C-PORT™
A Motorola Company

Freescale Semiconductor, Inc...

MOTOROLA
digital dna

C-5 Network Processor Architecture Guide

C-5 NP Do Release

C-Port Corporation
120 Water Street
N. Andover, MA
01845

www.cportcorp.com

C-Port Corporation reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of C-Port Corporation to provide notification of such revision or change.

C-Port Corporation provides this documentation without warranty, term, or condition of any kind, either implied or expressed, including, but not limited to, the implied warranties, terms or conditions of merchantability, satisfactory quality, and fitness for a particular purpose. C-Port may make improvements or changes in the product(s) and/or the program(s) described in this documentation at any time.

Unless otherwise indicated, C-Port registered trademarks are registered in the United States and may or may not be registered in other countries.

C-5, C-Port, the C-Port logo, and C-Ware are trademarks of C-Port Corporation.

Other Trademarks

Adobe and Acrobat are registered trademarks of Adobe Systems, Inc. DigitalDNA and The Heart of Smart are trademarks and Motorola is a registered trademark of Motorola, Inc. Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation. MIPS is a trademark of MIPS Technologies, Inc. Pentium is a registered trademark of Intel Corporation. PowerPC is a trademark of International Business Machines Corporation and used under license therefrom. Solaris is a trademark of Sun Microsystems, Inc. VxWorks, Tornado, and Wind River Systems are registered trademarks or service marks of Wind River Systems, Inc. TeraChannel is a registered trademark of Power X Networks, Inc.PRIZMA-E and PRIZMA-EP are trademarks of IBM, Inc. All other company and product names may be trademarks of their respective companies.

Document Part Number and Publication Date

Part Number: C5NPD0-AG/D
May 31, 2001

Contents

CHAPTER 2

Channel Processors

CHAPTER 3

Executive Processor

CHAPTER 7

Queue Management Unit

APPENDIX A

C-5 NP Registers

APPENDIX B

Using Aggregate Mode

APPENDIX D

PCI Byte Swapping

Figures

Tables

About This Guide

Guide Overview

The C-5 Network Processor Architecture Guide describes the full architecture of the C-5 Network Processor Revision D0. It is intended for system architects and developers to enable you to fully understand how the C-5 network processor (C-5 NP) works and how the processor can be used to implement your networking applications. This guide is also useful as a reference during product design and development, and a Register Reference is provided for that purpose. This guide assumes a good familiarity with communications hardware design and implementation.

This guide covers the following topics:

- Introduction describes the major components and functions of the C-5 NP, supported interfaces, addressing scheme, and cell/packet handling.
- Channel Processors describes the major components and functions of the CPs, processing of data streams, memory areas, interface transactions, configuration space, and using block moves.
- Executive Processor describes the major components and functions of the XP, memory areas, supported external interfaces, initialization options, and internal XP interfaces.
- Fabric Processor describes the major components and functions of the FP, flow process for both FPTx and FPRx, initialization, and Fabric configuration using Utopia, PRIZMA and PowerX interfaces.
- Buffer Management Unit describes the major components and functions of the BMU, memory areas, types of transactions, configuration space, and setup.
- Table Lookup Unit describes the major components and functions of the TLU, flow process, supported table types, commands, configuration and status registers, application considerations, and special applications.
- Queue Management Unit describes the major components and functions of the QMU, flow process, memory areas, queuing operations, types of transactions, configuration space, setup, multicast support, system level multicast operations, and performance.

- Internal Buses describes the interconnect components of the C-5NP including the Ring Bus, Payload Bus and Global Bus.
- C-5 NP Registers lists all the C-5 NP registers including their function, purpose, address, access, fields, bit positions, default values, and options.
- Using Aggregate Mode describes using the C-5 NP in this mode of operation to support Gigabit Ethernet, FibreChannel, OC-12 and OC-12c interfaces.
- SONET/SDH CP Support describes the mapping between the C-5 NP and SONET Byte Overhead definitions for OC-3, OC-3c, OC-12, and OC-12c protocols, and configuration.
- PCI Byte Swapping describes the C-5 NP feature that allows easy transition between the PCI Bus and C-5 NP environments.

Information contained in this guide does not represent a commitment on the part of C-Port Corporation.

Using C-Port Electronic Documents

C-Port electronic documents are provided as PDF files. Open and view them using the Adobe® Acrobat® Reader application, version 3.0 or later. If necessary, download the Acrobat Reader from the Adobe Systems, Inc. web site:

http://www.adobe.com/prodindex/acrobat/readstep.html

Each provided PDF file offers several ways for moving among the document's pages, as follows:

- To move quickly from section to section within the document, use the *Acrobat bookmarks* that appear on the left side of the Acrobat Reader window. The bookmarks provide an expandable 'outline' view of the document's contents. To display the document's Acrobat bookmarks, press the 'Display both bookmarks and page' button on the Acrobat Reader tool bar.
- To move to the referenced page of an entry in the document's Contents or Index, click on the entry itself, each of which is "hot linked."
- To follow a cross-reference to a heading, figure, or table, click the blue text.
- To move to the beginning or end of the document, to move page by page within the document, or to navigate among the pages you displayed by clicking on hyperlinks, use the Acrobat Reader navigation buttons shown in this figure:

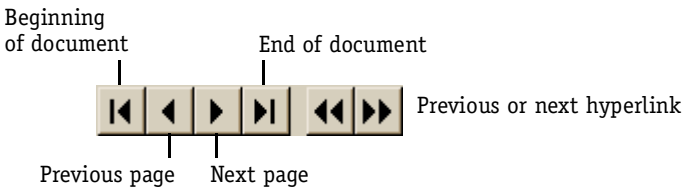


Table 1 summarizes how to navigate within a C-Port electronic document.

