,GO) )

#define PHYREG_waitResults( results )                                        \
             while( MDIO_FGET(USERACCESS0,GO) );                             \
             results = MDIO_FGET(USERACCESS0,DATA)
```

4.5 EMAC Module Operation

The EMAC module is used to send and receive data packets over the network. Most of the work in developing an application or device driver for Ethernet is programming this module. The software described is written to implement a basic Ethernet driver. The code is straight forward and non-reentrant. It is assumed that all reentrancy exclusion methods are handled external to this module.

4.5.1 Initialization

The following the initialization procedure to get the EMAC to the state where it is ready to receive and send Ethernet packets. Some of these steps are not necessary when performed immediately after device reset.

1) If enabled, clear the device interrupt enable in EWCTL.
2) Clear the MACCONTROL, RXCONTROL, and TXCONTROL registers (*not necessary immediately after reset*).
3) Initialize all 16 HDP registers (RX*n*HDP and TX*n*HDP) to 0.
4) Clear all 36 statistics registers by writing 0 (*not necessary immediately after reset*).

5) Setup the local Ethernet MAC address by programming the MACADDRL*n*, MACADDRM, and MACADDRH registers. Be sure to program all eight MAC addresses – whether the receive channel is to be enabled or not. Duplicate the same MAC address across all unused channels. When using more than one receive channel, start with channel 0 and progress upwards. Write all MACADDRL*n* values first, then MACADDRM and MACADDRH.

6) Initialize the RX*n*FREEBUFFER, RX*n*FLOWTHRESH, and RXFILTERLOWTHRESH registers, if flow control is to be enabled (not used here).

7) Most device drivers open with no multicast addresses, so clear MACHASH1 and MACHASH2 registers to 0.

8) Write the RXBUFFEROFFSET register value (typically zero).

9) Initially clear all unicast channels by writing FFh to the RXUNICASTCLEAR register. If unicast is desired, it can be enabled now by writing the RXUNICASTSET register. Some drivers will default to unicast on device open while others will not.

10) Setup the RXMBPENABLE register with an initial configuration. The configuration is based on the current receive filter settings of the device driver. Some drivers may enable things like broadcast and multicast packets immediately, while others may not.

11) Set the appropriate configuration bits in the MACCONTROL register (do not set the MIIEN bit).

12) Clear all unused channel interrupt bits by writing RXINMASKCLEAR and TXINTMASKCLEAR.

13) Enable the receive and transmit channel interrupt bits in RXINTMASKSET and TXINTMASKSET for the channels to be used, and enable the HOSTERRINT and STATINT bits using the MACINTMASKSET register.

14) Initialize the receive and transmit descriptor list queues. There is an infinite number of way this can be done using the 4K descriptor memory block contained in the EMAC control module. One particular method is detailed later in this chapter.

15) Prepare receive by writing a pointer to the head of the receive buffer descriptor list to RX*n*HDP. In this example we use only RX0HDP.

16) Enable the receive and transmit DMA controllers by setting the RXEN bit in the RXCONTROL register and the TXEN bit in the TXCONTROL register. Then set the MIIEN bit in MACCONTROL.

17) Enable the device interrupt in EWCTL.

The code in Figure 4−8 implements the initialization steps. Some simplifications have been made, but the full source code to the Ethernet module is available in the Chip Support Library (CSL).

Figure 4−8. EMAC Module Initialization Code

```
/*
// Disable receive, transmit, and clear MACCONTROL
// This is not really necessary if we assume EMAC was just reset
*/
EMAC_FSETS( TXCONTROL, TXEN, DISABLE );
EMAC_FSETS( RXCONTROL, RXEN, DISABLE );
EMAC_RSET( MACCONTROL, 0 );

/* Must manually init HDPs to NULL */
pRegAddr = EMAC_ADDR(TX0HDP);
for( i=0; i<8; i++ )
    *pRegAddr++ = 0;
pRegAddr = EMAC_ADDR(RX0HDP);
for( i=0; i<8; i++ )
    *pRegAddr++ = 0;

/*
// While MIIEN is clear in MACCONTROL, we can write directly to
// the statistics registers (there are 36 of them).
*/
pRegAddr = EMAC_ADDR(RXGOODFRAMES);
for( i=0; i<36; i++ )
    *pRegAddr++ = 0;

/* Setup device MAC address */
pRegAddr = EMAC_ADDR(MACADDRL0);
for( i=0; i<8; i++ )
    *pRegAddr++ = localDev.Config.MacAddr[5];
EMAC_RSET( MACADDRM, localDev.Config.MacAddr[4] );
tmpval = 0;
for( i=3; i>=0; i-- )
    tmpval = (tmpval<<8) | localDev.Config.MacAddr[i];
EMAC_RSET( MACADDRH, tmpval );

/* Clear multicast hash bits */
EMAC_RSET( MACHASH1, 0 );
EMAC_RSET( MACHASH2, 0 );

/* For us buffer offset will always be zero */
EMAC_RSET( RXBUFFEROFFSET, 0 );

/* Clear Unicast receive on channel 0-7 */
EMAC_RSET( RXUNICASTCLEAR, 0xFF );

/* Reset receive (M)ulticast (B)roadcast (P)romiscuous Enable register */
EMAC_RSET( RXMBPENABLE, 0 );
```

Figure 4−8. EMAC Module Initialization Code (Continued)

```
/* Set the pass receive CRC mode and adjust max buffer accordingly */
if( localDev.Config.ModeFlags & EMI_CONFIG_MODEFLG_RXCRC )
{
    EMAC_FSETS( RXMBPENABLE, RXPASSCRC, INCLUDE );
    localDev.PktMTU = 1518;
}
else
    localDev.PktMTU = 1514;

/* Set the channel configuration to priority if requested */
if( localDev.Config.ModeFlags & EMI_CONFIG_MODEFLG_CHPRIORITY )
    EMAC_FSETS( MACCONTROL, TXPTYPE, CHANNELPRI );

/*
// Enable transmit and receive channel interrupts (set mask bits)
// We only ever use on receive channel, but up to 8 transmit channels
// Enable Host interrupts
*/
EMAC_RSET( RXINTMASKCLEAR, 0xFF );
EMAC_RSET( TXINTMASKCLEAR, 0xFF );
EMAC_RSET( RXINTMASKSET, 1 );
for(i=0; i<localDev.Config.TxChannels; i++)
EMAC_RSET( TXINTMASKSET, (1<<i) );
EMAC_RSET( MACINTMASKSET, EMAC_FMK(MACINTMASKSET,HOSTERRINT,1) |
                          EMAC_FMK(MACINTMASKSET,STATINT,1) );

/*
// Setup Receive Buffers and Transmit Buffers
*/
```

[Discussed Later in this document]

```
/* Prepare receive */
EMAC_RSET( RX0HDP, (Uint32)localDev.RxCh.pDescRead );

/*
// Enable receive, transmit, and MII
*/
EMAC_FSETS( TXCONTROL, TXEN, ENABLE );
EMAC_FSETS( RXCONTROL, RXEN, ENABLE );
EMAC_FSETS( MACCONTROL, MIIEN, ENABLE );

/* Enable global interrupt in control module */
EMAC_FSETS( EWCTL, INTEN, ENABLE );
```

4.5.2 Configuration

The example code given in the previous section assumes that the EMAC is being initialized in a (mostly) idle state, and that it can not receive any type of Ethernet packet (unicast, broadcast, or multicast) in its default state. The software interface from which the example code is taken provides two functions to configure packet reception, setReceiveFilter() and setMulticast().

4.5.2.1 Setting the Receive Filter

There are two approaches to a receive filter in an Ethernet device driver. One approach is to treat unicast, broadcast, and multicast packets as all individual entities. The second approach is to treat each receive level as being inclusive of the previous level. This example takes the second approach.

Regardless of the software approach, to control unicast, broadcast, multicast, and promiscuous operations, the RXUNICASTSET, RXUNICASTCLEAR, RXMBPENABLE, and MACHASH*n* registers are used.

This code example assumes a filer value set as follows. Each successive filter is includes the previous, so the effect is cumulative:

```
#define EMI_RXFILTER_NOTHING            0  /* Receive nothing */
#define EMI_RXFILTER_DIRECT             1  /* Receive unicast packets */
#define EMI_RXFILTER_BROADCAST          2  /* Above plus broadcast packets */
#define EMI_RXFILTER_MULTICAST          3  /* Above plus specified multicast */
#define EMI_RXFILTER_ALLMULTICAST       4  /* Above plus all multicast */
#define EMI_RXFILTER_ALL                5  /* Any non-error packet */
```

The code to set the filter setting (stored in the variable `ReceiveFilter`) is shown in Figure 4–9. The logic is to disable anything that is not set, and then enable anything that is set. When receiving a specified list of multicast addresses, the bits representing the specified list are stored in `pd->MacHash1` and `pd->MacHash2`. The code to calculate these values is discussed in the next section.

Figure 4−9. Setting the Receive Filter Code

```
/*
// The following code relies on the numeric relation of the filter
// value such that the higher filter values receive more types of
// packets.
*/

/* Disable Section */
if( ReceiveFilter < EMI_RXFILTER_ALL )
    EMAC_FSETS( RXMBPENABLE, RXCAFEN, DISABLE );
if( ReceiveFilter < EMI_RXFILTER_ALLMULTICAST )
{
    EMAC_RSET( MACHASH1, pd->MacHash1 );
    EMAC_RSET( MACHASH2, pd->MacHash2 );
}
if( ReceiveFilter < EMI_RXFILTER_MULTICAST )
    EMAC_FSETS( RXMBPENABLE, MULTEN, DISABLE );
if( ReceiveFilter < EMI_RXFILTER_BROADCAST )
    EMAC_FSETS( RXMBPENABLE, BROADEN, DISABLE );
if( ReceiveFilter < EMI_RXFILTER_DIRECT )
    EMAC_RSET( RXUNICASTCLEAR, 1 );

/* Enable Section */
if( ReceiveFilter >= EMI_RXFILTER_DIRECT )
    EMAC_RSET( RXUNICASTSET, 1 );
if( ReceiveFilter >= EMI_RXFILTER_BROADCAST )
    EMAC_FSETS( RXMBPENABLE, BROADEN, ENABLE );
if( ReceiveFilter >= EMI_RXFILTER_MULTICAST )
    EMAC_FSETS( RXMBPENABLE, MULTEN, ENABLE );
if( ReceiveFilter >= EMI_RXFILTER_ALLMULTICAST )
{
    EMAC_RSET( MACHASH1, 0xffffffff );
    EMAC_RSET( MACHASH1, 0xffffffff );
}
if( ReceiveFilter == EMI_RXFILTER_ALL )
    EMAC_FSETS( RXMBPENABLE, RXCAFEN, ENABLE );

pd->RxFilter = ReceiveFilter;
```

4.5.2.2 Setting the Multicast List

Sometimes in a device driver, adding and removing addresses from a multicast list can be a single entry at a time. In other device drivers or mini-drivers, the multicast list is maintained by a parent driver or the application and always passed down as a list as in this example.

The code in Figure 4−10 has a very specific function. It take a list of Ethernet MAC addresses and for each address hashes it to calculate a bit to set in the MACHASH*n* register to allow the EMAC to receive packets destined for that address. The accumulated set of bits to set in MACHASH0 and MACHASH1 are stored in the variables `pd->MacHash1` and `pd->MacHash2` for use in the setReceiveFilter() function.

In Figure 4−10, `AddrCnt` is the number of 6-byte MAC addresses in the address list, and `pMCastList` is a pointer to a Uint8, pointing to a concatenated list of MAC addresses (each being 6 bytes in length).

Figure 4−10. Setting the Multicast List Code

```
Uint8 HashVal,tmpval;

/* Clear the hash bits */
pd->MacHash1 = 0;
pd->MacHash2 = 0;

/* For each address in the list, hash and set the bit */
for( tmp1=0; tmp1<AddrCnt; tmp1++ )
{
    HashVal=0;

    for( tmp2=0; tmp2<2; tmp2++ )
    {
        tmpval = *pMCastList++;
        HashVal ^= (tmpval>>2)^(tmpval<<4);
        tmpval = *pMCastList++;
        HashVal ^= (tmpval>>4)^(tmpval<<2);
        tmpval = *pMCastList++;
        HashVal ^= (tmpval>>6)^(tmpval);
    }

    if( HashVal & 0x20 )
        pd->MacHash2 |= (1<<(HashVal&0x1f));
    else
        pd->MacHash1 |= (1<<(HashVal&0x1f));
}

/* We only write the hash table if the filter setting allows */
if( pd->RxFilter < EMI_RXFILTER_ALLMULTICAST )
{
    EMAC_RSET( MACHASH1, pd->MacHash1 );
    EMAC_RSET( MACHASH2, pd->MacHash2 );
}
```

4.5.3 Receive

The reception of Ethernet packets is performed through the use of a buffer descriptor system where the application software or device driver describes empty memory buffers to the EMAC to which Ethernet packet data can be written. The buffer descriptor is a 16-byte memory structure that is stored in a 4K-byte memory space contained in the EMAC control module. The EMAC control module has space for up to 256 descriptors. You should be familiar with the EMAC operational overview in section 2.3 and the detailed description of the receive buffer descriptor fields in section 2.3.5.

There are a number of ways in which the descriptor memory contained in the EMAC control module can be managed. One option would be to write a memory allocation system where 16-byte descriptors are allocated and freed as needed, so that a descriptor may be used for an receive buffer at one point, and then a totally different transmit buffer the next. Another option would be to statically allocate packet buffers and permanently assign a descriptor slot to each. This way, the descriptor's pointer to the packet buffer would never have to be updated.

The method used in this example code uses a third option. Here, the 256 descriptor slots available in the control module are divided in an arbitrary method where each receive or transmit channel has its own set of descriptors. The descriptor structure is:

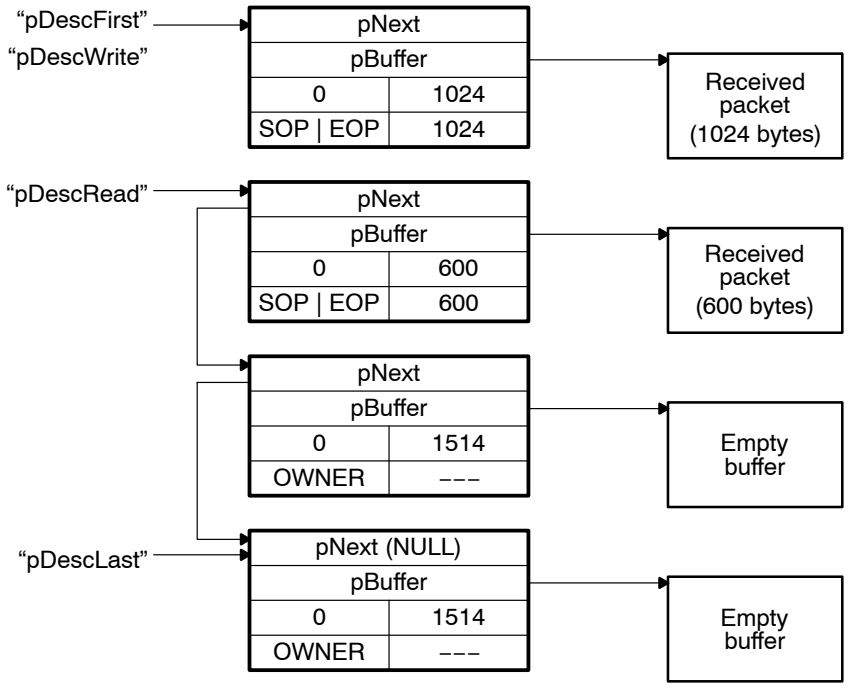
```
/*
// Transmit/Receive Descriptor Channel Structure
*/
typedef struct _EMI_DescCh {
    struct _EMI_Device *pd;       /* Pointer to parent structure   */
    PKTQ            DescQueue;    /* Packets queued as desc        */
    PKTQ            WaitQueue;    /* Packets waiting for transmit desc   */
    uint            ChannelIndex; /* Channel index 0-7             */
    uint            DescMax;      /* Max number of desc (buffers)  */
    uint            DescCount;    /* Current number of desc        */
    EMAC_Desc       *pDescFirst;  /* First desc location           */
    EMAC_Desc       *pDescLast;   /* Last desc location            */
    EMAC_Desc       *pDescRead;   /* Location to read next desc    */
    EMAC_Desc       *pDescWrite;  /* Location to write nest desc   */
} EMI_DescCh;
```

For a receive channel, each descriptor refers to a fixed length buffer that is always at least 1514 or 1518 bytes in length (depending on whether CRC is included in the data or not). Thus each descriptor represents one packet. There is a fixed number of descriptors and that number represents the maximum number of packets that can be received before the receive interrupt needs to be serviced.

Figure 4−11 illustrates some of the descriptor fields. Each receive channel has a fixed number of slots. When a packet is received and handed over to the software for processing, a fresh empty buffer is pulled from a central pool, and the descriptor slot is reused to point to the new buffer. The `DescQueue` field in the structure is a queue of physical packet buffers (in DSP memory) that are current being “described” by the descriptor list. Since there is no queue of free buffers for any given receive channel (other than those already contained in the descriptor list), the `WaitQueue` is not used.

The descriptors are tracked via the variables, `pDestFirst`, `pDestLast`, `pDescRead`, and `pDescWrite`. The `pDescFirst` and `pDescLast` pointers just point to the first and last descriptors in the fixed circular queue. These never change. The `pDescRead` pointer points to the next descriptor buffer that may contain a new packet received from the network. The `pDescWrite` pointer points to the descriptor to use when adding the next empty buffer to the queue.

Figure 4−11. Receive Descriptor Linked List



In Figure 4−11, there are four descriptors allocated to the receive channel. Of these, only three descriptors are in use. One of the descriptors has already received a packet that has been handed up to the software. This descriptor is current not in use. The next descriptor has received a packet, but has not been serviced yet by the software. The final two descriptors point to empty data buffers and are waiting to receive packet data from the EMAC.

In practice, the software always tries to keep all descriptors pointing to empty buffers. This allows the EMAC to run longer without being serviced and without experiencing a packet overrun condition.

For servicing a receive channel, there are two basic functions. The first function is called EnqueueRx(). Its job is to fill all possible receive descriptor slots so that they point to empty packet buffers. The second function is called DequeueRx(). Its job is to pull buffers from the list that have received packet data, and to update the corresponding descriptor so that is points to a new empty buffer.

It is helpful to consider the how packet buffers are represented in the code. The example code in Figure 4−12 uses a structure of type `EMI_Pkt` to define a packet. This structure has little to do with the EMAC hardware, but must be understood to follow the software examples. The structure and its related flags are defined below. Note that it is significantly similar to the descriptor format.

Figure 4−12. Receive Packets Example Code

```
typedef struct _EMI_Pkt {
    struct _EMI_Pkt *pPrev;       /* Previous record                   */
    struct _EMI_Pkt *pNext;       /* Next record                       */
    Uint8           *pDataBuffer; /* Pointer to Data Buffer            */
    Uint32          BufferLen;    /* Phys Length of buffer (read only) */
    Uint32          Flags;        /* Packet Flags                      */
    Uint32          ValidLen;     /* Length of valid data in buffer    */
    Uint32          DataOffset;   /* Byte offset to valid data         */
    Uint32          PktChannel;   /* Transmit Channel/Priority 0-7 (SOP only)*/
    Uint32          PktLength;    /* Length of Packet (SOP only)       */
    Uint32          PktFrags;     /* Num frags in packet (SOP only)    */
    } EMI_Pkt;

/*
// Packet Buffer Flags set in Flags
*/
#define EMI_PKT_FLAGS_SOP         0x80000000u /* Start of packet         */
#define EMI_PKT_FLAGS_EOP         0x40000000u /* End of packet           */

/*
// The Following Packet flags are set in Flags on receive packets only
*/
#define EMI_PKT_FLAGS_HASCRC      0x04000000u /* RxCrc: PKT has 4byte CRC*/
#define EMI_PKT_FLAGS_JABBER      0x02000000u /* RxErr: Jabber           */
#define EMI_PKT_FLAGS_OVERSIZE    0x01000000u /* RxErr: Oversize         */
#define EMI_PKT_FLAGS_FRAGMENT    0x00800000u /* RxErr: Fragment         */
#define EMI_PKT_FLAGS_UNDERSIZED  0x00400000u /* RxErr: Undersized       */
#define EMI_PKT_FLAGS_CONTROL     0x00200000u /* RxCtl: Control Frame    */
#define EMI_PKT_FLAGS_OVERRUN     0x00100000u /* RxErr: Overrun          */
#define EMI_PKT_FLAGS_CODEERROR   0x00080000u /* RxErr: Code Error       */
#define EMI_PKT_FLAGS_ALIGNERROR  0x00040000u /* RxErr: Alignment Error  */
#define EMI_PKT_FLAGS_CRCERROR    0x00020000u /* RxErr: Bad CRC          */
#define EMI_PKT_FLAGS_NOMATCH     0x00010000u /* RxPrm: No Match         */
```

4.5.3.1 Enqueue Receive Descriptor Function

In an ideal system, the only call to an EnqueueRx() function would occur during initialization. This is because part of the DequeueRx() function is to keep the descriptor list full of pointers to empty buffers. However, at any given time, an empty buffer may not be available, so one or more descriptor slots allocated to an receive channel can become empty. This was shown in Figure 4−11 that had one empty descriptor.

To fully understand the enqueue function, Figure 4−13 shows the code from the initialization function that allocates descriptor slots to the one receive channel and multiple transmit channels in the driver environment. Also, the calls for EnqueueRx() to fill the descriptors with pointers to empty buffers is shown.

In this code, the variable `localDev.RxCh` is a structure of type `EMI_DescCh` described earlier.

Figure 4−13. Initialization Code That Allocates Descriptor Slots

```
/*
// Setup Receive Buffers
*/

/*
// We give the first descriptors to receive The rest of the descriptors
// will be divided evenly among the transmit channels. Odds are this
// will leave transmit with a very large number of transmit descriptors, but
// we'll only use what we need (driven from the application send
// requests). The receive descriptors are always kept fully populated.
*/

/* Pointer to first descriptor to use on receive */
pDesc = (EMAC_Desc *)_EMAC_DSC_BASE_ADDR;

/* Number of descriptors for receive channel */
utemp1 = localDev.Config.RxMaxPktPool;

/* Init receive */
localDev.RxCh.pd         = &localDev;
localDev.RxCh.DescMax    = utemp1;
localDev.RxCh.pDescFirst = pDesc;
localDev.RxCh.pDescLast  = pDesc + (utemp1 - 1);
localDev.RxCh.pDescRead  = pDesc;
localDev.RxCh.pDescWrite = pDesc;

/* Fill the descriptor table */
EnqueueRx( &localDev.RxCh, 0 );
```

The second calling parameter to EnqueueRx, is a flag indicating that the function is being called at initialization time, and it should not restart the receiver. The only other time the function can be called if from a half second polling loop (the same that drives the MDIO software state machine). This is done so that if a buffer shortfall occurs, the system looks for new buffers every half second.

Now here is the enqueue function. The process for enqueuing a packet buffer to the descriptor ring is:

1) If the descriptor set is not full, call an application callback to get a free packet buffer.

2) If the packet buffer was obtained, get a pointer to the descriptor to fill from `pDescWrite` and advance the `pDescWrite` pointer while bumping the `DescCount`.

3) Fill in the descriptor with the pointer to the packet buffer. The size is fixed at max packet size (1514 or 1518). Also set the OWNER flag in the descriptor so that the EMAC knows it can use it.

4) Make the `pNext` pointer for the new descriptor NULL because it is always the end of the list. Make the previous descriptor in the set point to the new descriptor.

5) Push a structure pointer (handle) to the packet buffer (the thing the descriptor points to) onto its own software queue. The software queue of packet buffer handles is kept synchronized with the list of buffer descriptors. Thus the packet buffer handle can be given back to the application once a packet has been received into the buffer.

6) Return to step 1 until full or no more free buffers.

7) As a final step (if not called during initialization); if when the function was called, all the receive descriptors were used, then the receive engine must be stopped. If new descriptors have been added, then restart the receive engine by posting the head of the descriptor list (`pDescRead`) to RX0HDP.

The source code to implement this function is shown in Figure 4–14.

Figure 4−14. Enqueue Receive Descriptor Function Code