Table 1 Navigating Within a C-Port Electronic Document

To Navigate This Way	Click This
Move from section to section within the document.	A bookmark on the left side of the Acrobat Reader window
Move to an entry in the document's Contents or Index.	The entry itself
Follow a cross-reference (highlighted in blue text).	The cross-reference text
Move page by page.	The appropriate Acrobat Reader navigation buttons
Move to the beginning or end of the document.	The appropriate Acrobat Reader navigation buttons
Move backward or forward among a series of hyperlinks you have selected.	The appropriate Acrobat Reader navigation buttons

Guide Conventions

The following visual elements are used throughout this guide, where applicable:

This icon and text designates information of special note.

Warning: *This icon and text indicate a potentially dangerous procedure. Instructions contained in the warnings must be followed.*

Warning: *This icon and text indicate a procedure where the reader must take precautions regarding laser light.*

This icon and text indicate the possibility of electro-static discharge (ESD) in a procedure that requires the reader to take the proper ESD precautions.

Revision History

Table 2 provides details about changes made for each revision of this guide.

Table 2 C-5 NP Architecture Guide Revision History

Revision Date	CST Revision	CDS Revision	Changes
October 23, 2001	1.7	2.0	• Chapter 1, restructured. • Chapter 2, restructured, updated, and added information about understanding block moves of data. • Chapter 4, restructured, enhanced, and updated. • Chapter 5, restructured, enhanced, updated, and added several new topics. • Chapter 6, restructured, updated, and added information about partial CRC-32 support and external tables. • Chapter 7, restructured, enhanced, updated, and added several new topics. • Chapter 8, restructured and updated. • Appendix A, restructured and updated. • Appendix B, restructured. • Appendix C, restructured, updated, and added a monitoring example. • Appendix D, restructured. • Glossary added. Typographic corrections throughout.
September 14, 2000	1.5	1.0	Refer to that printing.

Related User Documentation

Table 3 presents a list of C-5 and CST user documentation that is related to this guide. Find these documents in the Support section of the C-Port Corporation web site:

http://www.cportcorp.com/support/

Table 3 Related Documentation

Document Subject	Document Name	Purpose	Part Number
Processor Information	*C-5 Network Processor D0 Data Sheet*	Describes hardware design specifications for the C-5 network processor	C5NPD0-DS
Software Development Tools	*C-Ware Application Building Guide*	Describes tools to build executable programs for the C-5 network processor or the C-5 Simulator	4-011
	C-Ware Debugger Guide	Describes the GNU-based tool for debugging software running on either the C-5 network processor or C-5 Simulator	4-013
	C-Ware Development System Getting Started Guide	Describes installation of the CDS	4-001
	C-Ware Development System User Guide	Describes operation of the CDS	4-002
	C-Ware Performance Analyzer Guide	Describes use of the Performance Analyzer tool for gathering performance metrics of a C-5 based application running under the C-5 Simulator	4-014
	C-5 Simulator Guide	Describes how to configure and run a simulation of a C-5 based application using the C-5 Simulator tool	4-012
	C-Ware Software Toolset Getting Started Guide	Describes how to quickly become acquainted with the CST's software development tools	4-010
Application Development	*C-Ware Application Design Guide*	Describes design guidelines and trade-offs for implementing new C-5 based communications applications	4-023
	C-Ware CPI User Guide	Describes the subsystems and services that make up the C-Ware Communications Programming Interface for C-5 based communications applications	4-015
	C-Ware Host Application Programming Guide	Describes the CST software infrastructure and CPIs that support host based communications applications	4-018
	C-Ware Microcode Programming Guide	Describes programming the C-5 network processor's Serial Data Processors and Fabric Processor	4-017
	C-Ware Reference Library Guide	Describes the CST's C-5 based Reference Library applications, including how they utilize C-5 network processor resources	4-016

Chapter 1

Introduction

Chapter Overview

This chapter covers the following topics:

- C-5 NP Architecture Overview
- C-5 NP Block Diagram and Flow Process
- C-5 NP Address Mapping