```
static void EnqueueRx( EMI_DescCh *pdc, uint fRestart )
{
    EMI_Pkt     *pPkt;
    EMAC_Desc   *pDesc;
    uint        CountOrg;

    /* Keep the old count around */
    CountOrg = pdc->DescCount;

    /* Fill receive Packets Until Full */
    while( pdc->DescCount < pdc->DescMax )
    {
        /* Get a buffer from the application */
        pPkt = (*localDev.Config.pfcbGetPacket)(pdc->pd->hApplication);

        /* If no more buffers are available, break out of loop */
        if( !pPkt ) break;

        /* Fill in the descriptor for this buffer */
        pDesc = pdc->pDescWrite;

        /* Move the write pointer and bump count */
        if( pdc->pDescWrite == pdc->pDescLast )
            pdc->pDescWrite = pdc->pDescFirst;
        else
            pdc->pDescWrite++;
        pdc->DescCount++;

        /* Supply buffer pointer with application supplied offset */
        pDesc->pNext     = 0;
        pDesc->pBuffer   = pPkt->pDataBuffer + pPkt->DataOffset;
        pDesc->BufOffLen = localDev.PktMTU;
        pDesc->PktFlgLen = EMAC_DSC_FLAG_OWNER;

        /* Make the previous buffer point to us */
        if( pDesc == pdc->pDescFirst )
             pdc->pDescLast->pNext = pDesc;
        else
            (pDesc-1)->pNext = pDesc;

        /* Push the packet buffer on the local descriptor queue */
        pqPush( &pdc->DescQueue, pPkt );
    }

    /* Restart receive if we had ran out of descriptors and got some here */
    if( fRestart && !CountOrg && pdc->DescCount )
        EMAC_RSET( RX0HDP, (Uint32)pdc->pDescRead );
}
```

4.5.3.2 Dequeue Receive Descriptor Function

The DequeueRx() function is the more interesting of the two receive descriptor-based functions. This function is to process new packets as they are received by the EMAC, and keep the receive descriptor set always pointing to fresh empty packet buffers.

To understand this function better, it is important to know what happens during a device interrupt. The ISR code relating to the receive operation is:

```
/* Look for receive interrupt (channel 0) */
if( intflags & EMAC_FMK( MACINVECTOR, RXPEND, 1<<0 ) )
{
       Desc = EMAC_RGET( RX0INTACK );
       EMAC_RSET( RX0INTACK, Desc );
       DequeueRx( &pd->RxCh, (EMAC_Desc *)Desc );
}
```

First, the RXPEND register is examined to determine which receive channels have had new activity. In this code, only receive channel 0 is used. Next the last descriptor to process can be read from RX0INTACK. This is also the register we write the value of the last descriptor processed. Since the DequeueRx() function processes all descriptors up to the one that it is passed, the receive interrupt can be immediately acknowledged by writing the value back to the RX0INTACK register. Finally the DequeueRx() function is called with a pointer to the receive descriptor channel structure and a pointer to the last descriptor to service.

The DequeueRx() function is shown in Figure 4–15. The functions it needs to perform are:

1) The next descriptor to process is always available at the `pDescRead` pointer. The flags for that descriptor are read. Also, the packet buffer that corresponds to the descriptor is popped of the software queue.

2) The `EMI_Pkt` structure fields are filled in based of the information in the buffer descriptor. If the driver is configured to receive error packets, then the error bits are potentially set in the flags field as well.

3) A pointer to the completed `EMI_Pkt` structure is passed to the application via a callback function. This function should return a pointer to an identical structure containing a new empty buffer.

4) If this is the last descriptor to process (`pDescRead == pDescAck`), then a flag is set to prevent the loop from executing again.

5) The `pDescRead` pointer is incremented and the `DescCount` is decremented.

6) If the application did supply a new empty buffer, the buffer is added to the next available descriptor as read from the `pDescWrite` pointer. Under ideal circumstances, this will be the same descriptor that just contained the received packet. However if there is a free buffer shortage, the read and write pointers will not be synchronized.

7) Next the descriptor is initialized to point to the empty packet buffer. This code is very similar to that described in section 4.5.3.1.

8) Continue until all the descriptors have been processed up to and including that indicated by the caller (in this case the ISR).

9) As a final step, if the last descriptor processed had the EMAC_DSC_FLAG_EOQ flag set in its flags field, this means that the EMAC interpreted the descriptor as being the last in the descriptor chain (its next pointer was NULL). This should not happen under normal operation, but can occur if the system runs out of receive buffer. Since the receive engine stops on this descriptor, it can only happen on the last descriptor to process. When the bit is set, and there are some free buffer descriptors ready, then restart the receive engine by posting the head of the descriptor list (`pDescRead`) RX0HDP.

The source code to implement this function is in Figure 4−15.

Figure 4−15. Dequeue Receive Descriptor Function Code

```
static void DequeueRx( EMI_DescCh *pdc, EMAC_Desc *pDescAck )
{
    EMI_Pkt     *pPkt;
    EMI_Pkt     *pPktNew;
    EMAC_Desc   *pDesc;
    uint        tmp;
    Uint32      PktFlgLen;

    /* Pop & Free Buffers 'till the last Descriptor */
    for( tmp=1; tmp; )
    {
        /* Get the status of this descriptor */
        PktFlgLen = pdc->pDescRead->PktFlgLen;

        /* Recover the buffer and free it */
        pPkt = pqPop( &pdc->DescQueue );
        if( pPkt )
        {
            /* Fill in the necessary packet header fields */
            pPkt->Flags = PktFlgLen & 0xFFFF0000;
            pPkt->ValidLen = pPkt->PktLength = PktFlgLen & 0xFFFF;
             pPkt->PktChannel = 0;
             pPkt->PktFrags = 1;
```

Figure 4−15. Dequeue Receive Descriptor Function Code (Continued)

```
            /* Pass the packet to the application */
            pPktNew = (*localDev.Config.pfcbRxPacket)
                                        (pdc->pd->hApplication,pPkt);
        }

        /* See if this was the last buffer */
        if( pdc->pDescRead == pDescAck )
            tmp = 0;

        /* Move the read pointer and decrement count */
        if( pdc->pDescRead == pdc->pDescLast )
            pdc->pDescRead = pdc->pDescFirst;
        else
            pdc->pDescRead++;
        pdc->DescCount--;

        /* See if we got a replacement packet */
        if( pPktNew )
        {
            /* We know we can immediately queue this packet */

            /* Fill in the descriptor for this buffer */
            pDesc = pdc->pDescWrite;

            /* Move the write pointer and bump count */
            if( pdc->pDescWrite == pdc->pDescLast )
                pdc->pDescWrite = pdc->pDescFirst;
            else
                pdc->pDescWrite++;
            pdc->DescCount++;

            /* Supply buffer pointer with application supplied offset */
            pDesc->pNext     = 0;
            pDesc->pBuffer   = pPktNew->pDataBuffer + pPktNew->DataOffset;
            pDesc->BufOffLen = localDev.PktMTU;
            pDesc->PktFlgLen = EMAC_DSC_FLAG_OWNER;

            /* Make the previous buffer point to us */
            if( pDesc == pdc->pDescFirst )
                 pdc->pDescLast->pNext = pDesc;
            else
                (pDesc-1)->pNext = pDesc;

            /* Push the packet buffer on the local descriptor queue */
            pqPush( &pdc->DescQueue, pPktNew );
        }
    }
    /* If the receiver stopped and we have more descriptors, then restart */
    if( (PktFlgLen & EMAC_DSC_FLAG_EOQ) && pdc->DescCount )
        EMAC_RSET( RX0HDP, (Uint32)pdc->pDescRead );
}
```

4.5.4 Transmit

The transmission of Ethernet packets is performed through the use of a buffer descriptor system where the application software or device driver describes packet to send using one or more memory buffers descriptors. There is one descriptor for each noncontiguous block of memory in the packet (packet fragment). The buffer descriptor is a 16-byte memory structure that is stored in a 4K-byte memory space contained in the EMAC control module. The control module has space for up to 256 descriptors. You should be familiar with the EMAC operational overview in section 2.3 and the detailed description of the transmit buffer descriptor fields in section 2.3.4.

As with the receive operation, there are a number of options for implementing the transmit operation on the EMAC hardware. The example code described here supports up to 8 different transmit channels. Each channel is allocated a static number of buffer descriptor slots from the EMAC control module memory block at initialization. The algorithm chosen for the example code is:

(256 less those required by receive) / number of transmit channels

Note that since a packet must fit entirely in the descriptor list in order to be sent, the maximum number of packet fragments that make up a packet can not exceed the total number of buffer descriptors allocated for a particular channel. For example, in a system that uses 64 buffer slots for receive and has eight transmit channels, each transmit channel would be allocated 24 buffer descriptor slots. Thus a single packet in such an environment could not contain more than 24 packet fragments. If only 2 transmit channels were used, each would have 96 buffer descriptors available. In environments where a static descriptor allocation does not yield acceptable results, a dynamic allocation method can be used.

In practice, there are usually more transmit descriptor slots available that are ever needed. However the software should be written to deal with transmit descriptor slot shortfalls. It is not necessary to have the transmit descriptor list as “deep” as receive because additional transmit packets can always be queued in software. Worst case for transmit is that there is a small delay in sending out the next packet, while the worst case for receive is a dropped packet.

Each transmit channel has its own channel descriptor structure. The structure is identical to that used for packet receive:

```
/*
// Transmit/Receive Descriptor Channel Structure
*/
typedef struct _EMI_DescCh {
    struct _EMI_Device *pd;       /* Pointer to parent structure   */
    PKTQ               DescQueue;    /* Packets queued as desc        */
    PKTQ               WaitQueue;    /* Packets waiting for transmit desc   */
    uint               ChannelIndex; /* Channel index 0-7             */
    uint               DescMax;      /* Max number of desc (buffers)  */
    uint               DescCount;    /* Current number of desc        */
    EMAC_Desc          *pDescFirst;  /* First desc location           */
    EMAC_Desc          *pDescLast;   /* Last desc location            */
    EMAC_Desc          *pDescRead;   /* Location to read next desc    */
    EMAC_Desc          *pDescWrite;  /* Location to write nest desc   */
} EMI_DescCh;
```

For a transmit channel, each descriptor refers to a full packet or a partial a packet (packet fragment). For each buffer descriptor, there is a corresponding packet structure. The packet structures are kept in two queues. The `DescQueue` represents packets or packet fragments that are already represented by buffer descriptors in the channel. The `WaitQueue` is a queue of packet structures that are waiting to be placed into buffer descriptors.

The descriptors are tracked using the variables, `pDestFirst`, `pDestLast`, `pDescRead`, and `pDescWrite`. The `pDescFirst` and `pDescLast` pointers just point to the first and last descriptors in the fixed circular queue; these never change. The `pDescRead` pointer points to the next descriptor whose packet buffer is the next to be sent out on the network. The `pDescWrite` pointer points to the descriptor to use when adding the next packet to be transmitted.

It is helpful to consider the how packet buffers are represented in the code. The example code in Figure 4–16 uses a structure of type `EMI_Pkt` to define a packet. This structure has little to do with the EMAC hardware, but must be understood to follow the software examples. The structure and its related flags are defined below. Note that it is significantly similar to the descriptor format.

Figure 4−16. Transmit Packets Example Code

```
typedef struct _EMI_Pkt {
    struct _EMI_Pkt *pPrev;       /* Previous record                  */
    struct _EMI_Pkt *pNext;       /* Next record                      */
    Uint8           *pDataBuffer; /* Pointer to Data Buffer           */
    Uint32          BufferLen;    /* Phys Length of buffer (read only) */
    Uint32          Flags;        /* Packet Flags                     */
    Uint32          ValidLen;     /* Length of valid data in buffer   */
    Uint32          DataOffset;   /* Byte offset to valid data        */
    Uint32          PktChannel;   /* transmit Channel/Priority 0-7 (SOP only)*/
    Uint32          PktLength;    /* Length of Packet (SOP only)      */
    Uint32          PktFrags;     /* Num frags in packet (SOP only)   */
    } EMI_Pkt;

/*
// Packet Buffer Flags set in Flags
*/
#define EMI_PKT_FLAGS_SOP         0x80000000u /* Start of packet        */
#define EMI_PKT_FLAGS_EOP         0x40000000u /* End of packet          */
```

4.5.4.1 Send Function

Since the packet send process starts with the send function, we need to understand how the send function works in order to understand the rest. In some applications or drivers, it may not be necessary to support fragmented packets. For example, some TCP/IP stacks will never build a packet for transmission that spans more than one memory buffer. However, since fragmented packets are still somewhat common, the example software we show here does support them.

The code in Figure 4−17 is taken from the packet send function in the example code. Much of the packet validation checking has been removed from this code. For purposes of sending the packet using the EMAC, the following operations are performed in the send function:

1) Make sure the first fragment of the packet has the SOP flag set in its flags member.

2) Count the number of packet fragments by parsing the packet until the EOP flag is found. This also verifies the correctness of the packet buffer chain. Note that only the first packet fragment can have the SOP flag set. This is also checked.

3) Get a pointer (in `pdc`) to the descriptor channel structure corresponding to the transmit channel specified by the caller.

4) Make sure the total number of fragments in the packet does not exceed DescMax; otherwise, the entire packet would never fit in the buffer descriptor list allocated for this channel.

5) Push the packet buffer(s) onto the WaitQueue. This is the queue for packer buffers waiting to be written out to the descriptor chain. At this point we do not know if the packet can be written or not. Even if it can, it must be placed in the queue behind any potential previously pending packets.

6) Call the Enqueuetransmit() function to remove as may packets as possible from the WaitQueue and write them into the buffer descriptor list.

The source code to implement this function is shown in Figure 4–17. The EMI_Pkt structure of the first fragment of the packet to send is pointed to by pPkt.

Figure 4–17. Send Function Code

```
uint        fragcnt;
EMI_Pkt     *pPktLast;
EMI_DescCh  *pdc;

/* Do some packet validation */
if( !(pPkt->Flags & EMI_PKT_FLAGS_SOP) )
    return( EMI_ERROR_BADPACKET );

/* Count the number of frags in this packet */
fragcnt = 1;
pPktLast = pPkt;
while( !(pPktLast->Flags & EMI_PKT_FLAGS_EOP) )
{
    if( !pPktLast->pNext )
        return( EMI_ERROR_INVALID );
    pPktLast = pPktLast->pNext;
    fragcnt++;

    /* At this point we can't have another SOP */
    if( pPktLast->Flags & EMI_PKT_FLAGS_SOP )
        return( EMI_ERROR_INVALID );}

/* Get a local pointer to the descriptor channel */
pdc = &(pd->TxCh[pPkt->PktChannel]);

/* Make sure this packet does not have too many frags to fit */
if( fragcnt > pdc->DescMax )
    return( EMI_ERROR_BADPACKET );

/*
// Queue and packet and service transmitter
*/
pqPushChain( &pdc->WaitQueue, pPkt, pPktLast, fragcnt );
EnqueueTx( pdc );
```

4.5.4.2 Enqueue Transmit Descriptor Function

The EnqueueTX() function is pretty simple mostly because the work of structuring the packet buffers has already been done. The process for enqueuing a packet to the descriptor ring for transmit is:

1) Record the state of the descriptor set (first writable descriptor and the current count). The pointer to the first writable descriptor is saved so that is can be linked to the currently active list (if any) once descriptors for all waiting packets (or packet fragments) have been written. Unlike receive, we can not chain as we go because it is illegal to have a partial packet in the active transmit list at any given time. The save count tells us if the transmitter was running when we first began to add buffer descriptors.

2) Access the `WaitQueue` count to see if there are any packets waiting. We try to read all the packets from the `WaitQueue` and write their buffers into the descriptor list. If at any time, there is not room in the descriptor list for all the fragments of the next waiting packet, we stop.

3) The number of packet fragments is known and part of the packet header. For each buffer in the packet, pop the packer header off `WaitQueue` and the fill in the descriptor list with the pointer to the packet (or packet fragment buffer). The next buffer descriptor to write is found in `pDescWrite`. The value of `pDescWrite` is then incremented.

4) When filling in the descriptor, the OWNER bit is added to all descriptors. Any SOP and EOP bits are also retained. On the SOP packet buffer, the total size of the packet is also written to the buffer descriptor.

5) The packet buffer head is then pushed onto the `DescQueue`. This queue is the holding spot for packet buffers that currently occupy slots in the buffer descriptor list, and the two are always kept synchronized.

6) Once all the packets have been written to descriptors, or when there is no more room in the descriptor list, the process stops. Next, the list must be appended onto any previously existing list, or if there was no list, the new entries written become the active list.

7) First verify that new entries have been written. If so, check to see if there were previous entries. If there were previous entries, chain the descriptor before the first new descriptor written to the new list..

8) If there were new entries written, but there were no previous entries, then the new entries constitute a net transmit descriptor list for the channel in question. Start the transmitter by writing a pointer to the head of the new list (the saved `pDescOrg` value) to TX*n*HDP. The correct index to use is based on the transmit channel being processed.

The source code to implement this function is in Figure 4–18.

Figure 4−18. Enqueue Transmit Descriptor Function Code

```
static void EnqueueTx( EMI_DescCh *pdc )
{
    EMAC_Desc   *pDescOrg,*pDescThis;
    EMI_Pkt     *pPkt;
    uint        PktFrags;
    uint        CountOrg;

    /*
    // We need to be careful that we don't post half a packet to
    // the list. Otherwise; we just fill in as much packet descriptor
    // stuff as we can.
    */
    pDescOrg = pdc->pDescWrite;
    CountOrg = pdc->DescCount;

    /* Try to post any waiting packets */
    while( pdc->WaitQueue.Count )
    {
        /* See if we have enough room for a new packet */
        pPkt = pdc->WaitQueue.pHead;
        PktFrags = pPkt->PktFrags;

        /* If we don't have room, break out */
        if( (PktFrags+pdc->DescCount) > pdc->DescMax )
            break;

        /* The next packet will fit, post it. */
        while( PktFrags )
        {
            /* Pop the next frag off the wait queue */
            pPkt = pqPop( &pdc->WaitQueue );

            /* Assign the pointer to "this" desc */
            pDescThis = pdc->pDescWrite;

            /* Move the write pointer and bump count */
            if( pdc->pDescWrite == pdc->pDescLast )
                pdc->pDescWrite = pdc->pDescFirst;
            else
                pdc->pDescWrite++;
            pdc->DescCount++;

            /*
            // If this is the last frag, the forward pointer is NULL
            // Otherwise; this desc points to the next frag's desc
            */
```

Figure 4−18. Enqueue Transmit Descriptor Function Code (Continued)

```
            if( PktFrags==1 )
                pDescThis->pNext = 0;
            else
                pDescThis->pNext = pdc->pDescWrite;

            pDescThis->pBuffer   = pPkt->pDataBuffer + pPkt->DataOffset;
            pDescThis->BufOffLen = pPkt->ValidLen;

            if( pPkt->Flags & EMI_PKT_FLAGS_SOP )
                pDescThis->PktFlgLen =
                            ((pPkt->Flags &
                            (EMI_PKT_FLAGS_SOP|EMI_PKT_FLAGS_EOP))
                            |pPkt->PktLength|EMAC_DSC_FLAG_OWNER);
            else
                pDescThis->PktFlgLen =
                            (pPkt->Flags&
                            EMI_PKT_FLAGS_EOP)
                            |EMAC_DSC_FLAG_OWNER;

            /* Enqueue this frag onto the desc queue */
            pqPush( &pdc->DescQueue, pPkt );
            PktFrags--;
        }
    }

    /* If we posted anything, chain on the list or start the transmitter */
    if( CountOrg != pdc->DescCount )
    {
        if( CountOrg )
        {
            /*
            // Transmitter is already running. Just tack this packet on
            // to the end of the list (we need to "back up" one descriptor)
            */
            if( pDescOrg == pdc->pDescFirst )
                pDescThis = pdc->pDescLast;
            else
                pDescThis = pDescOrg - 1;
            pDescThis->pNext = pDescOrg;
        }
        else
        {
            /* Transmitter is not running, start it up */
            EMAC_RSETI( TXHDP, pdc->ChannelIndex, (Uint32)pDescOrg );
        }
    }
}
```

4.5.4.3 Dequeue Transmit Descriptor Function

Once the EMAC has finished transmitting a packet, it returns the packet buffers associated with packet to the software application in much the same way the newly received receive packets are indicated. The DequeueTX() function removes the completed transmit buffers, returning the buffers to the software application, and marking the descriptors from transmit channel the descriptor list free for use for more transmit operations.

To understand this function better, it is important to know what happens during a device interrupt. The ISR code relating to the receive operation is:

```
/* Look for transmit interrupt (channel 0-max) */
for( tmp=0; tmp<pd->Config.TxChannels; tmp++ )
   if( intflags & EMAC_FMK( MACINVECTOR, TXPEND, 1<<tmp ) )
    {
        Desc = EMAC_RGETI( TXINTACK, tmp );
        EMAC_RSETI( TXINTACK, tmp, Desc );

        DequeueTx( &pd->TxCh[tmp], (EMAC_Desc *)Desc );
    }
```

For each active channel in the system, the TXPEND register is examined to see if the particular channel has seen new activity. Next the last descriptor to process in the given channel can be read from TX*n*INTACK register, where the index is based on the channel number. This is also the register where we write value of the last descriptor processed. Since the DequeueTx() function processes all descriptors up to the one that it is passed, the transmit interrupt can be immediately acknowledged by writing the value back to the TX*n*INTACK register. Finally the DequeueTx() function is called with a pointer to the transmit descriptor channel structure and a pointer to the last descriptor to service.

The DequeueTx() function is shown in Figure 4–19. The functions it needs to perform are:

1) The next descriptor to process is always available at the `pDescRead` pointer. The flags for that descriptor are read. The only flag that is important here is the EMAC_DSC_FLAG_EOQ flag that is checked at the end of the loop.

2) The `EMI_Pkt` structure corresponding to the descriptor is recovered from the `DescQueue`. This buffer is returned to the application by use of a callback function.

3) If this is the last descriptor to process (`pDescRead == pDescAck`), then a flag is set to prevent the loop from executing again.

4) The `pDescRead` pointer is incremented and the `DescCount` is decremented.

5) Continue until all the descriptors have been processed up to and including that indicated by the caller (in this case the ISR).

6) As a final step, if the last descriptor processed had the EMAC_DSC_FLAG_EOQ flag set in its flags field, this means that the EMAC interpreted the descriptor as being the last in the descriptor chain (its next pointer was NULL). This occurs if there are no more packets to transmit, or if any newly chained packets were chained on after the transmitter stopped. If the EOQ flag was set and there are more packet descriptors waiting, then restart the transmitter by posting the head of the descriptor list (`pDescRead`) to TX*n*HDP.

7) As a final step, since descriptor entries have been freed, if there are more transmit packets waiting on the `WaitQueue` (waiting to be added to the descriptor list), then call the EnqueueTX() function to enqueue these packets.

The source code to implement this function is in Figure 4–19.

Figure 4−19. Dequeue Transmit Descriptor Function Code

```
static void DequeueTx( EMI_DescCh *pdc, EMAC_Desc *pDescAck )
{
    EMI_Pkt     *pPkt;
    uint        tmp;
    Uint32      PktFlgLen;

    /* Pop & Free Buffers 'till the last Descriptor */
    for( tmp=1; tmp; )
    {
        /* Get the status of this descriptor */
        PktFlgLen = pdc->pDescRead->PktFlgLen;

        /* Recover the buffer and free it */
        pPkt = pqPop( &pdc->DescQueue );
        if( pPkt )
            (*localDev.Config.pfcbFreePacket)(pdc->pd->hApplication,pPkt);

        /* See if this was the last buffer */
        if( pdc->pDescRead == pDescAck )
            tmp = 0;

        /* Move the read pointer and decrement count */
        if( pdc->pDescRead == pdc->pDescLast )
            pdc->pDescRead = pdc->pDescFirst;
        else
            pdc->pDescRead++;
        pdc->DescCount--;
    }

    /* If the transmitter stopped and we have more descriptors, then restart */
    if( (PktFlgLen & EMAC_DSC_FLAG_EOQ) && pdc->DescCount )
        EMAC_RSETI( TXHDP, pdc->ChannelIndex, (Uint32)pdc->pDescRead );

    /* Try to post any waiting transmit packets */
    if( pdc->WaitQueue.Count )
        EnqueueTx( pdc );
}
```

4.5.5 Interrupt Processing

The interrupt signals on the EMAC and MDIO are combined into a single interrupt inside the EMAC control module. The interrupt is used to signal the application or device driver that work needs to be done on the EMAC or MDIO.

All the interrupt signals are combined in the EMAC control module, and this combined set is also fed back into the EMAC module and can be examined by software by reading the MACINVECTOR register. Note that this register represents the masked set of interrupt bits. If an interrupt is not enabled in its corresponding register on the EMAC or the MDIO, then its interrupt bit in the MACINVECTOR register will never be set.

The example software does not use interrupts on the MDIO module. This is because the same operations can be performed as a timer event driven state machine. There is no need for real time caliber response times in servicing MDIO.

4.5.5.1 Interrupt Deferral

Depending on the run-time environment, an application or device driver may or may not do any actual processing in its ISR. For example, consider a system that calls a function like netISR(), where the job of the function is just to turn off the device ISR and return TRUE if the device generated the interrupt, and FALSE if it did not. In a system like this, another work function would be called to actually do the ISR servicing, but not at interrupt time. An implementation of netISR() may look like:

```
netISR()
{
    Uint32      intflags;

       /* Read the interrupt cause */
       if( (intflags = EMAC_RGET( MACINVECTOR )) != 0 )
    {
    /* Disable EMAC/MDIO interrupts in the control module */
    EMAC_FSETS( EWCTL, INTEN, DISABLE );

    /* Tell the caller it was our interrupt */
    return(1);
    }

       /* Tell the caller it was not our interrupt */
       return(0);
}
```

Note that this function disables the device interrupt if it is going to return TRUE. The interrupt is then reenabled once processing is done.

When interrupt pacing is used (programmed using the EWINTTCNT register), the interrupt pace counter does not start counting down until interrupts are reenabled in EWCTL. Thus, if a static pace time is used (where the value of EWINTTCNT is not changed), the delay from the time netISR() is called to the time the interrupts are reenabled in EWCTL can alter interrupt timing. If a static count in EWINTTCNT is used, and the interrupts are certain to be serviced in that amount of time allotted via this register, then it is acceptable to rewrite the previous function as follows:

```
netISR()
{
    Uint32      intflags;

    /* Read the interrupt cause */
    if( (intflags = EMAC_RGET( MACINVECTOR )) != 0 )
    {
    /* Disable EMAC/MDIO interrupts in the control module */
    EMAC_FSETS( EWCTL, INTEN, DISABLE );

    /* Start counter to Re-Enable EMAC/MDIO interrupts */
    EMAC_FSETS( EWCTL, INTEN, ENABLE );

    /* Tell the caller it was our interrupt */
    return(1);
    }

    /* Tell the caller it was not our interrupt */
    return(0);
}
```

Keep in mind that this is only one approach to handling interrupts. In the example code, the interrupt processing is done directly by the ISR, and not deferred.

4.5.5.2 Interrupt Handling

As can be seen in the definition of the MACINVECTOR register, there are six reasons the EMAC control module interrupt can fire. They are listed in Table 4–1.

Table 4–1. Reasons EMAC Control Module Generates Interrupt

Name	Description
USERINT	The MDIO has completed a read or write access to a PHY control register.
LINKINT	The link status of a PHY monitored by the MDIO has changed.
HOSTPEND	A host interrupt is pending on the EMAC. This signifies an error condition.
STATPEND	One of the EMAC statistics registers is in danger of overflow (has its MSB set).
RXPEND	One or more of the 8 receive channels needs servicing.
TXPEND	One or more of the 8 transmit channels needs servicing.

The sample code does not use either the USERINT or LINKINT interrupt signals. The USERINT signal is only good for accessing PHY configuration registers as a background task through the MDIO module. Although accessing PHY configuration register does take many cycles, it is only done at initialization, and does not need to be a general background task. The LINKINT interrupt generates when the link status changes on a monitored PHY. However, since link status can take up to 3 seconds to change, it is perfectly acceptable to poll for this condition. An interrupt is not necessary. This is discussed more in section 4.4.

An excerpt form the sample code interrupt processing is shown in Figure 4–20. Note that is processing is independent of the DSP interrupt. The DSP interrupt is handled in the normal fashion. This interrupt processing code performs:

- ❏ Disable device interrupts by writing the EWCTL register. Note that this serves two purposes. It drives the interrupt signal low, so that the next rise triggers an interrupt on the DSP (that is edge triggered). Also, disabling then reenabling interrupts in the EWCTL register restarts the pace counter (when used) that determines when another interrupt can be generated to the DSP.
- ❏ The MACINVECTOR register is read into a temporary register. This value contains flags representing the state of every possible interrupt source on the EMAC and MDIO modules.

- When the HOSTPEND bit is set, the EMAC has encountered an error caused by the host software. The error status is reported to the application using a callback so that the application can correct the problem and reset the device.
- When the STATPEND bit is set, one of the EMAC statistics registers is in danger of overflow. Thus, the software calls a function to read and reset all the statistics values and keep a soft copy locally. It then notifies the application using a callback so that the application can read the new statistics values. However, since the EMAC statistics registers have already been read and cleared, the sample code does not need to rely on the application responding to the callback to clear the interrupt condition.
- Next check for each of the eight possible TXPEND bits, depending on how many transmit channels are in use. For each transmit channel requiring servicing, service it in accordance with the procedure outlined in section 4.5.4.
- Next check for each of the eight possible RXPEND bits, depending on how many receive channels are in use. This sample code only uses a single receive channel, so there is no for-next loop. If the receive channel requires servicing, service it in accordance with the procedure outlined in section 4.5.3.
- As a final step, interrupts are reenabled by writing the EWCTL register. If an interrupt is still pending, this causes another rising edge and retriggers the DSP interrupt. Interrupts are rearmed immediately, if interrupt pacing is not used. If a count is programmed into the EWINTTCNT register, then interrupts are not rearmed until that value of peripheral clock cycles have expired. The peripheral clock is CPUclk/4.

The source code to perform this operation is in Figure 4–20. The function UpdateStats() is used to read the statistics and then clear the statistics register. This is done by writing back the value for each statistic read to its corresponding register. The registers are write-to-decrement, so no stats are lost.

Figure 4–20. Interrupt Processing Example Code

```
Uint32      intflags,Desc;
uint        tmp;

/* Disable EMAC/MDIO interrupts in the control module */
EMAC_FSETS( EWCTL, INTEN, DISABLE );

/* Read the interrupt cause */
intflags = EMAC_RGET( MACINVECTOR );

/* Look for fatal errors first */
if( intflags & EMAC_FMK( MACINVECTOR, HOSTPEND, 1 ) )
{
    /* Read the error status - we'll decode it by hand */
    pd->FatalError = EMAC_RGET( MACSTATUS );
    /* Tell the application */
    (*localDev.Config.pfcbStatus)(pd->hApplication);
    /* return with interrupts still disabled in the control module */
    return;
}

/* Look for statistics interrupt */
if( intflags & EMAC_FMK( MACINVECTOR, STATPEND, 1 ) )
{
    /* Read the stats and write-decrement what we read */
    /* This is necessary to clear the interrupt */
    UpdateStats( pd );
    /* Tell the application */
    (*localDev.Config.pfcbStatistics)(pd->hApplication);    /*
}

/* Look for transmit interrupt (channel 0-max) */
for( tmp=0; tmp<pd->Config.TxChannels; tmp++ )
    if( intflags & EMAC_FMK( MACINVECTOR, TXPEND, 1<<tmp ) )
    {
        Desc = EMAC_RGETI( TXINTACK, tmp );
        EMAC_RSETI( TXINTACK, tmp, Desc );
        DequeueTx( &pd->TxCh[tmp], (EMAC_Desc *)Desc );
    }

/* Look for receive interrupt (channel 0) */
if( intflags & EMAC_FMK( MACINVECTOR, RXPEND, 1<<0 ) )
{
    Desc = EMAC_RGET( RX0INTACK );
    EMAC_RSET( RX0INTACK, Desc );
    DequeueRx( &pd->RxCh, (EMAC_Desc *)Desc );
}

/* Enable EMAC/MDIO interrupts in the control module */
EMAC_FSETS( EWCTL, INTEN, ENABLE );
```

4.5.6 Shutdown and Restarts

A shutdown is necessary to make sure the EMAC does not continue to access DSP memory (or generate interrupts) after the device is closed. Also, a graceful shutdown is the first stage of a proper device restart.

The example software discussed in this chapter implements device restart and a call to its close function followed by a second call to open. The open operation and device initialization steps are discussed earlier in this chapter. This section describes the device close procedure. The steps for shutting down the device are:

1) Disable device interrupts by writing the EWCTL register. This prevents further interrupts from the device. It is assumed that the DSP interrupt to which the EMAC control module is mapped has also been masked, and any pending condition cleared after this close function is complete (and most likely remain masked).

2) Initiate a teardown of each channel in use by using the RXTEARDOWN and TXTEARDOWN registers. In the example code, there is only one receive channel, but up to eight transmit channels.

3) When the HOSTPEND bit is set in the ISR, a fatal error occurs. If this close operation was started after a fatal error, then the teardown operations will never complete. Thus, the fatal error status of the device is checked before waiting for teardown to complete.

4) If no fatal error occurred, then the software should wait for the shutdown operation to complete on each channel by reading the RX*n*INTACK and TX*n*INTACK registers for each corresponding channel. The register reads FFFF FFFCh when the teardown operation is complete. This value is then written back to RX*n*INTACK or TX*n*INTACK by the software to acknowledge the teardown completion indication.

5) Clear the MACCCONTROL, RXCONTROL, and TXCONTROL registers.

6) Finally, clean up the software environment. In the example code, this involves releasing all memory buffers back to the application using a callback function.

The source code to implement this operation is shown in Figure 4–21.

Figure 4−21. Device Shutdown Example Code

```
/* Disable EMAC/MDIO interrupts in wrapper */
EMAC_FSETS( EWCTL, INTEN, DISABLE );

/*
// The close process consists of tearing down all the active
// channels (receive and transmit) and then waiting for the teardown
// complete indication from the MAC. Then, all queued packets
// will be returned.
*/

/* Teardown receive */
EMAC_RSET( RXTEARDOWN, 0 );

/* Teardown transmit channels in use */
for( i=0; i<pd->Config.TxChannels; i++)
    EMAC_RSET( TXTEARDOWN, i );

/* Only check teardown status if there was no fatal error         */
/* Otherwise; the EMAC is halted and can't be shutdown gracefully */
if( !pd->FatalError )
{
    /* Wait for the teardown to complete */
    for( tmp=0; tmp!=0xFFFFFFFC; tmp=EMAC_RGET(RX0INTACK) );
    EMAC_RSET( RX0INTACK, tmp );
    for( i=0; i<pd->Config.TxChannels; i++ )
    {
        for( tmp=0; tmp!=0xFFFFFFFC; tmp=EMAC_RGETI(TXINTACK,i) );
        EMAC_RSETI( TXINTACK, i, tmp );
    }
}
/* Disable RX, transmit, and clear MACCONTROL */
EMAC_FSETS( TXCONTROL, TXEN, DISABLE );
EMAC_FSETS( RXCONTROL, RXEN, DISABLE );
EMAC_RSET( MACCONTROL, 0 );

/* Free all receive buffers */
while( pPkt = pqPop( &pd->RxCh.DescQueue ) )
    (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt);