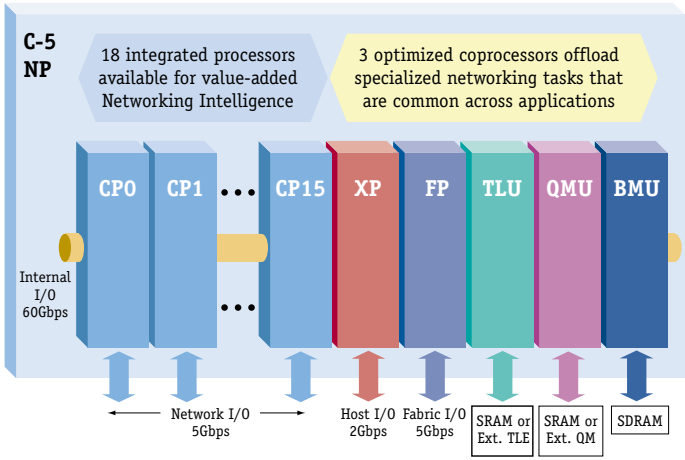
C-5 NP Architecture Overview

The C-5 NP implements an architecture specifically designed for communications applications. Cell and packet processing, table lookup processing, and queue management functions are all integrated into the C-5 NP architecture. With the addition of physical interface chips, memory chips (for payload, circuit/routing tables, and payload descriptor queues), and minimal support logic, a single C-5 NP can be used to implement highly-intelligent mixed media, multiport, multiprotocol switches, multiplexors, and concentrators. Multiple C-5 NPs can be used in conjunction with a switching fabric to implement larger scale switching systems.

Highly-Integrated Architecture

The C-5 NP's highly-integrated architecture employs dedicated processors for each networking channel and a series of coprocessors that offload many common networking-specific tasks. Refer to Figure 1 on page 42. This architecture allows the processors and coprocessors to support concurrent processing, which helps the C-5 NP to deliver software flexibility at hardware speeds. In addition, the C-5 NPs RISC instruction set is specially designed to handle communications functions efficiently, even further enhancing its performance.

Figure 1 C-5 NP Processors and Coprocessors



C-5 NP Modes of Operation

The C-5 NP supports three (3) different modes of operation (Single Channel, Pipeline Channel, and Aggregate Channel) based upon your application needs, allowing you to increase processing power or increase bandwidth.

Single Channel Mode

CPs operate independently of each other at full duplex and can support, for example, OC-3.

Pipeline Channel Mode

To scale processing power for a particular application, the CPs can be linked for pipelined processing on a single data stream. This allows processing power to be applied independently of data rate. Using this mode, different CPs sequentially process cells/packets, achieving a high-level of processing. For example, AAL2.

Aggregate Channel Mode

To scale serial bandwidth capabilities, the CPs can be aggregated into parallel clusters for wider data streams. The C-5 NP's 16 CPs can be partitioned into four (4) groups of four (4) CPs called *clusters*. Clusters allow the CPs to share resources (IMEM and DMEM) and support aggregation. A cluster of CPs can be configured, for example, to work together to support one physical interface (such as OC-12), or either the receive or transmit portion of one physical interface (such as Gigabit Ethernet). For more information about Aggregation in the C-5 NP. Refer to Appendix B.

C-5 NP Supported Interfaces

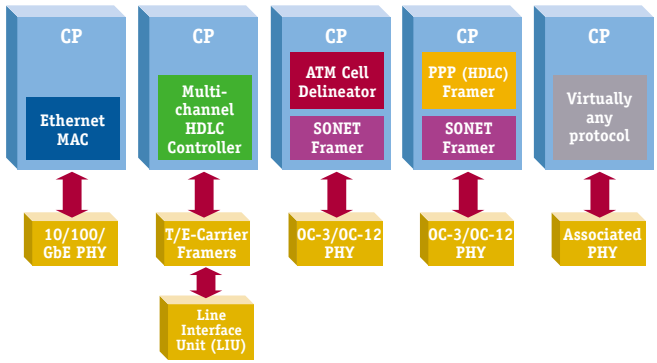
The C-5 NP's architecture supports a variety of industry-standard serial and parallel protocols and individual port data rates ranging from DS1 (1.544Mbps) to Gigabit Ethernet (1000Mbps). The interfaces supported include:

- 10/100Mb Ethernet (RMII)
- 1Gb Ethernet (GMII and TBI)
- OC-3c
- OC-12 (as an aggregation of four OC-3c data streams)/OC-12c
- FibreChannel

Each CP comprises of a set of microprogrammable, special-purpose processors, called Serial Data Processors (SDPs), that provide features such as Ethernet MAC and SONET framing, multichannel HDLC control, and ATM cell delineation. Figure 2 on page 44 shows the physical interfaces and examples of the processing provided by the CP's SDPs for the interface type.

The programmability of the C-5 NP can also support a variety of special interfaces, such as various xDSL encapsulations and proprietary protocols.

Figure 2 Examples of SDP Programmability



Major Components of the C-5 NP

The C-5 NP contains eighteen (18) processors (16CPs, XP, and FP) and three (3) coprocessors that operate as shared resources for the CPs and each other, and perform some networking-specific tasks. Refer to Table 4 on page 44.

Table 4 C-5 NP Major Components

Item	Device Type	Function
Channel Processor (CP)	Programmable Processor	The programmable Channel Processors (CPs) are responsible for receiving, processing, and transmitting cells or packets. The CP's design and on-chip memory architecture incorporate a number of features that result in a uniquely capable engine for the execution of high-performance data communication tasks.
Executive Processor (XP)	Programmable Processor	Provides network control and management functions in user applications. The XP's Peripheral Component Interface (PCI) supplies an industry-standard 32bit 33/66MHz channel to attach additional processors and line interfaces. The XP also has a PROM and serial bus interface.
Fabric Processor (FP)	Programmable Processor	Manages the high-speed fabric interface. FP channels attach to a switch fabric or very high performance line interfaces. The FP supports the UTOPIA-1, -2, and -3 interface standards, as well as, Power X and RRIZMA protocols.
Buffer Management Unit (BMU)	Programmable Coprocessor	Manages centralized payload storage during the forwarding process. An independent high-bandwidth memory interface connects to external memory for the actual storage of payload data.