/* Free all transmit buffers */
for( i=0; i<pd->Config.TxChannels; i++)
{
    while( pPkt = pqPop( &pd->TxCh[i].DescQueue ) )
        (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt);
    while( pPkt = pqPop( &pd->TxCh[i].WaitQueue ) )
        (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt);
}
```

Chapter 5

Registers

This chapter describes the registers of the EMAC control module, EMAC module, and MDIO module. As the supported feature set may vary between C6000 devices, all of the registers and fields described in this chapter are not supported on each C6000 device. Please see the device-specific datasheet for a listing of supported features.

Topic **Page**

5.1 EMAC Control Module Registers

Control registers for the EMAC control module are summarized in Table 5−1. See the device-specific datasheet for the memory address of these registers.

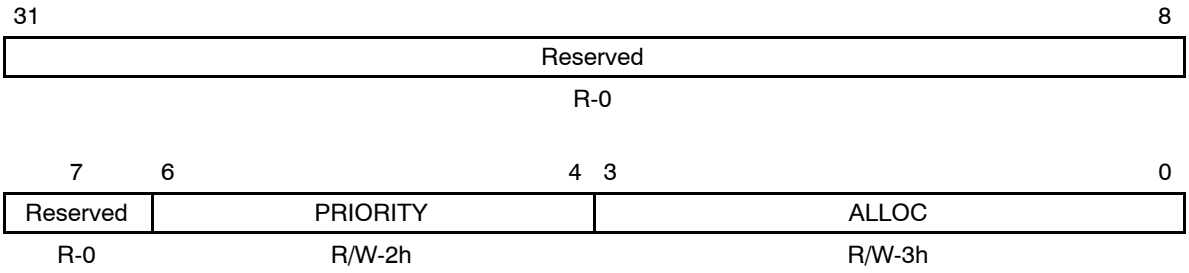
Table 5−1. EMAC Control Module Registers

Acronym	Register Name	Section
EWTRCTRL	EMAC Control Module Transfer Control Register	5.1.1
EWCTL	EMAC Control Module Interrupt Control Register	5.1.2
EWINTTCNT	EMAC Control Module Interrupt Timer Count Register	5.1.3

5.1.1 EMAC Control Module Transfer Control Register (EWTRCTRL)

The EMAC control module transfer control register (EWTRCTRL) is shown in Figure 5−1 and described in Table 5−2. EWTRCTRL is used to control the priority and allocation of transfer requests generated by the EMAC. EWTRCTRL should be written only when the EMAC is idle or when being held in reset using the EWCTL register.

Figure 5−1. EMAC Control Module Transfer Control Register (EWTRCTRL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−2. EMAC Control Module Transfer Control Register (EWTRCTRL) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−7	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
6−4	PRIORITY		0−7h	Priority bits specify the relative priority of EMAC packet data transfers relative to other memory operations in the system. Although the default value is medium priority, since the EMAC data transfer is real time (once a packet transfer begins), this priority may need to be raised in some system.
			0	Urgent priority
			1h	High priority
			2h	Medium priority
			3h	Low priority
			4h−7h	Reserved
3−0	ALLOC		0−Fh	Allocation bits specifiy the number of outstanding EMAC requests that can be pending at any given time. Since the EMAC has only three internal FIFOs, an allocation amount of 3 is ideal.

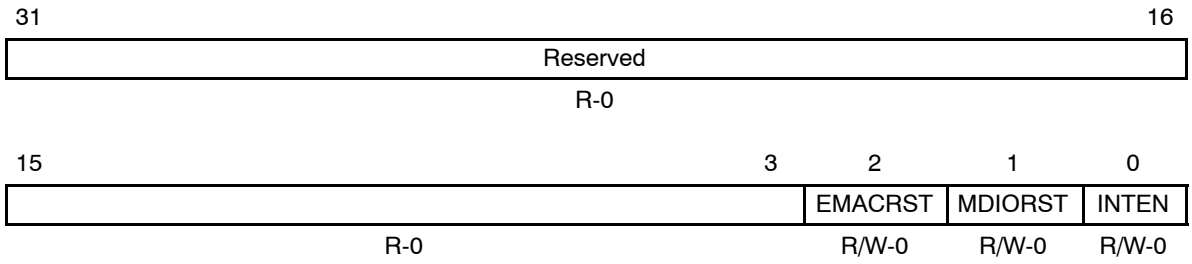
† For CSL implementation, use the notation EMAC_EWTRCTRL_*field_symval*

5.1.2 EMAC Control Module Interrupt Control Register (EWCTL)

The EMAC control module interrupt control register (EWCTL) is shown in Figure 5−2 and described in Table 5−3. EWCTL is used to enable and disable the central interrupt from the EMAC and MDIO modules and to reset both modules or either module independently.

It is expected that any time, the EMAC and MDIO interrupt is being serviced, the software disables the INTEN bit in EWCTL. This ensures that the interrupt line goes back to zero. The software reenables the INTEN bit after clearing all the pending interrupts and before leaving the interrupt service routine. At this point, if the EMAC control module monitors any interrupts still pending, it reasserts the interrupt line, and generates a new edge that the DSP can recognize.

Figure 5−2. EMAC Control Module Interrupt Control Register (EWCTL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−3. EMAC Control Module Interrupt Control Register (EWCTL) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−3	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
2	EMACRST			EMAC reset bit.
		NO	0	EMAC is not in reset.
		YES	1	EMAC is held in reset.
1	MDIORST			MDIO reset bit.
		NO	0	MDIO is not in reset.
		YES	1	MDIO is held in reset.
0	INTEN			EMAC and MDIO interrupt enable bit.
		DISABLE	0	EMAC and MDIO interrupts are disabled.
		ENABLE	1	EMAC and MDIO interrupts are enabled.

† For CSL implementation, use the notation EMAC_EWCTL_*field_symval*

5.1.3 EMAC Control Module Interrupt Timer Count Register (EWINTTCNT)

The EMAC control module interrupt timer count register (EWINTTCNT) is shown in Figure 5−3 and described in Table 5−4. EWINTTCNT is used to control the generation of back-to-back interrupts from the EMAC and MDIO modules. The value of this timer count is loaded into an internal counter every time interrupts are enabled using the EWCTL register. A second interrupt cannot be generated until this count reaches 0. The counter is decremented at a frequency of CPUclock/4; its default reset count is 0 (inactive), its maximum value is 1 FFFFh (131 071).

Figure 5−3. EMAC Control Module Interrupt Timer Count Register (EWINTTCNT)

31	17	16	0
Reserved		EWINTTCNT	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−4. EMAC Control Module Interrupt Timer Count Register (EWINTTCNT) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31−17	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
16−0	EWINTTCNT		0−1 FFFFh	Interrupt timer count.

† For CSL implementation, use the notation EMAC_EWINTTCNT_EWINTTCNT_*symval*

5.2 EMAC Module Registers

Control registers for the EMAC module are summarized in Table 5−5. See the device-specific datasheet for the memory address of these registers. Please see the device-specific datasheet for a listing of supported registers.

Table 5−5. EMAC Module Registers

Acronym	Register Name	Section
TXIDVER	Transmit Identification and Version Register	5.2.1
TXCONTROL	Transmit Control Register	5.2.2
TXTEARDOWN	Transmit Teardown Register	5.2.3
RXIDVER	Receive Identification and Version Register	5.2.4
RXCONTROL	Receive Control Register	5.2.5
RXTEARDOWN	Receive Teardown Register	5.2.6
RXMBPENABLE	Receive Multicast/Broadcast/Promiscuous Channel Enable Register	5.2.7
RXUNICASTSET	Receive Unicast Set Register	5.2.8
RXUNICASTCLEAR	Receive Unicast Clear Register	5.2.9
RXMAXLEN	Receive Maximum Length Register	5.2.10
RXBUFFEROFFSET	Receive Buffer Offset Register	5.2.11
RXFILTERLOWTHRESH	Receive Filter Low Priority Packets Threshold Register	5.2.12
RX*n*FLOWTHRESH	Receive Channel 0−7 Flow Control Threshold Registers	5.2.13
RX*n*FREEBUFFER	Receive Channel 0−7 Free Buffer Count Registers	5.2.14
MACCONTROL	MAC Control Register	5.2.15
MACSTATUS	MAC Status Register	5.2.16
TXINTSTATRAW	Transmit Interrupt Status (Unmasked) Register	5.2.17
TXINTSTATMASKED	Transmit Interrupt Status (Masked) Register	5.2.18
TXINTMASKSET	Transmit Interrupt Mask Set Register	5.2.19
TXINTMASKCLEAR	Transmit Interrupt Mask Clear Register	5.2.20
MACINVECTOR	MAC Input Vector Register	5.2.21
RXINTSTATRAW	Receive Interrupt Status (Unmasked) Register	5.2.22
RXINTSTATMASKED	Receive Interrupt Status (Masked) Register	5.2.23
RXINTMASKSET	Receive Interrupt Mask Set Register	5.2.24

Table 5–5. EMAC Module Registers (Continued)

Acronym	Register Name	Section
RXINTMASKCLEAR	Receive Interrupt Mask Clear Register	5.2.25
MACINTSTATRAW	MAC Interrupt Status (Unmasked) Register	5.2.26
MACINTSTATMASKED	MAC Interrupt Status (Masked) Register	5.2.27
MACINTMASKSET	MAC Interrupt Mask Set Register	5.2.28
MACINTMASKCLEAR	MAC Interrupt Mask Clear Register	5.2.29
MACADDRL*n*	MAC Address Channel 0–7 Lower Byte Register	5.2.30
MACADDRM	MAC Address Middle Byte Register	5.2.31
MACADDRH	MAC Address High Bytes Register	5.2.32
MACHASH1	MAC Address Hash 1 Register	5.2.33
MACHASH2	MAC Address Hash 2 Register	5.2.34
BOFFTEST	Backoff Test Register	5.2.35
TPACETEST	Transmit Pacing Test Register	5.2.36
RXPAUSE	Receive Pause Timer Register	5.2.37
TXPAUSE	Transmit Pause Timer Register	5.2.38
TX*n*HDP	Transmit Channel 0–7 DMA Head Descriptor Pointer Registers	5.2.39
RX*n*HDP	Receive Channel 0–7 DMA Head Descriptor Pointer Registers	5.2.40
TX*n*INTACK	Transmit Channel 0–7 Interrupt Acknowledge Registers	5.2.41
RX*n*INTACK	Receive Channel 0–7 Interrupt Acknowledge Registers	5.2.42
RXGOODFRAMES	Good Receive Frames Register	5.2.43
RXBCASTFRAMES	Broadcast Receive Frames Register	5.2.43
RXMCASTFRAMES	Multicast Receive Frames Register	5.2.43
RXPAUSEFRAMES	Pause Receive Frames Register	5.2.43
RXCRCERRORS	Receive CRC Errors Register	5.2.43
RXALIGNCODEERRORS	Receive Alignment/Code Errors Register	5.2.43
RXOVERSIZED	Receive Oversized Frames Register	5.2.43
RXJABBER	Receive Jabber Frames Register	5.2.43
RXUNDERSIZED	Receive Undersized Frames Register	5.2.43
RXFRAGMENTS	Receive Frame Fragments Register	5.2.43

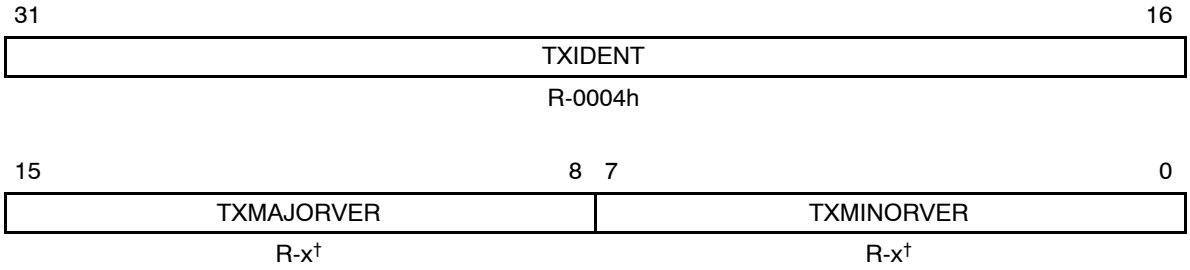
Table 5–5. EMAC Module Registers (Continued)

Acronym	Register Name	Section
RXFILTERED	Filtered Receive Frames Register	5.2.43
RXQOSFILTERED	Receive QOS Filtered Frames Register	5.2.43
RXOCTETS	Receive Octet Frames Register	5.2.43
RXSOFOVERRUNS	Receive Start of Frame Overruns Register	5.2.43
RXMOFOVERRUNS	Receive Middle of Frame Overruns Register	5.2.43
RXDMAOVERRUNS	Receive DMA Overruns Register	5.2.43
TXGOODFRAMES	Good Transmit Frames Register	5.2.43
TXBCASTFRAMES	Broadcast Transmit Frames Register	5.2.43
TXMCASTFRAMES	Multicast Transmit Frames Register	5.2.43
TXPAUSEFRAMES	Pause Transmit Frames Register	5.2.43
TXDEFERRED	Deferred Transmit Frames Register	5.2.43
TXCOLLISION	Collision Register	5.2.43
TXSINGLECOLL	Single Collision Transmit Frames Register	5.2.43
TXMULTICOLL	Multiple Collision Transmit Frames Register	5.2.43
TXEXCESSIVECOLL	Excessive Collisions Register	5.2.43
TXLATECOLL	Late Collisions Register	5.2.43
TXUNDERRUN	Transmit Underrun Register	5.2.43
TXCARRIERSLOSS	Transmit Carrier Sense Errors Register	5.2.43
TXOCTETS	Transmit Octet Frames Register	5.2.43
FRAME64	Transmit and Receive 64 Octet Frames Register	5.2.43
FRAME65T127	Transmit and Receive 65 to 127 Octet Frames Register	5.2.43
FRAME128T255	Transmit and Receive 128 to 255 Octet Frames Register	5.2.43
FRAME256T511	Transmit and Receive 256 to 511 Octet Frames Register	5.2.43
FRAME512T1023	Transmit and Receive 512 to 1023 Octet Frames Register	5.2.43
FRAME1024TUP	Transmit and Receive 1024 or Above Octet Frames Register	5.2.43
NETOCTETS	Network Octet Frames Register	5.2.43

5.2.1 Transmit Identification and Version Register (TXIDVER)

The transmit identification and version register (TXIDVER) is shown in Figure 5−4 and described in Table 5−6.

Figure 5−4. Transmit Identification and Version Register (TXIDVER)



Legend: R = Read only; -*n* = value after reset
† See the device-specific datasheet for the default value of this field.

Table 5−6. Transmit Identification and Version Register (TXIDVER) Field Descriptions

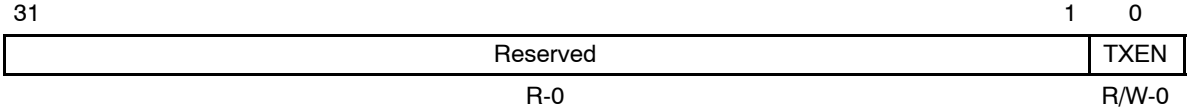
Bit	*field*†	*sym_val*†	Value	Description
31−16	TXIDENT			Transmit identification value bits.
			4h	EMAC
15−8	TXMAJORVER			Transmit major version value is the major version number.
			x	See the device-specific datasheet for the value.
7−0	TXMINORVER			Transmit minor version value is the minor version number.
			x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation EMAC_TXIDVER_*field_symval*

5.2.2 Transmit Control Register (TXCONTROL)

The transmit control register (TXCONTROL) is shown in Figure 5−5 and described in Table 5−7.

Figure 5−5. Transmit Control Register (TXCONTROL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−7. Transmit Control Register (TXCONTROL) Field Descriptions

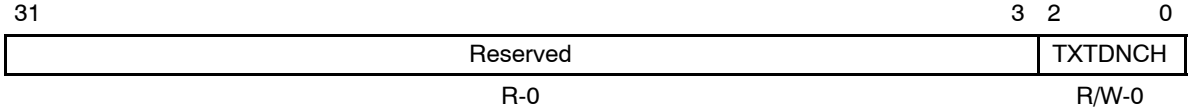
Bit	Field	*sym_val*†	Value	Description
31−1	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
0	TXEN			Transmit enable bit.
		DISABLE	0	Transmit is disabled.
		ENABLE	1	Transmit is enabled.

† For CSL implementation, use the notation EMAC_TXCONTROL_TXEN_*symval*

5.2.3 Transmit Teardown Register (TXTEARDOWN)

The transmit teardown register (TXTEARDOWN) is shown in Figure 5−6 and described in Table 5−8.

Figure 5−6. Transmit Teardown Register (TXTEARDOWN)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−8. Transmit Teardown Register (TXTEARDOWN) Field Descriptions

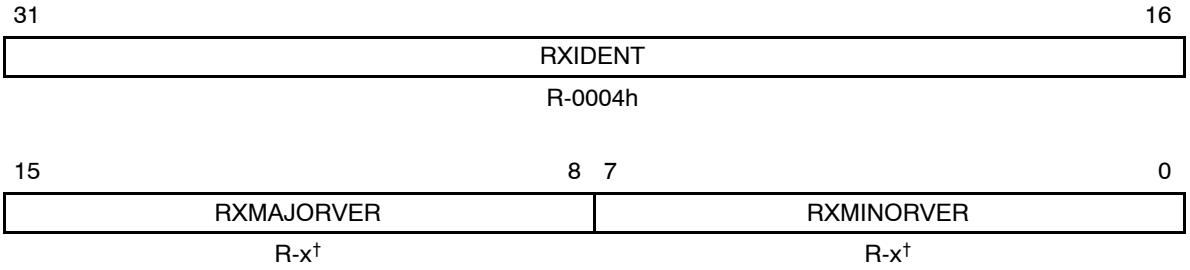
Bit	Field	*sym_val*†	Value	Description
31−3	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
2−0	TXTDNCH		0−7h	Transmit teardown channel bits determine the transmit channel to be torn down. The teardown register is read as 0.
			0	Teardown transmit channel 0.
			1h	Teardown transmit channel 1.
			2h	Teardown transmit channel 2.
			3h	Teardown transmit channel 3.
			4h	Teardown transmit channel 4.
			5h	Teardown transmit channel 5.
			6h	Teardown transmit channel 6.
			7h	Teardown transmit channel 7.

† For CSL implementation, use the notation EMAC_TXTEARDOWN_TXTDNCH_*symval*

5.2.4 Receive Identification and Version Register (RXIDVER)

The receive identification and version register (RXIDVER) is shown in Figure 5−7 and described in Table 5−9.

Figure 5−7. Receive Identification and Version Register (RXIDVER)



Legend: R = Read only; -*n* = value after reset
† See the device-specific datasheet for the default value of this field.

Table 5−9. Receive Identification and Version Register (RXIDVER) Field Descriptions

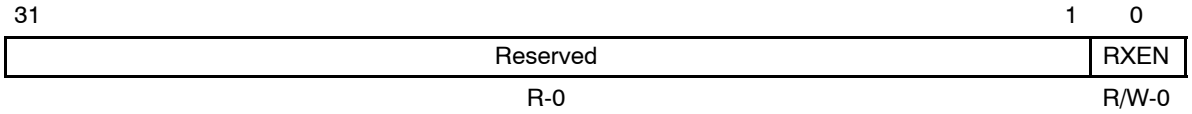
Bit	*field*†	*sym_val*†	Value	Description
31−16	RXIDENT			Receive identification value bits.
			4h	EMAC
15−8	RXMAJORVER			Receive major version value is the major version number.
			x	See the device-specific datasheet for the value.
7−0	RXMINORVER			Receive minor version value is the minor version number.
			x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation EMAC_RXIDVER_*field_symval*

5.2.5 Receive Control Register (RXCONTROL)

The receive control register (RXCONTROL) is shown in Figure 5−8 and described in Table 5−10.

Figure 5−8. Receive Control Register (RXCONTROL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−10. Receive Control Register (RXCONTROL) Field Descriptions

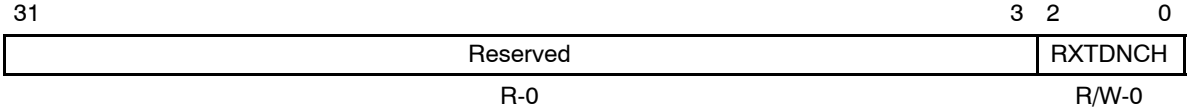
Bit	Field	*sym_val*†	Value	Description
31−1	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
0	RXEN			Receive DMA enable bit.
		DISABLE	0	Receive is disabled.
		ENABLE	1	Receive is enabled.

† For CSL implementation, use the notation EMAC_RXCONTROL_RXEN_*symval*

5.2.6 Receive Teardown Register (RXTEARDOWN)

The receive teardown register (RXTEARDOWN) is shown in Figure 5−9 and described in Table 5−11.

Figure 5−9. Receive Teardown Register (RXTEARDOWN)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−11. Receive Teardown Register (RXTEARDOWN) Field Descriptions

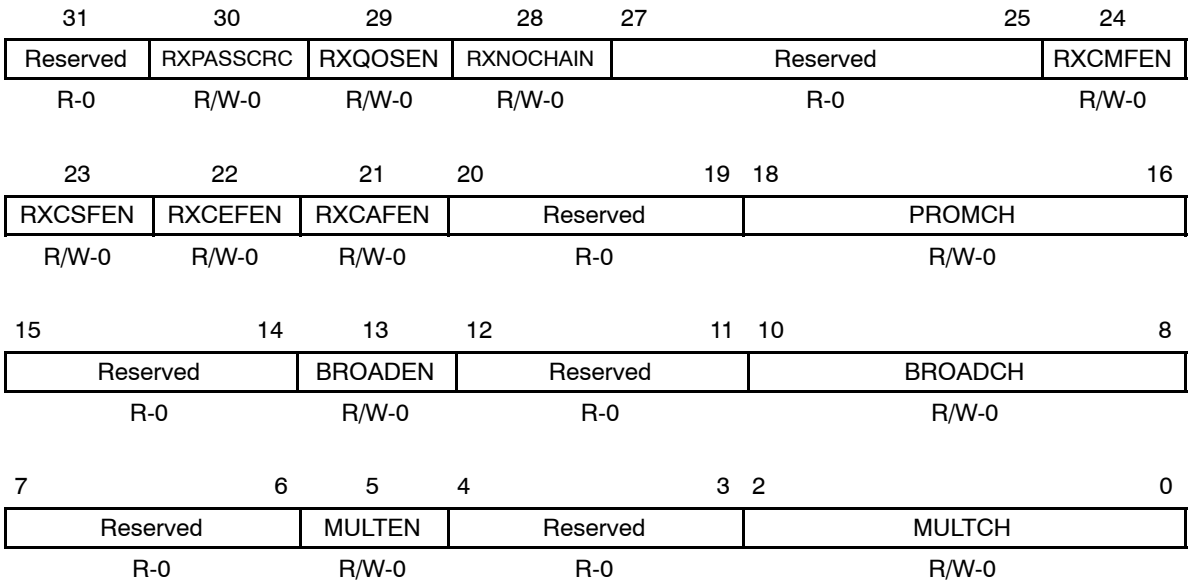
Bit	Field	*sym_val*†	Value	Description
31−3	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
2−0	RXTDNCH		0−7h	Receive teardown channel bits determine the receive channel to be torn down. The teardown register is read as 0.
			0	Teardown receive channel 0.
			1h	Teardown receive channel 1.
			2h	Teardown receive channel 2.
			3h	Teardown receive channel 3.
			4h	Teardown receive channel 4.
			5h	Teardown receive channel 5.
			6h	Teardown receive channel 6.
			7h	Teardown receive channel 7.

† For CSL implementation, use the notation EMAC_RXTEARDOWN_RXTDNCH_*symval*

5.2.7 Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE)

The receive multicast/broadcast/promiscuous channel enable register (RXMBPENABLE) is shown in Figure 5−10 and described in Table 5−12.

Figure 5−10. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−12. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
30	RXPASSCRC			Pass received CRC enable bit.
		DISCARD	0	Received CRC is discarded for all channels and is not included in the buffer descriptor packet length field.
		INCLUDE	1	Received CRC is transferred to memory for all channels and is included in the buffer descriptor packet length.

† For CSL implementation, use the notation EMAC_RXMBPENABLE_*field_symval*

Table 5−12. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
29	RXQOSEN			Receive quality of service (QOS) enable bit.
		DISABLE	0	Receive QOS is disabled.
		ENABLE	1	Receive QOS is enabled.
28	RXNOCHAIN			Receive no buffer chaining bit.
		DISABLE	0	Received frames can span multiple buffers.
		ENABLE	1	Receive DMA controller transfers each frame into a single buffer regardless of the frame or buffer size. All remaining frame data after the first buffer is discarded.
27−25	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
24	RXCMFEN			Receive copy MAC control frames enable bit. Enables MAC control frames to be transferred to memory. MAC control frames are normally acted upon (if enabled), but not copied to memory. MAC control frames that are pause frames will be acted upon if enabled in MACCONTROL, regardless of the value of RXCMFEN. Frames transferred to memory due to RXCMFEN will have the control bit set in their EOP buffer descriptor.
		DISABLE	0	MAC control frames are filtered (but acted upon if enabled).
		ENABLE	1	MAC control frames are transferred to memory.
23	RXCSFEN			Receive copy short frames enable bit. Enables frames or fragments shorter than 64 bytes to be copied to memory. Frames transferred to memory due to RXCSFEN will have the fragment or undersized bit set in their EOP buffer descriptor. Fragments are short frames that contain CRC/align/code errors and undersized are short frames without errors.
		DISABLE	0	Short frames are filtered.
		ENABLE	1	Short frames are transferred to memory.

† For CSL implementation, use the notation EMAC_RXMBPENABLE_*field_symval*

Table 5−12. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
22	RXCEFEN			Receive copy error frames enable bit. Enables frames containing errors to be transferred to memory. The appropriate error bit will be set in the frame EOP buffer descriptor.
		DISABLE	0	Frames containing errors are filtered.
		ENABLE	1	Frames containing errors are transferred to memory.
21	RXCAFEN			Receive copy all frames enable bit. Enables frames that do not address match (includes multicast frames that do not hash match) to be transferred to the promiscuous channel selected by PROMCH bits. Such frames will be marked with the no_match bit in their EOP buffer descriptor.
		DISABLE	0	
		ENABLE	1	Frames that do not address match (includes multicast frames that do not hash match) are transferred to the promiscuous channel selected by PROMCH bits.
20−19	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
18−16	PROMCH		0−7h	Receive promiscuous channel select bits.
			0	Select channel 0 to receive promiscuous frames.
			1h	Select channel 1 to receive promiscuous frames.
			2h	Select channel 2 to receive promiscuous frames.