Table 4 C-5 NP Major Components (continued)

Item	Device Type	Function
Table Lookup Unit (TLU)	Programmable Coprocessor	Provides table search and associated data storage services to the CPs, XP, and FP. An independent high-bandwidth memory interface connects to external memory for storage of circuit and forwarding tables.
Queue Management Unit (QMU)	Programmable Coprocessor	Manages application-defined descriptor queues among the CPs, FP, and the XP. An independent high-bandwidth memory interface connects to external memory for storage of payload descriptor queues.

C-5 NP Interconnect Components

The C-5 NP also contains three (3) independent data buses that provide internal communication paths between the eighteen (18) processors (16CPs, XP, and FP) and three (3) coprocessors, supporting concurrent processing. Refer to Table 5 on page 45.

Table 5 C-5 NP Interconnect Components

Item	Device Type	Function
Payload Bus	Slotted, multichannel, shared, arbitrated bus	Carries payload data and payload descriptors between the processors and the BMU and QMU.
RIng Bus	Slotted ring-topology bus	Provides bounded latency transactions between the processors and the TLU. It also supports inter-processor communication.
Global Bus	Slotted, multichannel, shared, arbitrated bus	Supports inter-processor communication via a conventional flat memory-mapped addressing scheme.

Other Supported Features

In addition, the C-5 NP provides these other features that provide better application integration. Refer to Table 6 on page 45.

Table 6 C-5 NP Other Supported Interfaces

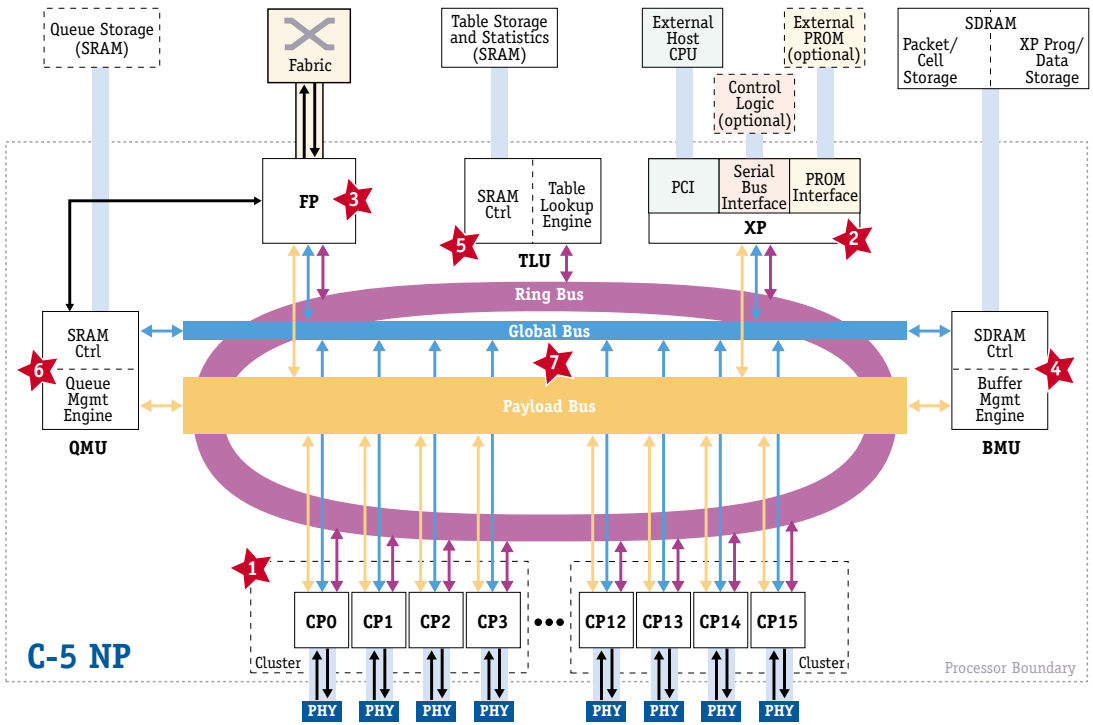
Features	Function
Byte Swapping	Used to move data between the PCI Bus Little Endian environment and the C-5 NP Big Endian environment. Refer to Appendix D for details.
SONET/SDH Support	Provides hardware support to extract, insert and analyze SONET/SDH (Synchronous Digital Hierarchy). Refer to Appendix C for details.
Multicast Operations	Allows multicast elaboration using the BMU and QMU components of the C-5 NP. Refer to "Multicast Support (System Level)" on page 361 for details.