			3h	Select channel 3 to receive promiscuous frames.
			4h	Select channel 4 to receive promiscuous frames.
			5h	Select channel 5 to receive promiscuous frames.
			6h	Select channel 6 to receive promiscuous frames.
			7h	Select channel 7 to receive promiscuous frames.
15−14	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation EMAC_RXMBPENABLE_*field_symval*

Table 5−12. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
13	BROADEN			Receive broadcast enable bit. Enable received broadcast frames to be copied to the channel selected by BROADCH bits.
		DISABLE	0	Broadcast frames are filtered.
		ENABLE	1	Broadcast frames are copied to the channel selected by BROADCH bits.
12−11	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10−8	BROADCH		0−7h	Receive broadcast channel select bits. Selects the receive channel for reception of all broadcast frames when enabled by BROADEN bit.
			0	Select channel 0 to receive broadcast frames.
			1h	Select channel 1 to receive broadcast frames.
			2h	Select channel 2 to receive broadcast frames.
			3h	Select channel 3 to receive broadcast frames.
			4h	Select channel 4 to receive broadcast frames.
			5h	Select channel 5 to receive broadcast frames.
			6h	Select channel 6 to receive broadcast frames.
			7h	Select channel 7 to receive broadcast frames.
7−6	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
5	MULTEN			Receive multicast enable bit. Enable received hash matching multicast frames to be copied to the channel selected by MULTCH bits.
		DISABLE	0	Multicast (group addressed) frames are filtered.
		ENABLE	1	Multicast frames are copied to the channel selected by MULTCH bits.

† For CSL implementation, use the notation EMAC_RXMBPENABLE_*field_symval*

Table 5−12. Receive Multicast/Broadcast/Promiscuous Channel Enable Register (RXMBPENABLE) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
4−3	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
2−0	MULTCH		0−7h	Receive multicast channel select bits selects the receive channel for reception of all hash matching multicast frames when enabled by MULTEN bit.
			0	Select channel 0 to receive hash matching multicast frames.
			1h	Select channel 1 to receive hash matching multicast frames.
			2h	Select channel 2 to receive hash matching multicast frames.
			3h	Select channel 3 to receive hash matching multicast frames.
			4h	Select channel 4 to receive hash matching multicast frames.
			5h	Select channel 5 to receive hash matching multicast frames.
			6h	Select channel 6 to receive hash matching multicast frames.
			7h	Select channel 7 to receive hash matching multicast frames.

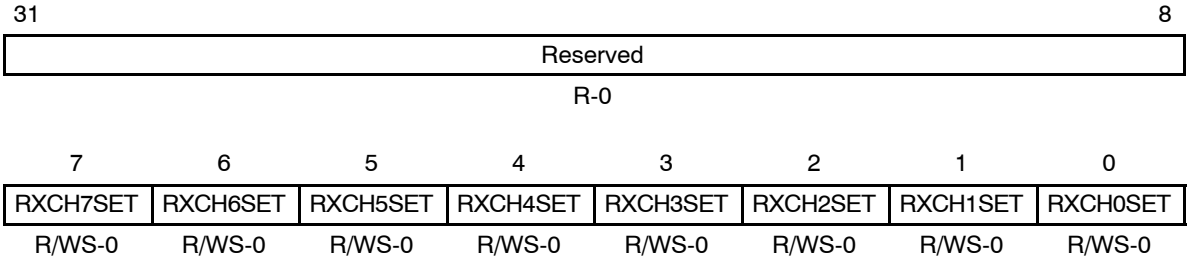
† For CSL implementation, use the notation EMAC_RXMBPENABLE_*field_symval*

5.2.8 Receive Unicast Set Register (RXUNICASTSET)

The receive unicast set register (RXUNICASTSET) is shown in Figure 5−11 and described in Table 5−13.

Each unicast channel is disabled by a write to the corresponding MACADDRL*n*, regardless of the setting of the corresponding bit in RXUNICASTCLEAR. Each unicast channel is enabled by a write to the MACADDRH, if the corresponding bit in RXUNICASTCLEAR is set. Reading the RXUNICASTCLEAR address returns the actual value of the unicast enable register. Reading the RXUNICASTSET address returns the value of the unicast enable register after gating with the MAC address logic.

Figure 5−11. Receive Unicast Set Register (RXUNICASTSET)



Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table 5−13. Receive Unicast Set Register (RXUNICASTSET) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RXCH7SET			Receive channel 7 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 7 unicast enable.
6	RXCH6SET			Receive channel 6 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 6 unicast enable.

† For CSL implementation, use the notation EMAC_RXUNICASTSET_*field_symval*

Table 5−13. Receive Unicast Set Register (RXUNICASTSET) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
5	RXCH5SET			Receive channel 5 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 5 unicast enable.
4	RXCH4SET			Receive channel 4 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 4 unicast enable.
3	RXCH3SET			Receive channel 3 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 3 unicast enable.
2	RXCH2SET			Receive channel 2 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 2 unicast enable.
1	RXCH1SET			Receive channel 1 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 1 unicast enable.
0	RXCH0SET			Receive channel 0 unicast enable set bit. Write 1 to set the enable, a write of 0 has no effect.
			0	No effect.
			1	Sets receive channel 0 unicast enable.

† For CSL implementation, use the notation EMAC_RXUNICASTSET_*field_symval*

5.2.9 Receive Unicast Clear Register (RXUNICASTCLEAR)

The receive unicast clear register (RXUNICASTCLEAR) is shown in Figure 5−12 and described in Table 5−14.

Each unicast channel is disabled by a write to the corresponding MACADDRL*n*, regardless of the setting of the corresponding bit in RXUNICASTCLEAR. Each unicast channel is enabled by a write to the MACADDRH, if the corresponding bit in RXUNICASTCLEAR is set. Reading the RXUNICASTCLEAR address returns the actual value of the unicast enable register. Reading the RXUNICASTSET address returns the value of the unicast enable register after gating with the MAC address logic.

Figure 5−12. Receive Unicast Clear Register (RXUNICASTCLEAR)

31	8
Reserved	
R-0	

7	6	5	4	3	2	1	0
RXCH7CLR	RXCH6CLR	RXCH5CLR	RXCH4CLR	RXCH3CLR	RXCH2CLR	RXCH1CLR	RXCH0CLR
R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−14. Receive Unicast Clear Register (RXUNICASTCLEAR) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RXCH7CLR			Receive channel 7 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 7 unicast enable.
6	RXCH6CLR			Receive channel 6 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 6 unicast enable.

† For CSL implementation, use the notation EMAC_RXUNICASTCLEAR_*field_symval*

Table 5−14. Receive Unicast Clear Register (RXUNICASTCLEAR) Field Descriptions (Continued)

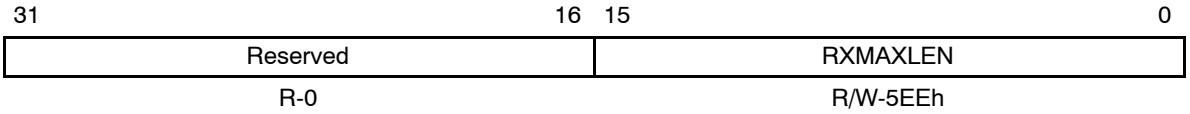
Bit	*field*†	*sym_val*†	Value	Description
5	RXCH5CLR			Receive channel 5 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 5 unicast enable.
4	RXCH4CLR			Receive channel 4 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 4 unicast enable.
3	RXCH3CLR			Receive channel 3 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 3 unicast enable.
2	RXCH2CLR			Receive channel 2 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 2 unicast enable.
1	RXCH1CLR			Receive channel 1 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 1 unicast enable.
0	RXCH0CLR			Receive channel 0 unicast enable clear bit. Write 1 to clear the enable, a write of 0 has no effect.
			0	No effect.
			1	Clears receive channel 0 unicast enable.

† For CSL implementation, use the notation EMAC_RXUNICASTCLEAR_*field_symval*

5.2.10 Receive Maximum Length Register (RXMAXLEN)

The receive maximum length register (RXMAXLEN) is shown in Figure 5−13 and described in Table 5−15.

Figure 5−13. Receive Maximum Length Register (RXMAXLEN)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−15. Receive Maximum Length Register (RXMAXLEN) Field Descriptions

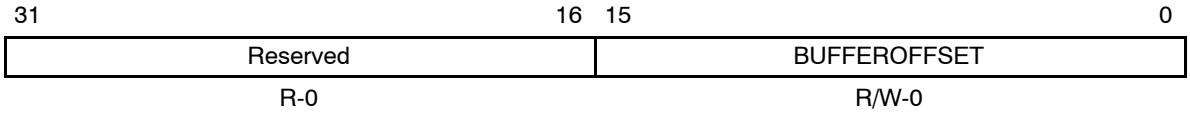
Bit	Field	*sym_val*†	Value	Description
31−16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−0	RXMAXLEN		0−FFFFh	Received maximum frame length bits determine the maximum length of a received frame. The reset value is 5EEh (1518). Frames with byte counts greater than RXMAXLEN are long frames. Long frames with no errors are oversized frames. Long frames with CRC, code, or alignment error are jabber frames.

† For CSL implementation, use the notation EMAC_RXMAXLEN_RXMAXLEN_*symval*

5.2.11 Receive Buffer Offset Register (RXBUFFEROFFSET)

The receive buffer offset register (RXBUFFEROFFSET) is shown in Figure 5−14 and described in Table 5−16.

Figure 5−14. Receive Buffer Offset Register (RXBUFFEROFFSET)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−16. Receive Buffer Offset Register (RXBUFFEROFFSET) Field Descriptions

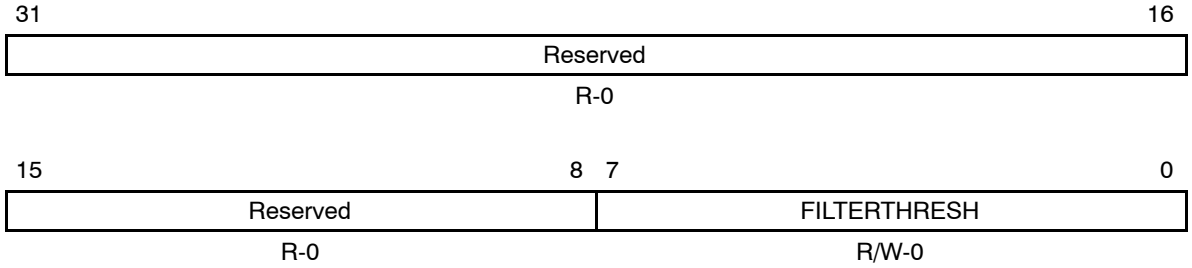
Bit	Field	*sym_val*†	Value	Description
31−16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−0	BUFFEROFFSET		0−FFFFh	Receive buffer offset bits are written by the EMAC into each frame SOP buffer descriptor Buffer Offset field. The frame data begins after the BUFFEROFFSET value of bytes. A value of 0 indicates that there are no unused bytes at the beginning of the data and that valid data begins on the first byte of the buffer. A value of Fh indicates that the first 15 bytes of the buffer are to be ignored by the EMAC and that valid buffer data starts on byte 16 of the buffer. This value is used for all channels.

† For CSL implementation, use the notation EMAC_RXBUFFEROFFSET_BUFFEROFFSET_*symval*

5.2.12 Receive Filter Low Priority Packets Threshold Register (RXFILTERLOWTHRESH)

The receive filter low priority packets threshold register (RXFILTERLOW-THRESH) is shown in Figure 5−15 and described in Table 5−17.

Figure 5−15. Receive Filter Low Priority Packets Threshold Register (RXFILTERLOWTHRESH)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−17. Receive Filter Low Priority Packets Threshold Register (RXFILTERLOWTHRESH) Field Descriptions

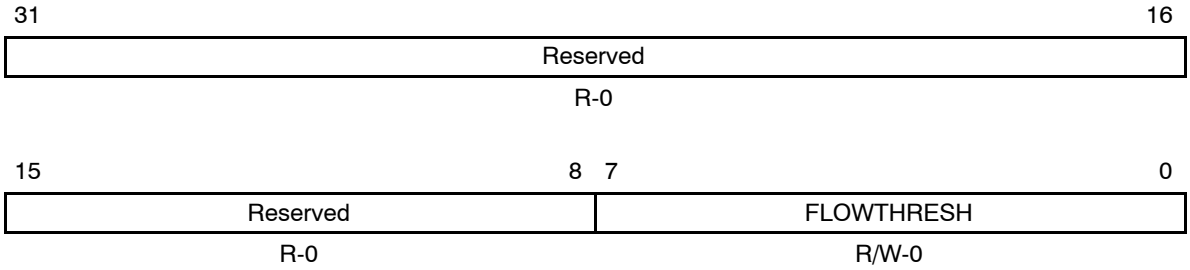
Bit	Field	*sym_val*†	Value	Description
31−8	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7−0	FILTERTHRESH		0−FFh	Receive filter low threshold bits contain the free buffer count threshold value for filtering low priority incoming frames. This field should remain zero, if no filtering is desired.

† For CSL implementation, use the notation EMAC_RXFILTERLOWTHRESH_FILTERTHRESH_*symval*

5.2.13 Receive Channel 0–7 Flow Control Threshold Registers (RX*n*FLOWTHRESH)

The receive channel *n* flow control threshold registers (RX*n*FLOWTHRESH) is shown in Figure 5–16 and described in Table 5–18.

Figure 5–16. Receive Channel n Flow Control Threshold Registers (RXnFLOWTHRESH)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5–18. Receive Channel n Flow Control Threshold Registers (RXnFLOWTHRESH) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7–0	FLOWTHRESH		0–FFh	Receive flow threshold bits contain the threshold value for issuing flow control on incoming frames (when enabled).

† For CSL implementation, use the notation EMAC_RX*n*FLOWTHRESH_FLOWTHRESH_*symval*

5.2.14 Receive Channel 0–7 Free Buffer Count Registers (RX*n*FREEBUFFER)

The receive channel *n* free buffer count registers (RX*n*FREEBUFFER) is shown in Figure 5–17 and described in Table 5–19.

Figure 5–17. Receive Channel n Free Buffer Count Registers (RXnFREEBUFFER)

31	16	15	0
Reserved		FREEBUF	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5–19. Receive Channel n Free Buffer Count Registers (RXnFREEBUFFER) Field Descriptions

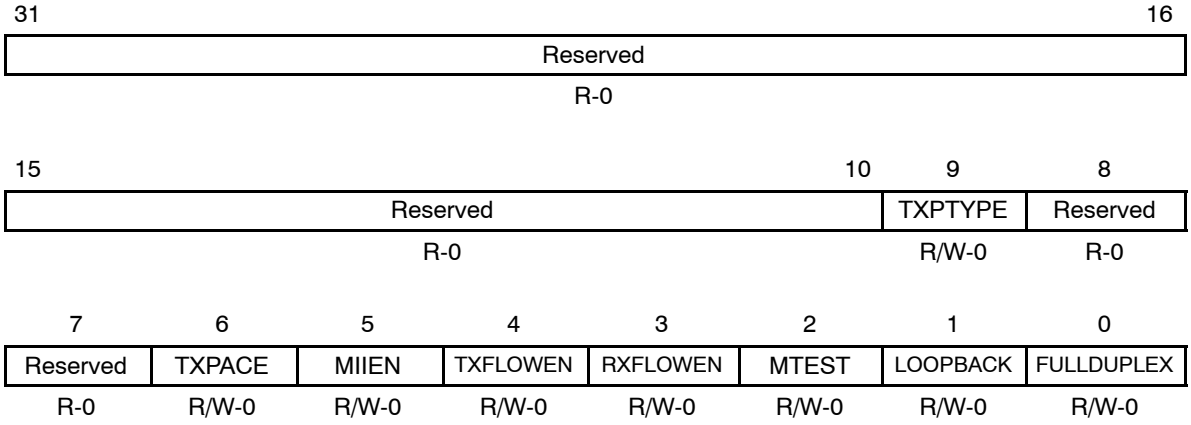
Bit	Field	*sym_val*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15–0	FREEBUF		0–FFFFh	Receive free buffer count bits contain the count of free buffers available. The RXFILTERLOWTHRESH value is compared with this field to determine if low priority frames should be filtered. The RX*n*FLOWTHRESH value is compared with this field to determine if receive flow control should be issued against incoming packets (if enabled). This is a write-to-increment field. This field rolls over to zero on overflow. If hardware flow control or QOS is used, the host must initialize this field to the number of available buffers (one register per channel). The EMAC decrements (by the number of buffers in the received frame) the associated channel register for each received frame. This is a write-to-increment field. The host must write this field with the number of buffers that have been freed due to host processing.

† For CSL implementation, use the notation EMAC_RX*n*FREEBUFFER_FREEBUF_*symval*

5.2.15 MAC Control Register (MACCONTROL)

The MAC control register (MACCONTROL) is shown in Figure 5−18 and described in Table 5−20.

Figure 5−18. MAC Control Register (MACCONTROL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−20. MAC Control Register (MACCONTROL) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−10	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
9	TXPTYPE			Transmit queue priority type bit.
		RROBIN	0	The queue uses a round-robin scheme to select the next channel for transmission.
		CHANNELPRI	1	The queue uses a fixed-priority (channel 7 highest priority) scheme to select the next channel for transmission.
8−7	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
6	TXPACE			Transmit pacing enable bit.
		DISABLE	0	Transmit pacing is disabled.
		ENABLE	1	Transmit pacing is enabled.

† For CSL implementation, use the notation EMAC_MACCONTROL_*field_symval*

Table 5−20. MAC Control Register (MACCONTROL) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
5	MIIEN			MII enable bit.
		DISABLE	0	MII receive and transmit are disabled (state machine reset).
		ENABLE	1	MII receive and transmit are enabled.
4	TXFLOWEN			Transmit flow control enable bit determines if incoming pause frames are acted upon in full-duplex mode. Incoming pause frames are not acted upon in half-duplex mode, regardless of this bit setting. The RXMBPENABLE bits determine whether or not received pause frames are transferred to memory.
		DISABLE	0	Transmit flow control is disabled. Full-duplex mode: incoming pause frames are not acted upon.
		ENABLE	1	Transmit flow control is enabled. Full-duplex mode: incoming pause frames are acted upon.
3	RXFLOWEN			Receive flow control enable bit.
		DISABLE	0	Receive flow control is disabled. Half-duplex mode: no flow control generated collisions are sent. Full-duplex mode: no outgoing pause frames are sent.
		ENABLE	1	Receive flow control is enabled. Half-duplex mode: collisions are initiated when receive flow control is triggered. Full-duplex mode: outgoing pause frames are sent when receive flow control is triggered.
2	MTEST			Manufacturing test mode bit.
		DISABLE	0	Writes to the BOFFTEST, RXPAUSE, and TXPAUSE registers are disabled.
		ENABLE	1	Writes to the BOFFTEST, RXPAUSE, and TXPAUSE registers are enabled.
1	LOOPBACK			Loopback mode enable bit. Loopback mode forces internal full-duplex mode regardless of the FULLDUPLEX bit. The loopback bit should be changed only when MIIEN bit is deasserted.
		DISABLE	0	Loopback mode is disabled.
		ENABLE	1	Loopback mode is enabled.

† For CSL implementation, use the notation EMAC_MACCONTROL_*field_symval*

Table 5−20. MAC Control Register (MACCONTROL) Field Descriptions (Continued)

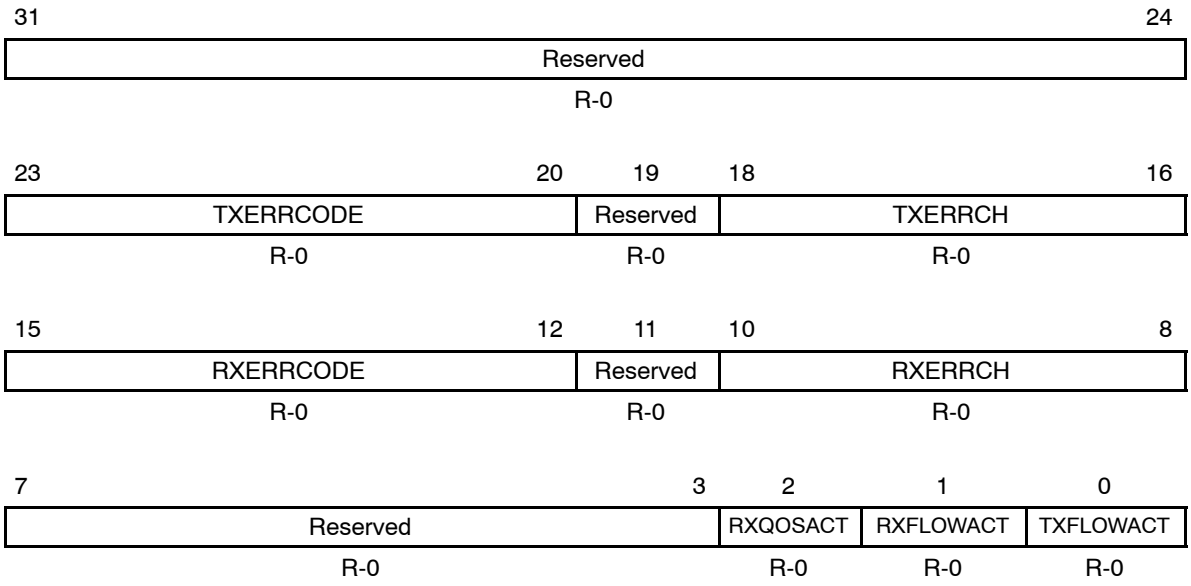
Bit	*field*†	*sym_val*†	Value	Description
0	FULLDUPLEX			Full-duplex mode enable bit.
		DISABLE	0	Half-duplex mode is enabled.
		ENABLE	1	Full-duplex mode is enabled.

† For CSL implementation, use the notation EMAC_MACCONTROL_*field_symval*

5.2.16 MAC Status Register (MACSTATUS)

The MAC status register (MACSTATUS) is shown in Figure 5−19 and described in Table 5−21.

Figure 5−19. MAC Status Register (MACSTATUS)



Legend: R = Read only; -*n* = value after reset

Table 5−21. MAC Status Register (MACSTATUS) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−24	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
23−20	TXERRCODE		0−Fh	Transmit host error code bits indicate EMAC detected transmit DMA related host errors. The host should read this field after a host error interrupt (HOSTERRINT) to determine the error. Host error interrupts require hardware reset in order to recover.
		NOERROR	0	No error
		SOPERROR	1h	SOP error
		OWNERSHIP	2h	Ownership bit is not set in SOP buffer
		NOEOP	3h	Zero next buffer descriptor pointer is without EOP
		NULLPTR	4h	Zero buffer pointer
		NULLEN	5h	Zero buffer length
		LENRRROR	6h	Packet length error
			7h−Fh	Reserved
19	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
18−16	TXERRCH		0−7h	Transmit host error channel bits indicate which transmit channel the host error occurred on. This field is cleared to 0 on a host read.
			0	The host error occurred on transmit channel 0.
			1h	The host error occurred on transmit channel 1.
			2h	The host error occurred on transmit channel 2.
			3h	The host error occurred on transmit channel 3.
			4h	The host error occurred on transmit channel 4.
			5h	The host error occurred on transmit channel 5.
			6h	The host error occurred on transmit channel 6.
			7h	The host error occurred on transmit channel 7.

† For CSL implementation, use the notation EMAC_MACSTATUS_*field_symval*

Table 5−21. MAC Status Register (MACSTATUS) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
15−12	RXERRCODE		0−Fh	Receive host error code bits indicate EMAC detected receive DMA related host errors. The host should read this field after a host error interrupt (HOSTERRINT) to determine the error. Host error interrupts require hardware reset in order to recover.
		NOERROR	0	No error
		SOPERROR	1h	SOP error
		OWNERSHIP	2h	Ownership bit is not set in input buffer
		NOEOP	3h	Zero next buffer descriptor pointer is without eop
		NULLPTR	4h	Zero buffer pointer
		NULLEN	5h	Zero buffer length
		LENRRROR	6h	Packet length error
			7h−Fh	Reserved
11	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10−8	RXERRCH		0−7h	Receive host error channel bits indicate which receive channel the host error occurred on. This field is cleared to 0 on a host read.
			0	The host error occurred on receive channel 0.
			1h	The host error occurred on receive channel 1.
			2h	The host error occurred on receive channel 2.
			3h	The host error occurred on receive channel 3.
			4h	The host error occurred on receive channel 4.
			5h	The host error occurred on receive channel 5.
			6h	The host error occurred on receive channel 6.
			7h	The host error occurred on receive channel 7.
7−3	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation EMAC_MACSTATUS_*field_symval*

Table 5–21. MAC Status Register (MACSTATUS) Field Descriptions (Continued)

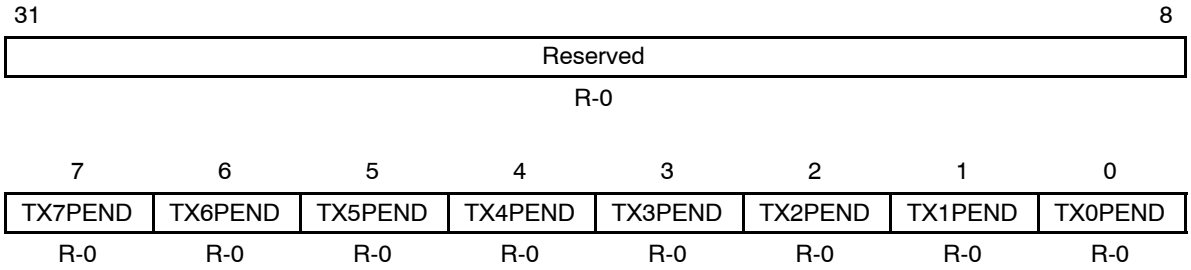
Bit	*field*†	*sym_val*†	Value	Description
2	RXQOSACT			Receive quality of service (QOS) active bit.
			0	Receive quality of service is disabled.