C-5 NP Block Diagram and Flow Process

The full architecture of the C-5 NP is shown in Figure 3 on page 46. The major components of the C-5 NP are numbered on the diagram, and include:

1. Channel Processors (CPs)
2. Executive Processor (XP)
3. Fabric Processor (FP)
4. Buffer Management Unit (BMU)
5. Table Lookup Unit (TLU)
6. Queue Management Unit (QMU)
7. Internal Buses (Ring Bus, Global Bus and Payload Bus)

Figure 3 C-5 NP Simplified Block Diagram



Cell and Packet Forwarding Overview

Each CP within the C-5 NP has special-purpose components that aid in cell/packet parsing and verification. These components also aid in creating separate data and control paths for the cell/packet. Data is sent via the Payload Bus to the BMU for temporary storage in SDRAM. In parallel with the data being stored, application-specific control data is abstracted into a short descriptor that is used to make forwarding decisions. All interfaces use the Ring Bus to consult forwarding tables in the TLU. The interfaces access the QMU (to enqueue frame descriptors to another interface or processor) through the Payload Bus. Cells/packets that are terminated in the chip or require management processing (such as for routing updates) are enqueued for handling by the XP.

Note that the receive and transmit processes can occur on the same or on different CPs.

T he CP handles typical packet forwarding as described in the following sections. The receive and transmit flows are described in detail and shown in Figure 4 on page 49.

Receiving Packets

1. On reception of a serial bit stream, the RxSDP (Receive Serial Data Processor) program detects the packet framing and organizes the bit stream into a byte stream. The SDP program also characterizes and parses the byte stream, performing pattern matching and checking validity criteria.
2. The RxSDP places application-defined fields in Extract Space for access by the CPRC (Channel Processor RISC Core).
3. The SDP launches table lookup requests on extracted data fields using the Ring Bus.
4. Concurrently, the byte stream is transported to a double 64-byte buffer in local DMEM (Data Memory), where it accumulates 64-byte segments (until reaching the end of the packet). The 64-byte segments are transported via the Payload Bus and the BMU to pre-allocated packet buffers in external SDRAM.
5. The CPRC program, upon receiving the extracted data fields and table lookup results, determines the destination queue and other forwarding parameters for the packet, and constructs a forwarding descriptor data structure. This data structure, at a minimum, includes the identity of the packet buffer in which the packet resides. The descriptor is transmitted by the CPRC receive program to the QMU via the Payload Bus. The QMU copies the descriptor into a descriptor buffer and chains that buffer onto the desired queue.

Transmitting Packets

6 The transmitting CPRC program discovers, via a background queue status distribution mechanism, that a queue it services contains a forwarding descriptor. The program reads the descriptor via the Payload Bus from the QMU.

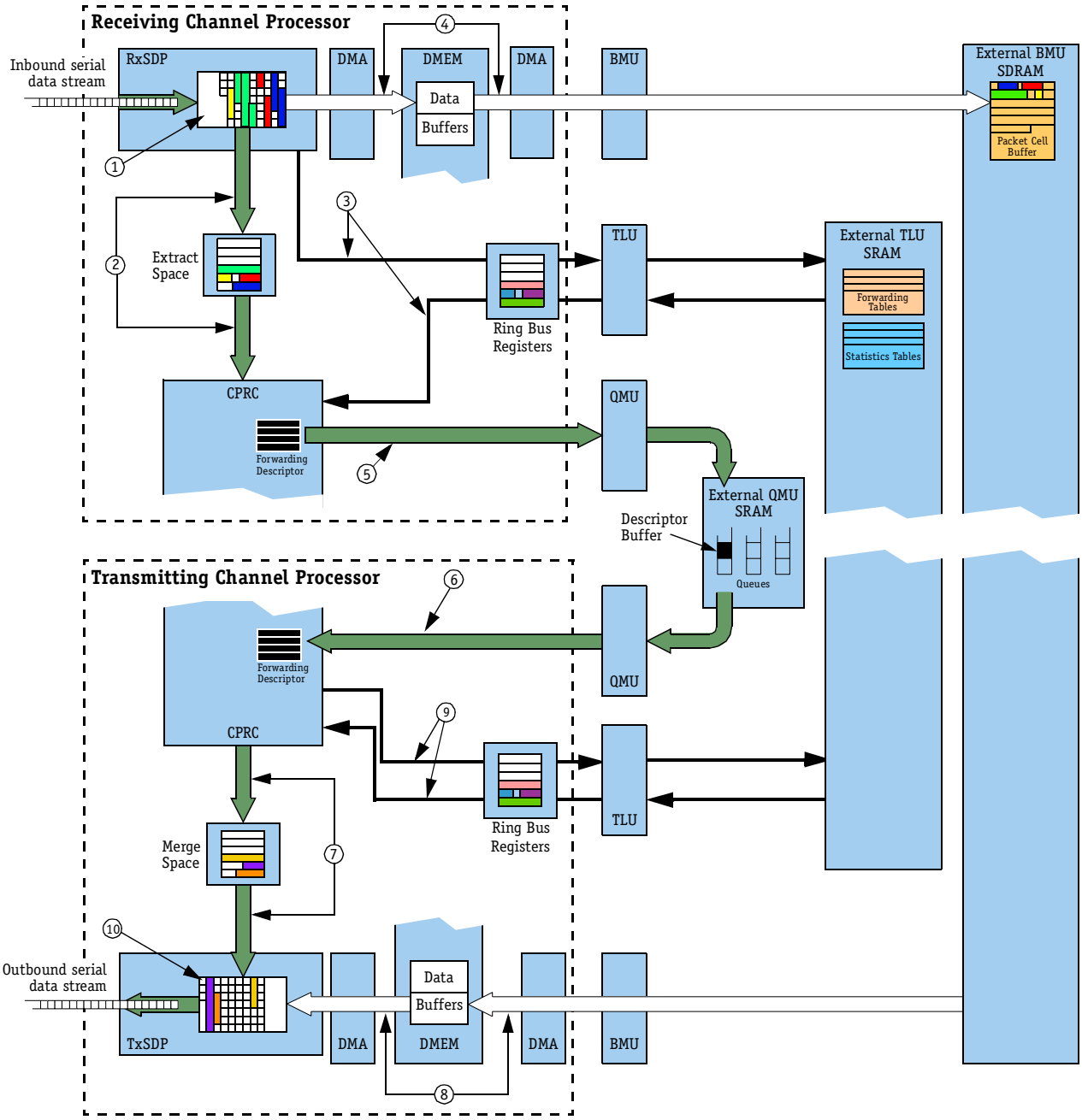
7 The transmit CPRC program inspects the descriptor and using the information it contains, parameterizes the TxSDP program by filling the Merge Space with the format information and packet data field contents necessary to perform the packet transformation. The CPRC sets up the payload transfer from SDRAM via the BMU to local DMEM.

8 The data stream is transported from the BMU via the Payload Bus in 64-byte segments to a double 64-byte buffer in DMEM. The segments are then passed as a byte stream to the TxSDP program, which transforms the packet, substituting data fields from the Merge Space.

9 As a part of either the receive or transmit process, the CPRC program can exercise other C-5 NP resources, such as performing additional table lookups or accessing packet data directly as it flows through the DMEM data buffers.

10 The TxSDP converts the byte stream to a serial bit stream, applies framing, and transmits the bit stream.

Figure 4 Typical Cell/Packet Forwarding Application Receive and Transmit Data Flow



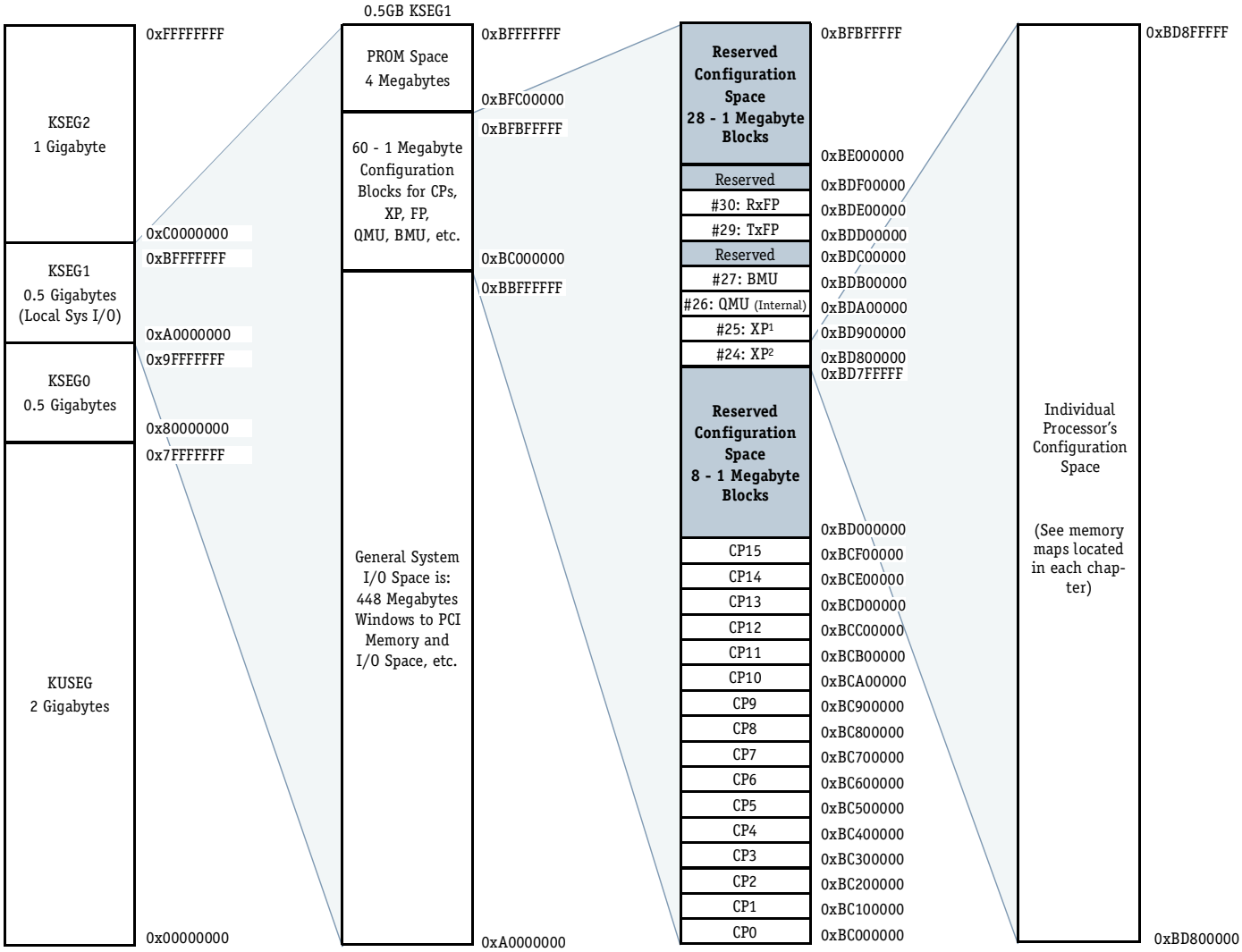
C-5 NP Address Mapping

The C-5 NP supports inter-processor communication using a single, flat memory model. This allows all C-5 NP processors to view all memory-mapped state and configuration registers.