			1	Receive quality of service is enabled and that at least one channel freebuffer count (RX*n*FREEBUFFER) value is less than or equal to the RXFILTERLOWTHRESH value.
1	RXFLOWACT			Receive flow control active bit.
			0	
			1	At least one channel freebuffer count (RX*n*FREEBUFFER) value is less than or equal to the channel's corresponding RX*n*FLOWTHRESH value.
0	TXFLOWACT			Transmit flow control active bit.
			0	
			1	The pause time period is being observed for a received pause frame. No new transmissions begin while this bit is asserted except for the transmission of pause frames. Any transmission in progress when this bit is asserted will complete.

† For CSL implementation, use the notation EMAC_MACSTATUS_*field_symval*

5.2.17 Transmit Interrupt Status (Unmasked) Register (TXINTSTATRAW)

The transmit interrupt status (unmasked) register (TXINTSTATRAW) is shown in Figure 5−20 and described in Table 5−22.

Figure 5−20. Transmit Interrupt Status (Unmasked) Register (TXINTSTATRAW)



Legend: R = Read only; -*n* = value after reset

Table 5−22. Transmit Interrupt Status (Unmasked) Register (TXINTSTATRAW) Field Descriptions

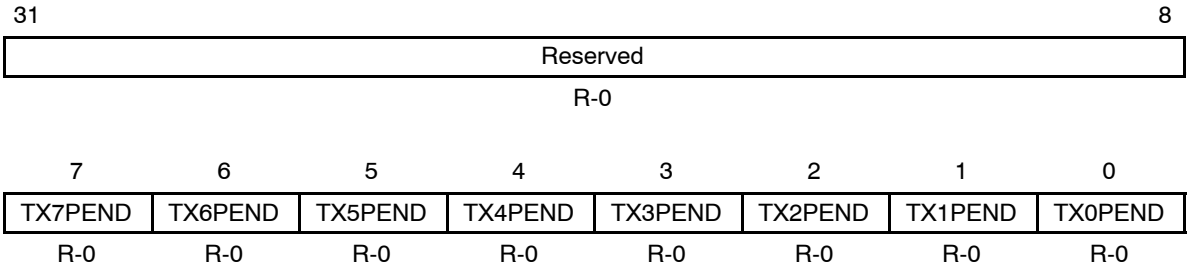
Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	TX7PEND			TX7PEND raw interrupt read (before mask)
6	TX6PEND			TX6PEND raw interrupt read (before mask)
5	TX5PEND			TX5PEND raw interrupt read (before mask)
4	TX4PEND			TX4PEND raw interrupt read (before mask)
3	TX3PEND			TX3PEND raw interrupt read (before mask)
2	TX2PEND			TX2PEND raw interrupt read (before mask)
1	TX1PEND			TX1PEND raw interrupt read (before mask)
0	TX0PEND			TX0PEND raw interrupt read (before mask)

† For CSL implementation, use the notation EMAC_TXINTSTATRAW_*field_symval*

5.2.18 Transmit Interrupt Status (Masked) Register (TXINTSTATMASKED)

The transmit interrupt status (masked) register (TXINTSTATMASKED) is shown in Figure 5−21 and described in Table 5−23.

Figure 5−21. Transmit Interrupt Status (Masked) Register (TXINTSTATMASKED)



Legend: R = Read only; -*n* = value after reset

Table 5−23. Transmit Interrupt Status (Masked) Register (TXINTSTATMASKED) Field Descriptions

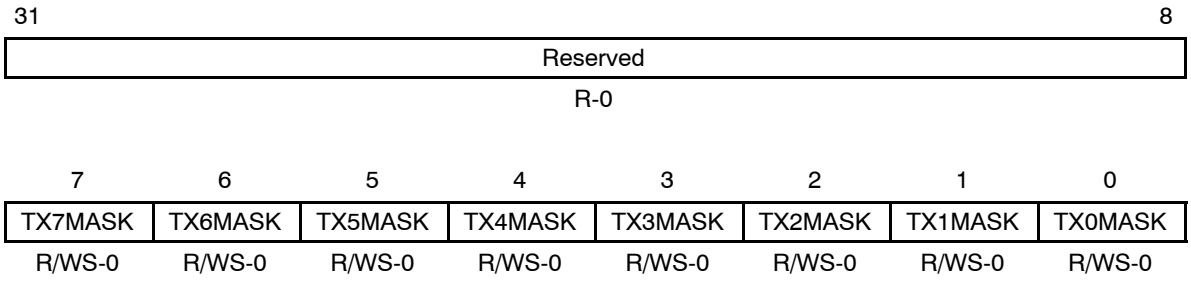
Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	TX7PEND			TX7PEND masked interrupt read
6	TX6PEND			TX6PEND masked interrupt read
5	TX5PEND			TX5PEND masked interrupt read
4	TX4PEND			TX4PEND masked interrupt read
3	TX3PEND			TX3PEND masked interrupt read
2	TX2PEND			TX2PEND masked interrupt read
1	TX1PEND			TX1PEND masked interrupt read
0	TX0PEND			TX0PEND masked interrupt read

† For CSL implementation, use the notation EMAC_TXINTSTATMASKED_*field_symval*

5.2.19 Transmit Interrupt Mask Set Register (TXINTMASKSET)

The transmit interrupt mask set register (TXINTMASKSET) is shown in Figure 5−22 and described in Table 5−24.

Figure 5−22. Transmit Interrupt Mask Set Register (TXINTMASKSET)



Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table 5−24. Transmit Interrupt Mask Set Register (TXINTMASKSET) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	TX7MASK			Transmit channel 7 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 7 interrupt is enabled.
6	TX6MASK			Transmit channel 6 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 6 interrupt is enabled.
5	TX5MASK			Transmit channel 5 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 5 interrupt is enabled.

† For CSL implementation, use the notation EMAC_TXINTMASKSET_*field_symval*

Table 5−24. Transmit Interrupt Mask Set Register (TXINTMASKSET) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
4	TX4MASK			Transmit channel 4 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 4 interrupt is enabled.
3	TX3MASK			Transmit channel 3 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 3 interrupt is enabled.
2	TX2MASK			Transmit channel 2 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 2 interrupt is enabled.
1	TX1MASK			Transmit channel 1 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 1 interrupt is enabled.
0	TX0MASK			Transmit channel 0 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 0 interrupt is enabled.

† For CSL implementation, use the notation EMAC_TXINTMASKSET_*field_symval*

5.2.20 Transmit Interrupt Mask Clear Register (TXINTMASKCLEAR)

The transmit interrupt mask clear register (TXINTMASKCLEAR) is shown in Figure 5−23 and described in Table 5−25.

Figure 5−23. Transmit Interrupt Mask Clear Register (TXINTMASKCLEAR)

31							8
Reserved							
R-0							

7	6	5	4	3	2	1	0
TX7MASK	TX6MASK	TX5MASK	TX4MASK	TX3MASK	TX2MASK	TX1MASK	TX0MASK
R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−25. Transmit Interrupt Mask Clear Register (TXINTMASKCLEAR) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	TX7MASK			Transmit channel 7 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 7 interrupt is disabled.
6	TX6MASK			Transmit channel 6 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 6 interrupt is disabled.
5	TX5MASK			Transmit channel 5 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 5 interrupt is disabled.

† For CSL implementation, use the notation EMAC_TXINTMASKCLEAR_*field_symval*

Table 5–25. Transmit Interrupt Mask Clear Register (TXINTMASKCLEAR) Field Descriptions (Continued)

Bit	*field*†	*sym_val*†	Value	Description
4	TX4MASK			Transmit channel 4 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 4 interrupt is disabled.
3	TX3MASK			Transmit channel 3 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 3 interrupt is disabled.
2	TX2MASK			Transmit channel 2 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 2 interrupt is disabled.
1	TX1MASK			Transmit channel 1 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 1 interrupt is disabled.
0	TX0MASK			Transmit channel 0 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Transmit channel 0 interrupt is disabled.

† For CSL implementation, use the notation EMAC_TXINTMASKCLEAR_*field_symval*

5.2.21 MAC Input Vector Register (MACINVECTOR)

The MAC input vector register (MACINVECTOR) is shown in Figure 5−24 and described in Table 5−26. MACINVECTOR contains the current interrupt status of all individual EMAC and MDIO module interrupts. With a single MACINVECTOR read, you can monitor the status of all device interrupts.

Figure 5−24. MAC Input Vector Register (MACINVECTOR)

31	30	29	18	17	16
USERINT	LINKINT	Reserved		HOSTPEND	STATPEND
R-0	R-0	R-0		R-0	R-0

15	8	7	0
RXPEND		TXPEND	
R-0		R-0	

Legend: R = Read only; -*n* = value after reset

Table 5−26. MAC Input Vector Register (MACINVECTOR) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31	USERINT			MDIO module user interrupt (USERINT) pending status bit.
30	LINKINT			MDIO module link change interrupt (LINKINT) pending status bit.
29−18	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
17	HOSTPEND			EMAC module host error interrupt pending (HOSTPEND) status bit.
16	STATPEND			EMAC module statistics interrupt pending (STATPEND) status bit.
15−8	RXPEND		0−FFh	Receive channel 0−7 interrupt pending (RXPEND) status bit. Bit 8 is receive channel 0.
7−0	TXPEND		0−FFh	Transmit channel 0−7 interrupt pending (TXPEND) status bit. Bit 0 is transmit channel 0.

† For CSL implementation, use the notation EMAC_MACINVECTOR_*field_symval*

5.2.22 Receive Interrupt Status (Unmasked) Register (RXINTSTATRAW)

The receive interrupt status (unmasked) register (RXINTSTATRAW) is shown in Figure 5−25 and described in Table 5−27.

Figure 5−25. Receive Interrupt Status (Unmasked) Register (RXINTSTATRAW)

31							8
Reserved							
R-0							
7	6	5	4	3	2	1	0
RX7PEND	RX6PEND	RX5PEND	RX4PEND	RX3PEND	RX2PEND	RX1PEND	RX0PEND
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

Legend: R = Read only; -*n* = value after reset

Table 5−27. Receive Interrupt Status (Unmasked) Register (RXINTSTATRAW) Field Descriptions

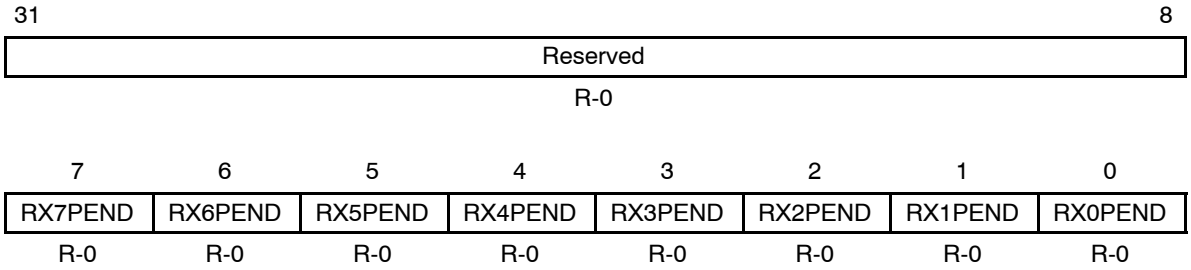
Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RX7PEND			RX7PEND raw interrupt read (before mask)
6	RX6PEND			RX6PEND raw interrupt read (before mask)
5	RX5PEND			RX5PEND raw interrupt read (before mask)
4	RX4PEND			RX4PEND raw interrupt read (before mask)
3	RX3PEND			RX3PEND raw interrupt read (before mask)
2	RX2PEND			RX2PEND raw interrupt read (before mask)
1	RX1PEND			RX1PEND raw interrupt read (before mask)
0	RX0PEND			RX0PEND raw interrupt read (before mask)

† For CSL implementation, use the notation EMAC_RXINTSTATRAW_*field_symval*

5.2.23 Receive Interrupt Status (Masked) Register (RXINTSTATMASKED)

The receive interrupt status (masked) register (RXINTSTATMASKED) is shown in Figure 5−26 and described in Table 5−28.

Figure 5−26. Receive Interrupt Status (Masked) Register (RXINTSTATMASKED)



Legend: R = Read only; -*n* = value after reset

Table 5−28. Receive Interrupt Status (Masked) Register (RXINTSTATMASKED) Field Descriptions

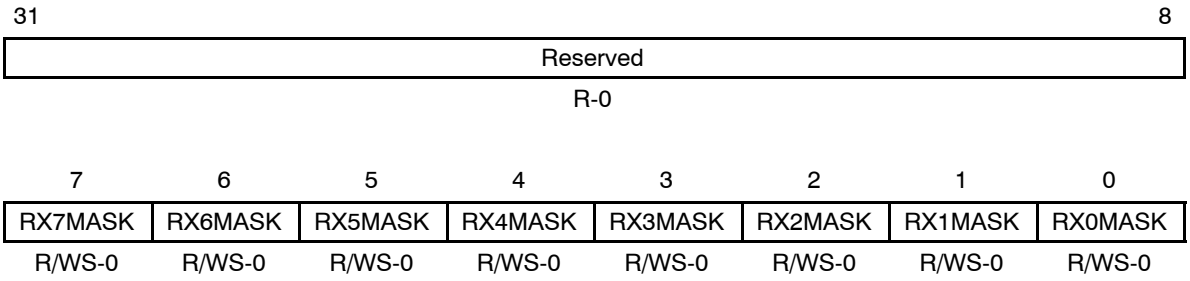
Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RX7PEND			RX7PEND masked interrupt read
6	RX6PEND			RX6PEND masked interrupt read
5	RX5PEND			RX5PEND masked interrupt read
4	RX4PEND			RX4PEND masked interrupt read
3	RX3PEND			RX3PEND masked interrupt read
2	RX2PEND			RX2PEND masked interrupt read
1	RX1PEND			RX1PEND masked interrupt read
0	RX0PEND			RX0PEND masked interrupt read

† For CSL implementation, use the notation EMAC_RXINTSTATMASKED_*field_symval*

5.2.24 Receive Interrupt Mask Set Register (RXINTMASKSET)

The receive interrupt mask set register (RXINTMASKSET) is shown in Figure 5−27 and described in Table 5−29.

Figure 5−27. Receive Interrupt Mask Set Register (RXINTMASKSET)



Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; *-n* = value after reset

Table 5−29. Receive Interrupt Mask Set Register (RXINTMASKSET) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RX7MASK			Receive channel 7 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 7 interrupt is enabled.
6	RX6MASK			Receive channel 6 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 6 interrupt is enabled.
5	RX5MASK			Receive channel 5 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 5 interrupt is enabled.

† For CSL implementation, use the notation EMAC_RXINTMASKSET_*field_symval*

Table 5–29. Receive Interrupt Mask Set Register (RXINTMASKSET) Field Descriptions (Continued)

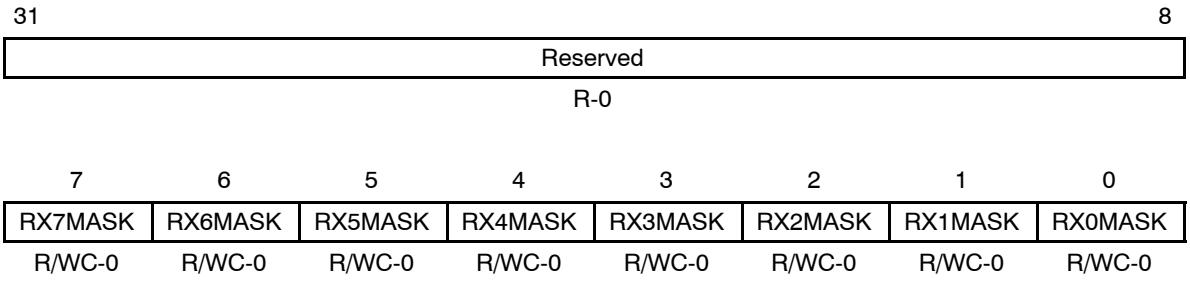
Bit	*field*†	*sym_val*†	Value	Description
4	RX4MASK			Receive channel 4 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 4 interrupt is enabled.
3	RX3MASK			Receive channel 3 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 3 interrupt is enabled.
2	RX2MASK			Receive channel 2 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 2 interrupt is enabled.
1	RX1MASK			Receive channel 1 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 1 interrupt is enabled.
0	RX0MASK			Receive channel 0 interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 0 interrupt is enabled.

† For CSL implementation, use the notation EMAC_RXINTMASKSET_*field_symval*

5.2.25 Receive Interrupt Mask Clear Register (RXINTMASKCLEAR)

The receive interrupt mask clear register (RXINTMASKCLEAR) is shown in Figure 5−28 and described in Table 5−30.

Figure 5−28. Receive Interrupt Mask Clear Register (RXINTMASKCLEAR)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−30. Receive Interrupt Mask Clear Register (RXINTMASKCLEAR) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	RX7MASK			Receive channel 7 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 7 interrupt is disabled.
6	RX6MASK			Receive channel 6 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 6 interrupt is disabled.
5	RX5MASK			Receive channel 5 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 5 interrupt is disabled.

† For CSL implementation, use the notation EMAC_RXINTMASKCLEAR_*field_symval*

Table 5–30. Receive Interrupt Mask Clear Register (RXINTMASKCLEAR) Field Descriptions (Continued)

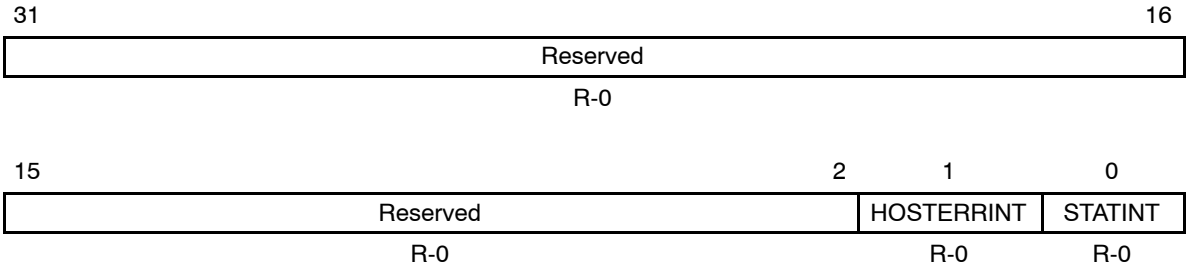
Bit	*field*†	*sym_val*†	Value	Description
4	RX4MASK			Receive channel 4 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 4 interrupt is disabled.
3	RX3MASK			Receive channel 3 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 3 interrupt is disabled.
2	RX2MASK			Receive channel 2 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 2 interrupt is disabled.
1	RX1MASK			Receive channel 1 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 1 interrupt is disabled.
0	RX0MASK			Receive channel 0 interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Receive channel 0 interrupt is disabled.

† For CSL implementation, use the notation EMAC_RXINTMASKCLEAR_*field_symval*

5.2.26 MAC Interrupt Status (Unmasked) Register (MACINTSTATRAW)

The MAC interrupt status (unmasked) register (MACINTSTATRAW) is shown in Figure 5−29 and described in Table 5−31.

Figure 5−29. MAC Interrupt Status (Unmasked) Register (MACINTSTATRAW)



Legend: R = Read only; -*n* = value after reset

Table 5−31. MAC Interrupt Status (Unmasked) Register (MACINTSTATRAW) Field Descriptions

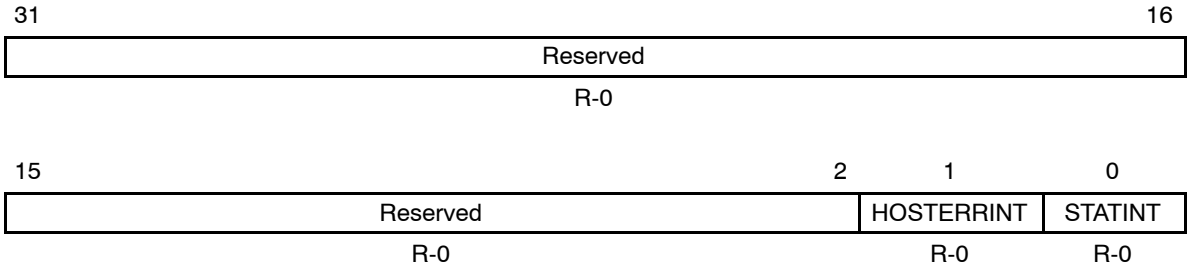
Bit	*field*†	*sym_val*†	Value	Description
31−2	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	HOSTERRINT			Host error interrupt bit. Raw interrupt read (before mask).
0	STATINT			Statistics interrupt bit. Raw interrupt read (before mask).

† For CSL implementation, use the notation EMAC_MACINTSTATRAW_*field_symval*

5.2.27 MAC Interrupt Status (Masked) Register (MACINTSTATMASKED)

The MAC interrupt status (masked) register (MACINTSTATMASKED) is shown in Figure 5−30 and described in Table 5−32.

Figure 5−30. MAC Interrupt Status (Masked) Register (MACINTSTATMASKED)



Legend: R = Read only; -*n* = value after reset

Table 5−32. MAC Interrupt Status (Masked) Register (MACINTSTATMASKED) Field Descriptions

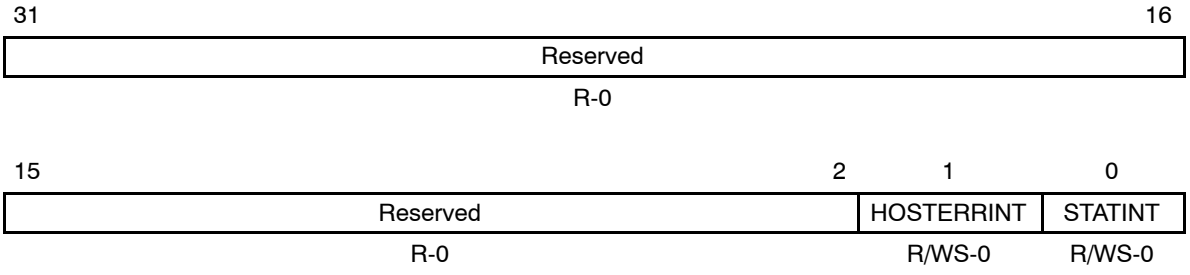
Bit	*field*†	*sym_val*†	Value	Description
31−2	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	HOSTERRINT			Host error interrupt bit. Masked interrupt read.
0	STATINT			Statistics interrupt bit. Masked interrupt read.

† For CSL implementation, use the notation EMAC_MACINTSTATMASKED_*field_symval*

5.2.28 MAC Interrupt Mask Set Register (MACINTMASKSET)

The MAC interrupt mask set register (MACINTMASKSET) is shown in Figure 5−31 and described in Table 5−33.

Figure 5−31. MAC Interrupt Mask Set Register (MACINTMASKSET)



Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; *-n* = value after reset

Table 5−33. MAC Interrupt Mask Set Register (MACINTMASKSET) Field Descriptions

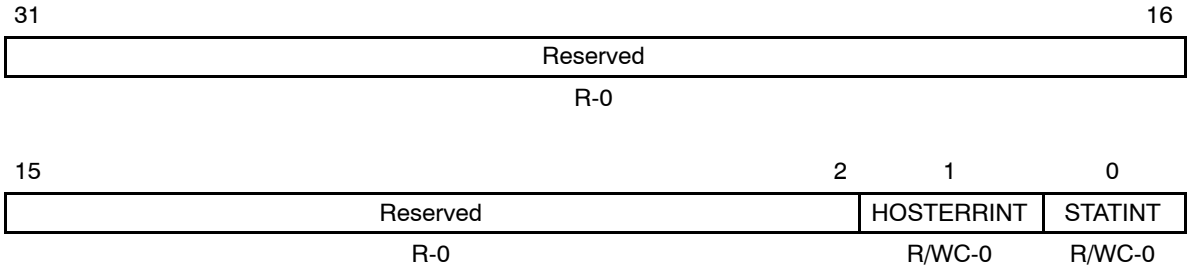
Bit	*field*†	*sym_val*†	Value	Description
31−2	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	HOSTERRINT			Host error interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Host error interrupt is enabled.
0	STATINT			Statistics interrupt mask set bit. Write 1 to enable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Statistics interrupt is enabled.

† For CSL implementation, use the notation EMAC_MACINTMASKSET_*field_symval*

5.2.29 MAC Interrupt Mask Clear Register (MACINTMASKCLEAR)

The MAC interrupt mask clear register (MACINTMASKCLEAR) is shown in Figure 5−32 and described in Table 5−34.

Figure 5−32. MAC Interrupt Mask Clear Register (MACINTMASKCLEAR)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; *-n* = value after reset

Table 5−34. MAC Interrupt Mask Clear Register (MACINTMASKCLEAR) Field Descriptions

Bit	*field*†	*sym_val*†	Value	Description
31−2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	HOSTERRINT			Host error interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Host error interrupt is disabled.
0	STATINT			Statistics interrupt mask clear bit. Write 1 to disable interrupt, a write of 0 has no effect.
			0	No effect.
			1	Statistics interrupt is disabled.

† For CSL implementation, use the notation EMAC_MACINTMASKCLEAR_*field_symval*

5.2.30 MAC Address Channel 0–7 Lower Byte Registers (MACADDRL*n*)

The MAC address channel *n* lower byte register (MACADDRL*n*) is shown in Figure 5–33 and described in Table 5–35.

In order to facilitate changing the MACADDR values while the device is operating, a channel is disabled when MACADDRL*n* is written and enabled when MACADDRH is written (provided that the unicast, broadcast, or multicast enable is set). MACADDRH should be written last.

Figure 5–33. MAC Address Channel n Lower Byte Register (MACADDRLn)

31	8	7	0
Reserved		MACADDR8 [7–0]	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5–35. MAC Address Channel n Lower Byte Register (MACADDRLn) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7–0	MACADDR8		0–FFh	Sixth byte (bits 0–7) of MAC specific address.

† For CSL implementation, use the notation EMAC_MACADDRL*n*_MACADDR8_*symval*

5.2.31 MAC Address Middle Byte Register (MACADDRM)

The MAC address middle byte register (MACADDRM) is shown in Figure 5–34 and described in Table 5–36.

Figure 5–34. MAC Address Middle Byte Register (MACADDRM)

31	8	7	0
Reserved		MACADDR8 [15–8]	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5–36. MAC Address Middle Byte Register (MACADDRM) Field Descriptions

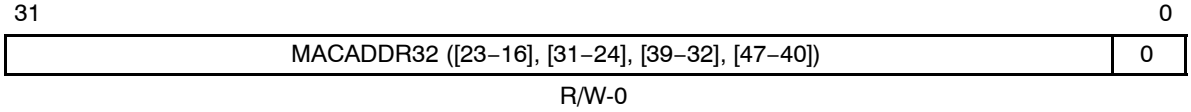
Bit	Field	*sym_val*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7–0	MACADDR8		0–FFh	Fifth byte (bits 8–15) of MAC specific address.

† For CSL implementation, use the notation EMAC_MACADDRM_MACADDR8_*symval*

5.2.32 MAC Address High Bytes Register (MACADDRH)

The MAC address high bytes register (MACADDRH) is shown in Figure 5−35 and described in Table 5−37.

Figure 5−35. MAC Address High Bytes Register (MACADDRH)



Legend: R/W = Read/Write; *-n* = value after reset

Table 5−37. MAC Address High Bytes Register (MACADDRH) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31−0	MACADDR32		0−FFFF FFFEh	First 32 bits (bits 16−47) of MAC specific address. Bit 0 is considered the group/specific bit and is hard-wired to 0, writes have no effect. Bit 0 corresponds to the group/specific address bit. Specific addresses always have this bit cleared to 0.

† For CSL implementation, use the notation EMAC_MACADDRH_MACADDR32_*symval*

5.2.33 MAC Address Hash 1 Register (MACHASH1)

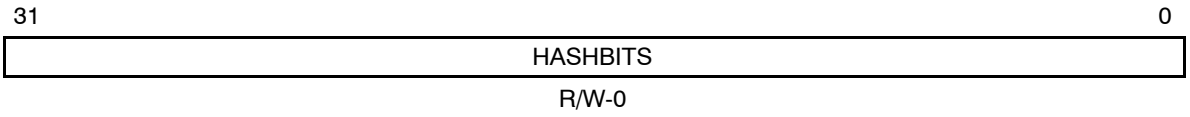
The MAC hash registers allow group addressed frames to be accepted on the basis of a hash function of the address. The hash function creates a 6-bit data value (Hash_fun) from the 48-bit destination address (DA) as follows:

```
Hash_fun(0)=DA(0) XOR DA(6) XOR DA(12) XOR DA(18) XOR DA(24) XOR DA(30) XOR DA(36) XOR DA(42);
Hash_fun(1)=DA(1) XOR DA(7) XOR DA(13) XOR DA(19) XOR DA(25) XOR DA(31) XOR DA(37) XOR DA(43);
Hash_fun(2)=DA(2) XOR DA(8) XOR DA(14) XOR DA(20) XOR DA(26) XOR DA(32) XOR DA(38) XOR DA(44);
Hash_fun(3)=DA(3) XOR DA(9) XOR DA(15) XOR DA(21) XOR DA(27) XOR DA(33) XOR DA(39) XOR DA(45);
Hash_fun(4)=DA(4) XOR DA(10) XOR DA(16) XOR DA(22) XOR DA(28) XOR DA(34) XOR DA(40) XOR DA(46);
Hash_fun(5)=DA(5) XOR DA(11) XOR DA(17) XOR DA(23) XOR DA(29) XOR DA(35) XOR DA(41) XOR DA(47);
```

This function is used as an offset into a 64-bit hash table stored in MACHASH1 and MACHASH2 that indicates whether a particular address should be accepted or not.

The MAC address hash 1 register (MACHASH1) is shown in Figure 5−36 and described in Table 5−38.

Figure 5−36. MAC Address Hash 1 Register (MACHASH1)



Legend: R/W = Read/Write; *-n* = value after reset

Table 5−38. MAC Address Hash 1 Register (MACHASH1) Field Descriptions

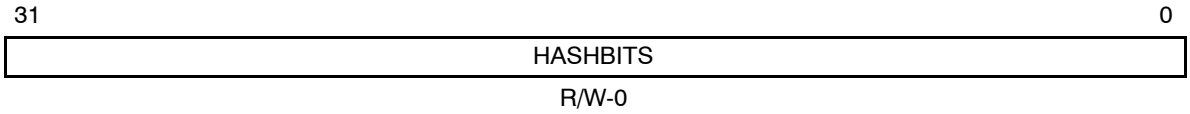
Bit	Field	*sym_val*†	Value	Description
31−0	HASHBITS		0−FFFF FFFFh	Least-significant 32 bits of the hash table corresponding to hash values 0 to 31. If a hash table bit is set, then a group address that hashes to that bit index is accepted.

† For CSL implementation, use the notation EMAC_MACHASH1_HASHBITS_*symval*

5.2.34 MAC Address Hash 2 Register (MACHASH2)

The MAC address hash 2 register (MACHASH2) is shown in Figure 5−37 and described in Table 5−39.

Figure 5−37. MAC Address Hash 2 Register (MACHASH2)



Legend: R/W = Read/Write; *-n* = value after reset

Table 5−39. MAC Address Hash 2 Register (MACHASH2) Field Descriptions

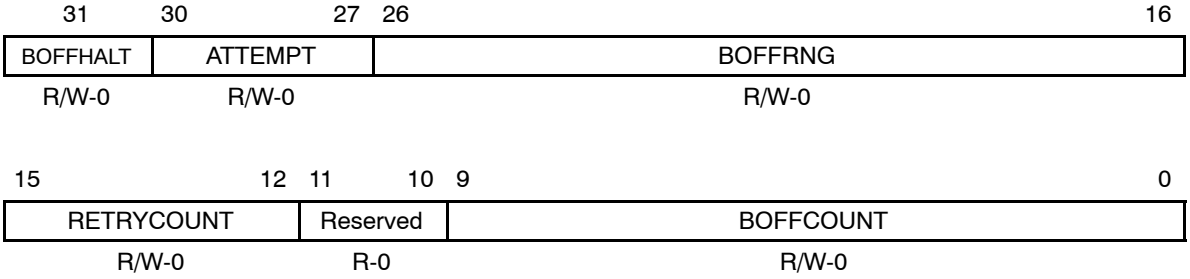
Bit	Field	*sym_val*†	Value	Description
31−0	HASHBITS		0−FFFF FFFFh	Most-significant 32 bits of the hash table corresponding to hash values 32 to 63. If a hash table bit is set, then a group address that hashes to that bit index is accepted.

† For CSL implementation, use the notation EMAC_MACHASH2_HASHBITS_*symval*

5.2.35 Backoff Test Register (BOFFTEST)

The backoff test register (BOFFTEST) is shown in Figure 5−38 and described in Table 5−40.

Figure 5−38. Backoff Test Register (BOFFTEST)



Legend: R = Read only; R/W = Read/Write; *-n* = value after reset

Table 5−40. Backoff Test Register (BOFFTEST) Field Descriptions

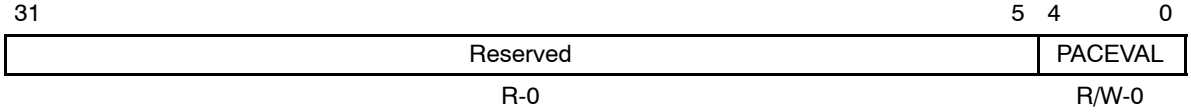
Field	*field*†	*sym_val*†	Value	Function
31	BOFFHALT			
30−27	ATTEMPT		0−Fh	Initial collision attempt count bits is the number of collisions the current frame has experienced.
26−16	BOFFRNG		0−7FFh	Backoff random number generator bits allow the backoff random number generator to be read (or written in test mode only). This field can be written only when the MTEST bit in MACCONTROL has previously been set. Reading this field returns the generator's current value. The value is reset to 0 and begins counting on the clock after the deassertion of reset.
15−12	RETRYCOUNT		0−Fh	
11−10	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
9−0	BOFFCOUNT		0−3FFh	Backoff current count bits allow the current value of the backoff counter to be observed for test purposes. This field is loaded automatically according to the backoff algorithm and is decremented by 1 for each slot time after the collision.

† For CSL implementation, use the notation EMAC_BOFFTEST_*field_symval*

5.2.36 Transmit Pacing Test Register (TPACETEST)

The transmit pacing test register (TPACETEST) is shown in Figure 5−39 and described in Table 5−41.

Figure 5−39. Transmit Pacing Test Register (TPACETEST)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−41. Transmit Pacing Test Register (TPACETEST) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31−5	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
4−0	PACEVAL		0−1Fh	Pacing register current value. A nonzero value in this field indicates that transmit pacing is active. A transmit frame collision or deferral causes PACEVAL to be loaded with 1Fh (31), good frame transmissions (with no collisions or deferrals) cause PACEVAL to be decremented down to 0. When PACEVAL is nonzero, the transmitter delays four IPGs between new frame transmissions after each successfully transmitted frame that had no deferrals or collisions. If a transmit frame is deferred or suffers a collision, the IPG time is not stretched to four times the normal value. Transmit pacing helps reduce capture effects, which improves overall network bandwidth.

† For CSL implementation, use the notation EMAC_TPACETEST_PACEVAL_*symval*

5.2.37 Receive Pause Timer Register (RXPAUSE)

The receive pause timer register (RXPAUSE) is shown in Figure 5−40 and described in Table 5−42.

Figure 5−40. Receive Pause Timer Register (RXPAUSE)

31	16	15	0
Reserved		PAUSETIMER	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−42. Receive Pause Timer Register (RXPAUSE) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31−16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−0	PAUSETIMER		0−FFFFh	Pause timer value bits. This field allows the contents of the receive pause timer to be observed (and written in test mode). The receive pause timer is loaded with FF00h when the EMAC sends an outgoing pause frame (with pause time of FFFFh). The receive pause timer is decremented at slot time intervals. If the receive pause timer decrements to 0, then another outgoing pause frame is sent and the load/decrement process is repeated.

† For CSL implementation, use the notation EMAC_RXPAUSE_PAUSETIMER_*symval*

5.2.38 Transmit Pause Timer Register (TXPAUSE)

The transmit pause timer register (TXPAUSE) is shown in Figure 5−41 and described in Table 5−43.

Figure 5−41. Transmit Pause Timer Register (TXPAUSE)

31	16	15	0
Reserved		PAUSETIMER	
R-0		R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−43. Transmit Pause Timer Register (TXPAUSE) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31−16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−0	PAUSETIMER		0−FFFFh	Pause timer value bits − This field allows the contents of the transmit pause timer to be observed (and written in test mode). The transmit pause timer is loaded by a received (incoming) pause frame, and then decremented at slot time intervals down to 0 at which time EMAC transmit frames are again enabled.

† For CSL implementation, use the notation EMAC_TXPAUSE_PAUSETIMER_*symval*

5.2.39 Transmit Channel 0–7 DMA Head Descriptor Pointer Registers (TX*n*HDP)

The transmit channel *n* DMA head descriptor pointer register (TX*n*HDP) is shown in Figure 5–42 and described in Table 5–44.

Figure 5–42. Transmit Channel n DMA Head Descriptor Pointer Register (TXnHDP)

31	0
DESCPTR	
R/W-0	

Legend: R/W = Read/Write; -*n* = value after reset

Table 5–44. Transmit Channel n DMA Head Descriptor Pointer Register (TXnHDP) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31–0	DESCPTR		0–FFFF FFFFh	Descriptor pointer bits. Writing a transmit DMA buffer descriptor address to a head pointer location initiates transmit DMA operations in the queue for the selected channel. Writing to these locations when they are nonzero is an error (except at reset). Host software must initialize these locations to zero on reset.

† For CSL implementation, use the notation EMAC_TX*n*HDP_DESCPTR_*symval*

5.2.40 Receive Channel 0–7 DMA Head Descriptor Pointer Registers (RX*n*HDP)

The receive channel *n* DMA head descriptor pointer register (RX*n*HDP) is shown in Figure 5–43 and described in Table 5–45.

Figure 5–43. Receive Channel n DMA Head Descriptor Pointer Register (RXnHDP)

31	0
DESCPTR	
R/W-0	

Legend: R/W = Read/Write; -*n* = value after reset

Table 5–45. Receive Channel n DMA Head Descriptor Pointer Register (RXnHDP) Field Descriptions

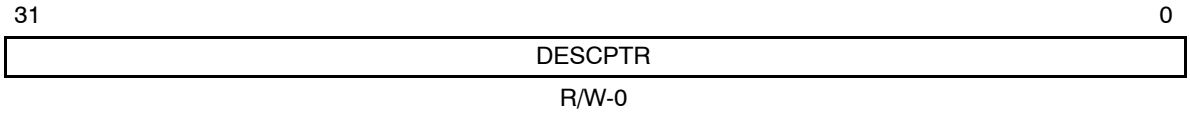
Bit	Field	*sym_val*†	Value	Description
31–0	DESCPTR		0–FFFF FFFFh	Descriptor pointer bits. Writing a receive DMA buffer descriptor address to this location allows receive DMA operations in the selected channel when a channel frame is received. Writing to these locations when they are nonzero is an error (except at reset). Host software must initialize these locations to zero on reset.

† For CSL implementation, use the notation EMAC_RX*n*HDP_DESCPTR_*symval*

5.2.41 Transmit Channel 0–7 Interrupt Acknowledge Registers (TX*n*INTACK)

The transmit channel *n* interrupt acknowledge register (TX*n*INTACK) is shown in Figure 5–44 and described in Table 5–46.

Figure 5–44. Transmit Channel n Interrupt Acknowledge Register (TXnINTACK)



Legend: R/W = Read/Write; -*n* = value after reset

Table 5–46. Transmit Channel n Interrupt Acknowledge Register (TXnINTACK) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31–0	DESCPTR		0–FFFF FFFFh	Transmit host interrupt acknowledge register bits. This register is written by the host with the buffer descriptor address for the last buffer processed by the host during interrupt processing. The EMAC uses the value written to determine if the interrupt should be deasserted.

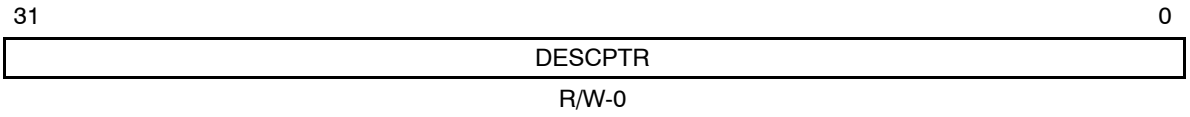
† For CSL implementation, use the notation EMAC_TX*n*INTACK_DESCPTR_*symval*

5.2.42 Receive Channel 0–7 Interrupt Acknowledge Registers (RX*n*INTACK)

The receive channel *n* interrupt acknowledge registers (RX*n*INTACK) is shown in Figure 5–45 and described in Table 5–47.

The value read is the interrupt acknowledge value that was written by the EMAC DMA controller. The value written to RX*n*INTACK by the host is compared with the value that the EMAC wrote to determine if the interrupt should remain asserted. The value written is not actually stored in this location. The interrupt is deasserted, if the two values are equal.

Figure 5–45. Receive Channel n Interrupt Acknowledge Register (RXnINTACK)



Legend: R/W = Read/Write; -*n* = value after reset

Table 5–47. Receive Channel n Interrupt Acknowledge Register (RXnINTACK) Field Descriptions

Bit	Field	*sym_val*†	Value	Description
31–0	DESCPTR		0–FFFF FFFFh	Receive host interrupt acknowledge register bits. This register is written by the host with the buffer descriptor address for the last buffer processed by the host during interrupt processing. The EMAC uses the value written to determine if the interrupt should be deasserted.

† For CSL implementation, use the notation EMAC_RX*n*INTACK_DESCPTR_*symval*

5.2.43 Network Statistics Registers

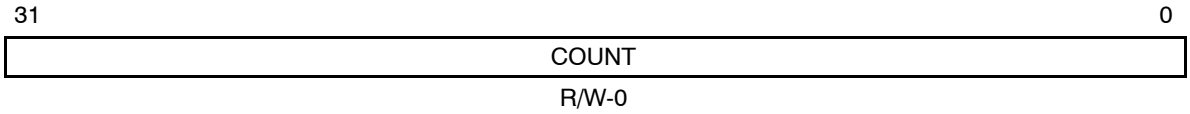
The EMAC has a set of statistics that record events associated with frame traffic. The statistics values are cleared to zero, 38 clocks after the rising edge of reset. When the MIIEN bit in the MACCONTROL register is set, all statistics registers are write-to-decrement. The value written is subtracted from the register value with the result stored in the register. If a value greater than the statistics value is written, then zero is written to the register (writing FFFF FFFFh clears a statistics location). When the MIIEN bit is cleared, all statistics registers are read/write (normal write direct, so writing 0000 0000h clears a statistics location). All write accesses must be 32-bit accesses.

The statistics interrupt (STATPEND) is issued, if enabled, when any statistics value is greater than or equal to 8000 0000h. The statistics interrupt is removed by writing to decrement any statistics value greater than 8000 0000h. The statistics are mapped into internal memory space and are 32-bits wide. All statistics rollover from FFFF FFFFh to 0000 0000h.

The statistics registers are 32-bit registers as shown in Figure 5−46.

For CSL implementation, use: EMAC_*register name*_COUNT_*symval*

Figure 5−46. Statistics Register



Legend: R/W = Read/Write; -*n* = value after reset

5.2.43.1 Good Receive Frames Register (RXGOODFRAMES)

The total number of good frames received on the EMAC. A good frame is defined as having all of the following:

- ❑ Any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❑ Was of length 64 to RXMAXLEN bytes inclusive
- ❑ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.2 Broadcast Receive Frames Register (RXBCASTFRAMES)

The total number of good broadcast frames received on the EMAC. A good broadcast frame is defined as having all of the following:

- ❑ Any data or MAC control frame that was destined for address FF−FF−FF−FF−FF−FFh only
- ❑ Was of length 64 to RXMAXLEN bytes inclusive
- ❑ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.3 Multicast Receive Frames Register (RXMCASTFRAMES)

The total number of good multicast frames received on the EMAC. A good multicast frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any multicast address other than FF−FF−FF−FF−FF−FFh
- ❏ Was of length 64 to RXMAXLEN bytes inclusive
- ❏ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.4 Pause Receive Frames Register (RXPAUSEFRAMES)

The total number of IEEE 802.3X pause frames received by the EMAC (whether acted upon or not). A pause frame is defined as having all of the following:

- ❏ Contained any unicast, broadcast, or multicast address
- ❏ Contained the length/type field value 88.08h and the opcode 0001h
- ❏ Was of length 64 to RXMAXLEN bytes inclusive
- ❏ Had no CRC error, alignment error, or code error
- ❏ Pause-frames had been enabled on the EMAC (TXFLOWEN bit is set in MACCONTROL).

The EMAC could have been in either half-duplex or full-duplex mode.

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.5 Receive CRC Errors Register (RXCRCERRORS)

The total number of frames received on the EMAC that experienced a CRC error. A frame is defined as having all of the following:

- ❏ Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was of length 64 to RXMAXLEN bytes inclusive
- ❏ Had no alignment or code error
- ❏ Had a CRC error. A CRC error is defined as having all of the following:
  - ■ A frame containing an even number of nibbles
  - ■ Fails the frame check sequence test

See section 2.3.5 for definitions of alignment and code errors. Overruns have no effect on this statistic.

5.2.43.6 Receive Alignment/Code Errors Register (RXALIGNCODEERRORS)

The total number of frames received on the EMAC that experienced an alignment error or code error. Such a frame is defined as having all of the following:

- Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- Was of length 64 to RXMAXLEN bytes inclusive
- Had either an alignment error or a code error
  - An alignment error is defined as having all of the following:
    - A frame containing an odd number of nibbles
    - Fails the frame check sequence test, if the final nibble is ignored
  - A code error is defined as:
    - A frame that has been discarded because the EMACs MRXER pin is driven with a one for at least one bit-time's duration at any point during the frame's reception.

Overruns have no effect on this statistic.

RFC 1757 etherStatsCRCAlignErrors Ref. 1.5 can be calculated by summing receive alignment errors, receive code errors, and receive CRC errors.

5.2.43.7 Receive Oversized Frames Register (RXOVERSIZED)

The total number of oversized frames received on the EMAC. An oversized frame is defined as having all of the following:

- Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- Was greater than RXMAXLEN in bytes
- Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.8 Receive Jabber Frames Register (RXJABBER)

The total number of jabber frames received on the EMAC. A jabber frame is defined as having all of the following:

- Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- Was greater than RXMAXLEN bytes long
- Had a CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.9 Receive Undersized Frames Register (RXUNDERSIZED)

The total number of undersized frames received on the EMAC. An undersized frame is defined as having all of the following:

- ❏ Was any data frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was less than 64 bytes long
- ❏ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.10 Receive Frame Fragments Register (RXFRAGMENTS)

The total number of frame fragments received on the EMAC. A frame fragment is defined as having all of the following:

- ❏ Any data frame (address matching does not matter)
- ❏ Was less than 64 bytes long
- ❏ Had a CRC error, alignment error, or code error
- ❏ Was not the result of a collision caused by half duplex, collision based flow control

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.11 Filtered Receive Frames Register (RXFILTERED)

The total number of frames received on the EMAC that the EMAC address matching process indicated should be discarded. Such a frame is defined as having all of the following:

- ❏ Was any data frame (not MAC control frame) destined for any unicast, broadcast, or multicast address
- ❏ Did not experience any CRC error, alignment error, code error
- ❏ The address matching process decided that the frame should be discarded (filtered) because it did not match the unicast, broadcast, or multicast address, and it did not match due to promiscuous mode.

To determine the number of receive frames discarded by the EMAC for any reason, sum the following statistics (promiscuous mode disabled):

- ❏ Receive fragments
- ❏ Receive undersized frames
- ❏ Receive CRC errors
- ❏ Receive alignment/code errors
- ❏ Receive jabbers
- ❏ Receive overruns
- ❏ Receive filtered frames

This may not be an exact count because the receive overruns statistic is independent of the other statistics, so if an overrun occurs at the same time as one of the other discard reasons, then the above sum double-counts that frame.

5.2.43.12 Receive QOS Filtered Frames Register (RXQOSFILTERED)

The total number of frames received on the EMAC that were filtered due to receive quality of service (QOS) filtering. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ The frame destination channel flow control threshold register (RX*n*FLOWTHRESH) value was greater than or equal to the channel's corresponding free buffer register (RX*n*FREEBUFFER) value
- ❏ Was of length 64 to RXMAXLEN
- ❏ RXQOSEN bit is set in RXMBPENABLE
- ❏ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.13 Receive Octet Frames Register (RXOCTETS)

The total number of bytes in all good frames received on the EMAC. A good frame is defined as having all of the following:

- ❏ Any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was of length 64 to RXMAXLEN bytes inclusive
- ❏ Had no CRC error, alignment error, or code error

See section 2.3.5 for definitions of alignment, code, and CRC errors. Overruns have no effect on this statistic.

5.2.43.14 Receive Start of Frame Overruns Register (RXSOFOVERRUNS)

The total number of frames received on the EMAC that had either a FIFO or DMA start of frame (SOF) overrun. A SOF overrun frame is defined as having all of the following:

- ❏ Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was any length (including <64 bytes and > RXMAXLEN bytes)
- ❏ The EMAC was unable to receive it because it did not have the resources to receive it (cell FIFO full or no DMA buffer available at the start of the frame).

CRC errors, alignment errors, and code errors have no effect on this statistic.

5.2.43.15 Receive Middle of Frame Overruns Register (RXMOFOVERRUNS)

The total number of frames received on the EMAC that had either a FIFO or DMA middle of frame (MOF) overrun. A MOF overrun frame is defined as having all of the following:

- ❏ Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was any length (including <64 bytes and > RXMAXLEN bytes)
- ❏ The EMAC was unable to receive it because it did not have the resources to receive it (cell FIFO full or no DMA buffer available after the frame was successfully started — no SOF overrun).

CRC errors, alignment errors, and code errors have no effect on this statistic.

5.2.43.16 Receive DMA Overruns Register (RXDMAOVERRUNS)

The total number of frames received on the EMAC that had either a DMA start of frame (SOF) overrun or a DMA middle of frame (MOF) overrun. A receive DMA overrun frame is defined as having all of the following:

- ❏ Was any data or MAC control frame that matched a unicast, broadcast, or multicast address, or matched due to promiscuous mode
- ❏ Was any length (including <64 bytes and > RXMAXLEN bytes)
- ❏ The EMAC was unable to receive it because it did not have the DMA buffer resources to receive it (zero head descriptor pointer at the start or during the middle of the frame reception).

CRC errors, alignment errors, and code errors have no effect on this statistic.

5.2.43.17 Good Transmit Frames Register (TXGOODFRAMES)

The total number of good frames transmitted on the EMAC. A good frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Was any length
- ❏ Had no late or excessive collisions, no carrier loss, and no underrun

5.2.43.18 Broadcast Transmit Frames Register (TXBCASTFRAMES)

The total number of good broadcast frames transmitted on the EMAC. A good broadcast frame is defined as having all of the following:

- ❏ Any data or MAC control frame destined for address FF–FF–FF–FF–FF–FFh only
- ❏ Was of any length
- ❏ Had no late or excessive collisions, no carrier loss, and no underrun

5.2.43.19 Multicast Transmit Frames Register (TXMCASTFRAMES)

The total number of good multicast frames transmitted on the EMAC. A good multicast frame is defined as having all of the following:

- ❏ Any data or MAC control frame destined for any multicast address other than FF–FF–FF–FF–FF–FFh
- ❏ Was of any length
- ❏ Had no late or excessive collisions, no carrier loss, and no underrun

5.2.43.20 Pause Transmit Frames Register (TXPAUSEFRAMES)

The total number of IEEE 802.3X pause frames transmitted by the EMAC. Pause frames cannot underrun or contain a CRC error because they are created in the transmitting MAC, so these error conditions have no effect on this statistic. Pause frames sent by software are not included in this count. Since pause frames are only transmitted in full-duplex mode, carrier loss and collisions have no effect on this statistic.

Transmitted pause frames are always 64-byte multicast frames so appear in the multicast transmit frames register and 64 octect frames register statistics.

5.2.43.21 Deferred Transmit Frames Register (TXDEFERRED)

The total number of frames transmitted on the EMAC that first experienced deferment. Such a frame is defined as having all of the following:

- ❏ Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- ❏ Was any size
- ❏ Had no carrier loss and no underrun
- ❏ Experienced no collisions before being successfully transmitted
- ❏ Found the medium busy when transmission was first attempted, so had to wait.

CRC errors have no effect on this statistic.

See RFC1623 Ref. 2.6 dot3StatsDefferredTransmissions.

5.2.43.22 Collision Register (TXCOLLISION)

The total number of times that the EMAC experienced a collision. Collisions occur under two circumstances:

- ❏ When a transmit data or MAC control frame has all of the following:
  - ■ Was destined for any unicast, broadcast, or multicast address
  - ■ Was any size
  - ■ Had no carrier loss and no underrun
  - ■ Experienced a collision. A jam sequence is sent for every nonlate collision, so this statistic increments on each occasion if a frame experiences multiple collisions (and increments on late collisions).

  CRC errors have no effect on this statistic.
- ❏ When the EMAC is in half-duplex mode, flow control is active, and a frame reception begins.

5.2.43.23 Single Collision Transmit Frames Register (TXSINGLECOLL)

The total number of frames transmitted on the EMAC that experienced exactly one collision. Such a frame is defined as having all of the following:

- ❏ Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- ❏ Was any size
- ❏ Had no carrier loss and no underrun
- ❏ Experienced one collision before successful transmission. The collision was not late.

CRC errors have no effect on this statistic.

5.2.43.24 Multiple Collision Transmit Frames Register (TXMULTICOLL)

The total number of frames transmitted on the EMAC that experienced multiple collisions. Such a frame is defined as having all of the following:

- ❏ Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- ❏ Was any size