The Channel Processors (CPs) cannot access certain registers reserved exclusively for the Executive Processor (XP).

Each memory resource in the C-5 NP is mapped within its own contiguous 1MByte block of memory. Thus, the specific location of any processor's resource block (local DMEM and registers) within the physical memory space can be mapped using multiple 1MByte offsets.

Figure 5 C-5 NP Physical Address Memory Map



1: XP #25 can only be accessed by the XP, it is not visible to CPs.
2: The CPs can only access DMEM in XP #24.

Configuration Register Definitions

The CP configuration registers form the "base register set" for the C-5 NP. Each CP duplicates the base register set within its own configuration space.

The XP registers include a subset of the base register set, as well as the system interface registers. The XP's "CP-like" base registers are located at the same address offsets within the XP's configuration space as are the CP configuration registers. The FP has a subset of the "CP-like" base registers.

The configuration registers are listed in this chapter by incremental address (CPs, XP, FP, QMU, and so on).

Processor Base Address Offsets

Each C-5 NP processors/coprocessors have a unique 5bit processor ID value. The starting address for any CP (as well as the XP, FP, QMU, BMU, and TLU) can be determined by adding 0xBC000000 to the 5bit processor ID shifted left 20 bit positions, to provide the 1MByte of address space assigned to each processor. For example, the address space for CP5 is: 0xBC000000 + (0x05 << 20) = 0xBC500000. Refer to the memory map in Figure 5 on page 51. In addition, the Ring Bus node IDs for the CPs, XP, FP, and TLU are listed in Table 7 on page 52.

Table 7 Ring Bus Node IDs

Unit	Node ID	Unit	Node ID
CP0	0	CP10	10
CP1	1	CP11	11
CP2	2	CP12	12
CP3	3	CP13	13
CP4	4	CP14	14
CP5	5	CP15	15
CP6	6	XP	24
CP7	7	FP*	30
CP8	8	TLU	31
CP9	9		

* Transmit only. The FP cannot read messages on the Ring Bus. Thus any messages sent to the FP cannot be removed from the Ring Bus, eventually filling up the Ring Bus.

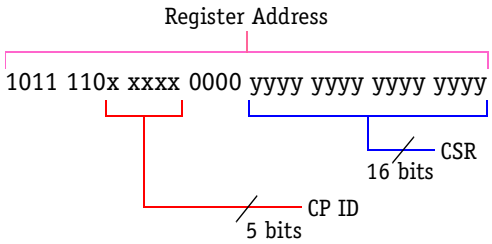
As shown in Figure 5 on page 51, within this space, CP0's 12kBytes of local DMEM begins at the base address (for CP0 this is 0xBC000000), the configuration register space begins at the base address plus 16kBytes (0xBC004000) (with a CP ID shifted of 20 bits). This mapping makes all of DMEM and configuration space available within the positive signed 16bit offset from the CP0 base address.

Configuration Register Address Offsets

Most configuration registers are described only once in this manual. However, some registers have unique variations for different processors. In this case, each register variation is defined.

The register addresses as listed substitute the letter *n* in the address for the processor's ID. By substituting the processor's ID for *n*, you can calculate the address for all individual registers in the C-5 NPs address space. Refer to Figure 6 on page 53 and Figure 5 on page 51.

Figure 6 Register Address Format (in bits)



Byte Ordering

The C-5 NP uses Big Endian byte ordering. However, the XP because of its Peripheral Component Interface (PCI), is capable of handling either Big or Little Endian format from the PCI.

It is recommended that developers use only Big Endian format when developing applications.

Chapter 2

Channel Processors

Chapter Overview

This chapter covers the following topics:

- Channel Processors (CPs) Overview
- Serial Data Processors (SDPs) Overview
- CP RISC (CPRC) Overview
- CP Memory (IMEM and DMEM)
- CP Memory Interface Transactions
- CP Configuration Space
- Understanding Block Moves of Data

Channel Processors (CPs) Overview

The C-5 NP has a dedicated, programmable CP for each of its sixteen (16) line interfaces to handle cell and packet forwarding. The CPs are designated (CP0,CP1 ... CP14, CP15) and can be aggregated to handle higher-speed interfaces and share memory resources. Each CP consists of Serial Data Processors (SDP) and a Channel Processor RISC Core (CPRC), which together perform cell and packet processing via special-purpose memories, namely Instruction Memory (IMEM) and Data Memory (DMEM) that loosely couple these processors. Refer to Appendix B for details about Aggregation.

CP Major Components

The major components of each CP are listed in Table 8 on page 56. In addition, Figure 7 on page 57 shows the CP Block Diagram.