- ❏ Had no carrier loss and no underrun
- ❏ Experienced 2 to 15 collisions before being successfully transmitted. None of the collisions were late.

CRC errors have no effect on this statistic.

5.2.43.25 Excessive Collisions Register (TXEXCESSIVECOLL)

The total number of frames when transmission was abandoned due to excessive collisions. Such a frame is defined as having all of the following:

- ❏ Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- ❏ Was any size
- ❏ Had no carrier loss and no underrun
- ❏ Experienced 16 collisions before abandoning all attempts at transmitting the frame. None of the collisions were late.

CRC errors have no effect on this statistic.

5.2.43.26 Late Collisions Register (TXLATECOLL)

The total number of frames when transmission was abandoned due to a late collision. Such a frame is defined as having all of the following:

- Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- Was any size
- Had no carrier loss and no underrun
- Experienced a collision later than 512 bit-times into the transmission. There may have been up to 15 previous (non-late) collisions that had previously required the transmission to be reattempted. The late collisions statistic dominates over the single, multiple, and excessive collisions statistics. If a late collision occurs, the frame is not counted in any of these other three statistics.

CRC errors, carrier loss, and underrun have no effect on this statistic.

5.2.43.27 Transmit Underrun Register (TXUNDERRUN)

The number of frames sent by the EMAC that experienced FIFO underrun. Late collisions, CRC errors, carrier loss, and underrun have no effect on this statistic.

5.2.43.28 Transmit Carrier Sense Errors Register (TXCARRIERSLOSS)

The total number of frames on the EMAC that experienced carrier loss. Such a frame is defined as having all of the following:

- Was any data or MAC control frame destined for any unicast, broadcast, or multicast address
- Was any size
- The carrier sense condition was lost or never asserted when transmitting the frame (the frame is not retransmitted)

CRC errors and underrun have no effect on this statistic.

5.2.43.29 Transmit Octet Frames Register (TXOCTETS)

The total number of bytes in all good frames transmitted on the EMAC. A good frame is defined as having all of the following:

- Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- Was any length
- Had no late or excessive collisions, no carrier loss, and no underrun

5.2.43.30 Transmit and Receive 64 Octet Frames Register (FRAME64)

The total number of 64-byte frames received and transmitted on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was exactly 64-bytes long. (If the frame was being transmitted and experienced carrier loss that resulted in a frame of this size being transmitted, then the frame is recorded in this statistic).

CRC errors, alignment/code errors, and overruns do not affect the recording of frames in this statistic.

5.2.43.31 Transmit and Receive 65 to 127 Octet Frames Register (FRAME65T127)

The total number of 65- to 127-byte frames received and transmitted on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was 65- to 127-bytes long

CRC errors, alignment/code errors, underruns, and overruns do not affect the recording of frames in this statistic.

5.2.43.32 Transmit and Receive 128 to 255 Octet Frames Register (FRAME128T255)

The total number of 128- to 255-byte frames received and transmitted on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was 128- to 255-bytes long

CRC errors, alignment/code errors, underruns, and overruns do not affect the recording of frames in this statistic.

For receive reference only, see RFC1757 Ref. 1.13 etherStatsPkts128to255Octets.

5.2.43.33 Transmit and Receive 256 to 511 Octet Frames Register (FRAME256T511)

The total number of 256- to 511-byte frames received and transmitted on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was 256- to 511-bytes long

CRC errors, alignment/code errors, underruns, and overruns do not affect the recording of frames in this statistic.

5.2.43.34 Transmit and Receive 512 to 1023 Octet Frames Register (FRAME512T1023)

The total number of 512- to 1023-byte frames received and transmitted on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was 512- to 1023-bytes long

CRC errors, alignment/code errors, and overruns do not affect the recording of frames in this statistic.

5.2.43.35 Transmit and Receive 1024 or Above Octet Frames Register (FRAME1024TUP)

The total number of 1024- to RXMAXLEN-byte frames for receive or 1024-byte frames and above for transmit on the EMAC. Such a frame is defined as having all of the following:

- ❏ Any data or MAC control frame that was destined for any unicast, broadcast, or multicast address
- ❏ Did not experience late collisions, excessive collisions, underrun, or carrier sense error
- ❏ Was 1024- to RXMAXLEN-bytes long for receive, or any size for transmit

CRC errors, alignment/code errors, underruns, and overruns do not affect the recording of frames in this statistic.

5.2.43.36 Network Octet Frames Register (NETOCTETS)

The total number of bytes of frame data received and transmitted on the EMAC. Each frame counted has all of the following:

- ❏ Was any data or MAC control frame destined for any unicast, broadcast, or multicast address (address match does not matter)
- ❏ Was of any size (including <64 byte and > RXMAXLEN byte frames)

Also counted in this statistic is:

- ❏ Every byte transmitted before a carrier-loss was experienced
- ❏ Every byte transmitted before each collision was experienced, (multiple retries are counted each time)
- ❏ Every byte received if the EMAC is in half-duplex mode until a jam sequence was transmitted to initiate flow control. (The jam sequence is not counted to prevent double-counting).

Error conditions such as alignment errors, CRC errors, code errors, overruns, and underruns do not affect the recording of bytes in this statistic. The objective of this statistic is to give a reasonable indication of Ethernet utilization.

5.3 MDIO Module Registers

Control registers for the MDIO module are summarized in Table 5−48. See the device-specific datasheet for the memory address of these registers. Please see the device-specific datasheet for a listing of supported registers.

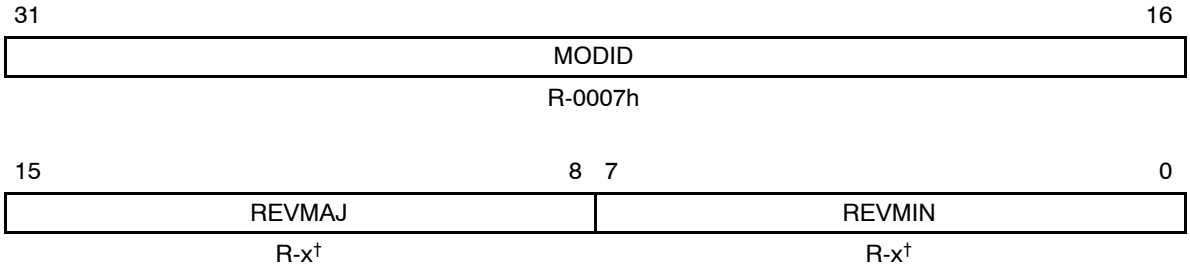
Table 5−48. MDIO Module Registers

Acronym	Register Name	Section
VERSION	MDIO Version Register	5.3.1
CONTROL	MDIO Control Register	5.3.2
ALIVE	MDIO PHY Alive Indication Register	5.3.3
LINK	MDIO PHY Link Status Register	5.3.4
LINKINTRAW	MDIO Link Status Change Interrupt Register	5.3.5
LINKINTMASKED	MDIO Link Status Change Interrupt (Masked) Register	5.3.6
USERINTRAW	MDIO User Command Complete Interrupt Register	5.3.7
USERINTMASKED	MDIO User Command Complete Interrupt (Masked) Register	5.3.8
USERINTMASKSET	MDIO User Command Complete Interrupt Mask Set Register	5.3.9
USERINTMASKCLEAR	MDIO User Command Complete Interrupt Mask Clear Register	5.3.10
USERACCESS0	MDIO User Access Register 0	5.3.11
USERACCESS1	MDIO User Access Register 1	5.3.12
USERPHYSEL0	MDIO User PHY Select Register 0	5.3.13
USERPHYSEL1	MDIO User PHY Select Register 1	5.3.14

5.3.1 MDIO Version Register (VERSION)

The MDIO version register (VERSION) is shown in Figure 5−47 and described in Table 5−49.

Figure 5−47. MDIO Version Register (VERSION)



Legend: R = Read only; -*n* = value after reset
† See the device-specific datasheet for the default value of this field.

Table 5−49. MDIO Version Register (VERSION) Field Descriptions

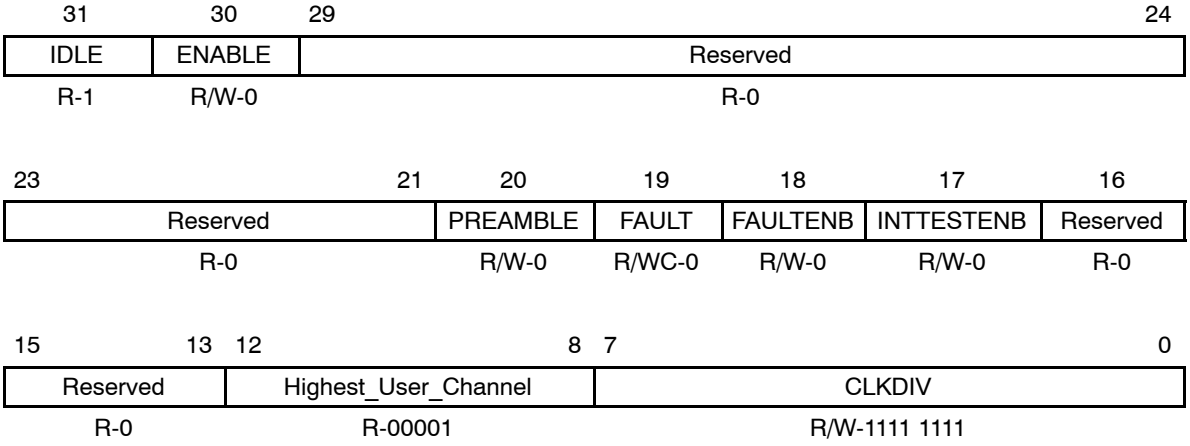
Bit	*field*†	*symval*†	Value	Description
31−16	MODID			Identifies type of peripheral.
			7h	MDIO
15−8	REVMAJ			Identifies major revision of peripheral.
			x	See the device-specific datasheet for the value.
7−0	REVMIN			Identifies minor revision of peripheral.
			x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation MDIO_VERSION_*field_symval*

5.3.2 MDIO Control Register (CONTROL)

The MDIO control register (CONTROL) is shown in Figure 5−48 and described in Table 5−50.

Figure 5−48. MDIO Control Register (CONTROL)



Legend: R = Read only; WC = Write to clear; R/W = Read/Write; *-n* = value after reset

Table 5−50. MDIO Control Register (CONTROL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	IDLE			MDIO state machine IDLE status bit.
		NO	0	State machine is not in the idle state.
		YES	1	State machine is in the idle state.
30	ENABLE			MDIO state machine enable control bit. If the MDIO state machine is active at the time it is disabled, it completes the current operation before halting and setting the IDLE bit. If using byte access, the ENABLE bit has to be the last bit written in this register.
		NO	0	Disables the MDIO state machine.
		YES	1	Enables the MDIO state machine.
29−21	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation MDIO_CONTROL_*field_symval*

Table 5−50. MDIO Control Register (CONTROL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
20	PREAMBLE			MDIO frame preamble disable bit.
		ENABLED	0	Standard MDIO preamble is used.
		DISABLED	1	Disables this device from sending MDIO frame preambles.
19	FAULT			Fault indicator bit. Writing a 1 to this bit clears this bit.
		NO	0	No failure.
		YES	1	The MDIO pins fail to read back what the device is driving onto them indicating a physical layer fault. The MDIO state machine is reset.
18	FAULTENB			Fault detect enable bit.
		NO	0	Disables the physical layer fault detection.
		YES	1	Enables the physical layer fault detection.
17	INTTESTENB			Interrupt test enable bit.
		NO	0	Interrupt test bits are not set.
		YES	1	Enables the host to set the USERINTRAW, USERINTMASKED, LINKINTRAW, and LINKINTMASKED register bits for test purposes.
16−13	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
12−8	Highest_User _Channel		0−1Fh	Highest user-access channel bits specify the highest user-access channel that is available in the MDIO and is currently set to 1.
7−0	CLKDIV		0−FFh	Clock divider bits. Specifies the division ratio between peripheral clock and the frequency of MDCLK. MDCLK is disabled when CLKDIV is cleared to 0. MDCLK frequency = peripheral clock/(CLKDIV + 1).
			0	MDCLK is disabled.

† For CSL implementation, use the notation MDIO_CONTROL_*field_symval*

5.3.3 MDIO PHY Alive Indication Register (ALIVE)

The MDIO PHY alive indication register (ALIVE) is shown in Figure 5−49 and described in Table 5−51.

Figure 5−49. MDIO PHY Alive Indication Register (ALIVE)

31	0
ALIVE	
R/WC-0	

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−51. MDIO PHY Alive Indication Register (ALIVE) Field Descriptions

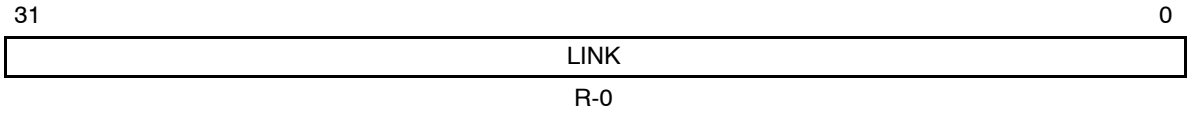
Bit	Field	*symval*†	Value	Description
31−0	ALIVE			MDIO ALIVE bits. Both user and polling accesses to a PHY cause the corresponding ALIVE bit to be updated. The ALIVE bits are only meant to give an indication of the presence or not of a PHY with the corresponding address. Writing a 1 to any bit clears that bit, writing a 0 has no effect.
			0	The PHY fails to acknowledge the access.
			1	The most recent access to the PHY with an address corresponding to the register bit number was acknowledged by the PHY.

† For CSL implementation, use the notation MDIO_ALIVE_ALIVE_*symval*

5.3.4 MDIO PHY Link Status Register (LINK)

The MDIO PHY link status register (LINK) is shown in Figure 5−50 and described in Table 5−52.

Figure 5−50. MDIO PHY Link Status Register (LINK)



Legend: R = Read only; -*n* = value after reset

Table 5−52. MDIO PHY Link Status Register (LINK) Field Descriptions

Bit	Field	*symval*†	Value	Description
31−0	LINK			MDIO link state bits. These bits are updated after a read of the PHY generic status register. Writes to these bits have no effect.
			0	The PHY indicates it does not have a link or fails to acknowledge the read transaction.
			1	The PHY with the corresponding address has a link and the PHY acknowledges the read transaction.

† For CSL implementation, use the notation MDIO_LINK_LINK_*symval*

5.3.5 MDIO Link Status Change Interrupt Register (LINKINTRAW)

The MDIO PHY link status change interrupt register (LINKINTRAW) is shown in Figure 5−51 and described in Table 5−53.

Figure 5−51. MDIO Link Status Change Interrupt Register (LINKINTRAW)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−53. MDIO Link Status Change Interrupt Register (LINKINTRAW) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO link change event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 1 (USERPHYSEL1).
0	MAC0			MDIO link change event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 0 (USERPHYSEL0).

† For CSL implementation, use the notation MDIO_LINKINTRAW_*field_symval*

5.3.6 MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED)

The MDIO PHY link status change interrupt (masked) register (LINKINTMASKED) is shown in Figure 5−52 and described in Table 5−54.

Figure 5−52. MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−54. MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO link change interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 1 (USERPHYSEL1) and the LINKINTENB bit in USERPHYSEL1 is set to 1.
0	MAC0			MDIO link change interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 0 (USERPHYSEL0) and the LINKINTENB bit in USERPHYSEL0 is set to 1.

† For CSL implementation, use the notation MDIO_LINKINTMASKED_*field_symval*

5.3.7 MDIO User Command Complete Interrupt Register (USERINTRAW)

The MDIO user command complete interrupt register (USERINTRAW) is shown in Figure 5−53 and described in Table 5−55.

Figure 5−53. MDIO User Command Complete Interrupt Register (USERINTRAW)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−55. MDIO User Command Complete Interrupt Register (USERINTRAW) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 1 (USERACCESS1) has completed.
0	MAC0			MDIO user command complete event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 0 (USERACCESS0) has completed.

† For CSL implementation, use the notation MDIO_USERINTRAW_*field_symval*

5.3.8 MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED)

The MDIO user command complete interrupt (masked) register (USERINTMASKED) is shown in Figure 5−54 and described in Table 5−56.

Figure 5−54. MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−56. MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 1 (USERACCESS1) has completed and the MAC1 bit in USERINTMASKSET is set to 1.
0	MAC0			MDIO user command complete interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 0 (USERACCESS0) has completed and the MAC0 bit in USERINTMASKSET is set to 1.

† For CSL implementation, use the notation MDIO_USERINTMASKED_*field_symval*

5.3.9 MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET)

The MDIO user command complete interrupt mask set register (USERINTMASKSET) is shown in Figure 5−55 and described in Table 5−57.

Figure 5−55. MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WS-0	R/WS-0

Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table 5−57. MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	−	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt mask set bit for MAC1 in USERINTMASKED. Writing a 1 sets the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are disabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are enabled.
0	MAC0			MDIO user command complete interrupt mask set bit for MAC0 in USERINTMASKED. Writing a 1 sets the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are disabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are enabled.

† For CSL implementation, use the notation MDIO_USERINTMASKSET_*field_symval*

5.3.10 MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR)

The MDIO user command complete interrupt mask clear register (USERINTMASKCLEAR) is shown in Figure 5−56 and described in Table 5−58.

Figure 5−56. MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table 5−58. MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt mask clear bit for MAC1 in USERINTMASKED. Writing a 1 clears the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are enabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are disabled.
0	MAC0			MDIO user command complete interrupt mask clear bit for MAC0 in USERINTMASKED. Writing a 1 clears the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are enabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are disabled.

† For CSL implementation, use the notation MDIO_USERINTMASKCLEAR_*field_symval*

5.3.11 MDIO User Access Register 0 (USERACCESS0)

The MDIO user access register 0 (USERACCESS0) is shown in Figure 5−57 and described in Table 5−59.

Figure 5−57. MDIO User Access Register 0 (USERACCESS0)

31	30	29	28 26	25 21	20 16
GO	WRITE	ACK	Reserved	REGADR	PHYADR
R/WS-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0

15 0
DATA
R/W-0

Legend: R = Read only; R/W = Read/Write; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table 5−59. MDIO User Access Register 0 (USERACCESS0) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	GO			GO bit is writable only if the MDIO state machine is enabled (ENABLE bit in MDIO control register is set to 1). If byte access is being used, the GO bit should be written last. Writing a 1 sets the bit and writing a 0 has no effect.
			0	No effect. The GO bit clears when the requested access has been completed.
			1	The MDIO state machine performs an MDIO access when it is convenient, this is not an instantaneous process. Any writes to USERACCESS0 are blocked.
30	WRITE			Write enable bit determines the MDIO transaction type.
			0	MDIO transaction is a register read.
			1	MDIO transaction is a register write.
29	ACK			Acknowledge bit determines if the PHY acknowledges the read transaction.
			0	No acknowledge.
			1	PHY acknowledges the read transaction.
28−26	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation MDIO_USERACCESS0_*field_symval*

Table 5−59. MDIO User Access Register 0 (USERACCESS0) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
25−21	REGADR		0−1Fh	Register address bits specify the PHY register to be accessed for this transaction.
20−16	PHYADR		0−1Fh	PHY address bits specify the PHY to be accessed for this transaction.
15−0	DATA		0−FFFFh	User data bits specify the data value read from or to be written to the specified PHY register.

† For CSL implementation, use the notation MDIO_USERACCESS0_*field_symval*

5.3.12 MDIO User Access Register 1 (USERACCESS1)

The MDIO user access register 1 (USERACCESS1) is shown in Figure 5−58 and described in Table 5−60.

Figure 5−58. MDIO User Access Register 1 (USERACCESS1)

31	30	29	28 26	25 21	20 16
GO	WRITE	ACK	Reserved	REGADR	PHYADR
R/WS-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0

15 0
DATA
R/W-0

Legend: R = Read only; R/W = Read/Write; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table 5−60. MDIO User Access Register 1 (USERACCESS1) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	GO			GO bit is writable only if the MDIO state machine is enabled (ENABLE bit in MDIO control register is set to 1). If byte access is being used, the GO bit should be written last. Writing a 1 sets the bit and writing a 0 has no effect.
			0	No effect. The GO bit clears when the requested access has been completed.
			1	The MDIO state machine performs an MDIO access when it is convenient, this is not an instantaneous process. Any writes to USERACCESS1 are blocked.
30	WRITE			Write enable bit determines the MDIO transaction type.
			0	MDIO transaction is a register read.
			1	MDIO transaction is a register write.
29	ACK			Acknowledge bit determines if the PHY acknowledges the read transaction.
			0	No acknowledge.
			1	PHY acknowledges the read transaction.
28−26	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation MDIO_USERACCESS1_*field_symval*

Table 5−60. MDIO User Access Register 1 (USERACCESS1) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
25−21	REGADR		0−1Fh	Register address bits specify the PHY register to be accessed for this transaction.
20−16	PHYADR		0−1Fh	PHY address bits specify the PHY to be accessed for this transaction.
15−0	DATA		0−FFFFh	User data bits specify the data value read from or to be written to the specified PHY register.

† For CSL implementation, use the notation MDIO_USERACCESS1_*field_symval*

5.3.13 MDIO User PHY Select Register 0 (USERPHYSEL0)

The MDIO user PHY select register 0 (USERPHYSEL0) is shown in Figure 5−59 and described in Table 5−61.

Figure 5−59. MDIO User PHY Select Register 0 (USERPHYSEL0)

31	16
Reserved	
R-0	

15	8	7	6	5	4	0
Reserved		LINKSEL	LINKINTENB	—	PHYADDR	
R-0		R/W-0	R/W-0	R-0	R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−61. MDIO User PHY Select Register 0 (USERPHYSEL0) Field Descriptions

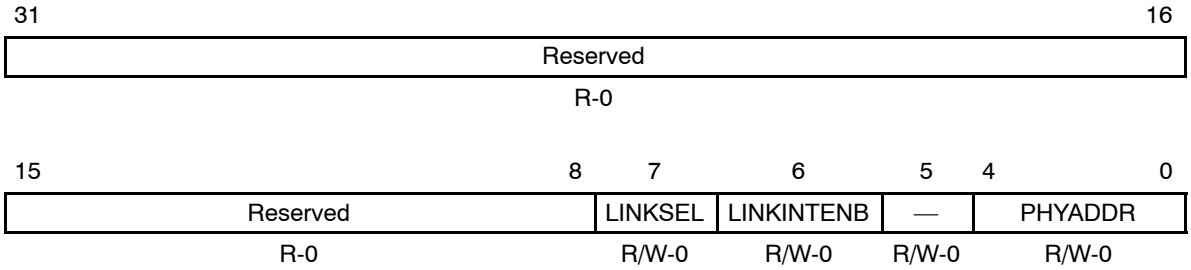
Bit	*field*†	*symval*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	LINKSEL			Link status determination select bit.
		MDIO	0	Link status is determined by the MDIO state machine.
			1	Value must be set to MDIO.
6	LINKINTENB			Link change interrupt enable bit.
		DISABLE	0	Link change interrupts are disabled.
		ENABLE	1	Link change status interrupts for PHY address specified in PHYADDR bits are enabled.
5	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
4−0	PHYADDR		0−1Fh	PHY address bits specify the PHY address to be monitored.

† For CSL implementation, use the notation MDIO_USERPHYSEL0_*field_symval*

5.3.14 MDIO User PHY Select Register 1 (USERPHYSEL1)

The MDIO user PHY select register 1 (USERPHYSEL1) is shown in Figure 5−60 and described in Table 5−62.

Figure 5−60. MDIO User PHY Select Register 1 (USERPHYSEL1)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table 5−62. MDIO User PHY Select Register 1 (USERPHYSEL1) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31−8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	LINKSEL			Link status determination select bit.
		MDIO	0	Link status is determined by the MDIO state machine.
			1	Value must be set to MDIO.
6	LINKINTENB			Link change interrupt enable bit.
		DISABLE	0	Link change interrupts are disabled.
		ENABLE	1	Link change status interrupts for PHY address specified in PHYADDR bits are enabled.
5	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
4−0	PHYADDR		0−1Fh	PHY address bits specify the PHY address to be monitored.

† For CSL implementation, use the notation MDIO_USERPHYSEL1_*field_symval*

Appendix A

Revision History

Table A–1 lists the changes made since the previous version of this document.

Table A–1. Document Revision History

Page	Additions/Modifications/Deletions
Global	Changed all occurrences of C6x device to C6000 device.
iii	Added sentence to first paragraph: Although the entire feature set of the EMAC and MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC and MDIO features.
1-1	Added sentence to first paragraph: Although the entire feature set of the EMAC and MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC and MDIO features.
1-3	Added note to section 1.2: The feature set of the EMAC module may vary between C6000 devices. Please see the device-specific datasheet for a listing of supported features.
2-1	Added sentence to first paragraph: Although the entire feature set of the EMAC module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported EMAC features.
3-1	Added sentence to first paragraph: Although the entire feature set of the MDIO module is described here, the feature set supported on each C6000 device may vary. Please see the device-specific datasheet for a listing of supported MDIO features.
5-1	Added sentence to first paragraph: As the supported feature set may vary between C6000 devices, all of the registers and fields described in this chapter are not supported on each C6000 device. Please see the device-specific datasheet for a listing of supported features.
5-6	Added last sentence to paragraph of section 5.2: Please see the device-specific datasheet for a listing of supported registers.
5-76	Added last sentence to paragraph of section 5.3: Please see the device-specific datasheet for a listing of supported registers.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W