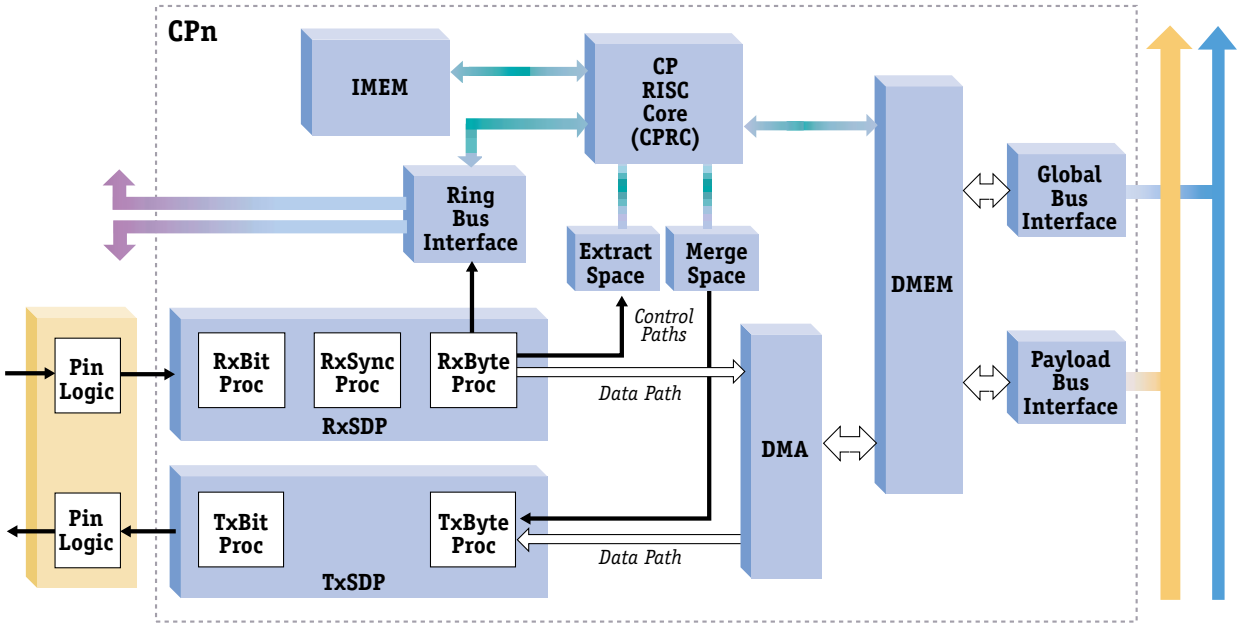
Table 8 Major Components of the CPs and Their Functions

Item	Function
SDPs	Provide microprogrammable interfaces for receive (Rx) and transmit (Tx) between external serial streams and the rest of the CP elements. The receive SDP (RxSDP) and the transmit SDP (TxSDP) can be programmed to process some of the most common types of networking traffic, such as SONET, Ethernet, and ATM. The RxByte programmable processor of the RxSDP and the TxByte programmable processor of the TxSDP can be further programmed for specialized applications. On receipt of a cell/packet, the RxSDP provides serial-to-parallel conversion, validates and interprets the header and payload, and initiates table lookups. On transmission of a cell/packet, the TxSDP applies the header and payload, and provides parallel-to-serial conversion.
CPRC	Programmed to support the following application functions: • Characterizing cells/packets and building descriptors • Initiating additional table lookups • Collecting all table lookup results • Making forwarding and filtering decisions based on the parsed header data and table lookup results (classifying cells/packets) • Making scheduling decisions (based on the characterization of cells/packets) The CPRC implements a subset of the MIPS™ 1 instruction set (excluding multiply, divide, floating point, and CP0 instructions). In addition, the CPRCs support four-way fast context switching.

Table 8 Major Components of the CPs and Their Functions (continued)

Item	Function
Memory	Two (2) types of memory are available: IMEM and DMEM. • Each CP has 6kBytes IMEM that contain the RISC Instructions in RAM. In Cluster mode, four (4) adjacent CPs provide 24kBytes instruction memory (IMEM) that are shared among the CPs within that cluster. • Each CP has 12kBytes of local non-cached data memory (DMEM) for storage of data. Each CPRC can access the local DMEM of any other CPRC within that cluster within one to four additional cycles of latency (depending on CP contention for the DMEM) for a total of 48kBytes. In addition, the DMEM can also be accessed as remote memory by other CPs and the XP via the Global Bus.
Configuration Space	This area of the CP contains a number of registers used to communicate with the SDP and the bus controllers (Payload Bus and Global Bus). The CP's registers can also be accessed by other components of the C-5 NP, (XP and other CPs via the Global Bus).

Figure 7 Channel Processor Block Diagram



Serial Data Processors (SDPs) Overview

Each CP includes a Serial Data Processor (SDP) that contains microcode-programmable components for receive processing (RxSDP) and for transmit processing (TxSDP). Each SDP provides a programmable bit- and byte-level interface to the physical layer (PHY) and acts as the interface between external serial data streams and all other CP elements. The SDP can also launch table lookups to the Table Lookup Unit (TLU).

The SDPs and CPRC operate independently on their specific forwarding tasks and interact to forward a packet to its destination. For example, during receive operations, the RxSDP assembles cell/packet data into DMEM buffers that are written to SDRAM over the Payload Bus under CPRC control. In the process, the RxSDP extracts application-defined fields, placing them in shared registers for access by the CPRC.

Code running on the CPRC characterizes the incoming cell/packet, synthesizes a descriptor that directs the management and routing of the cell/packet, and classifies the cell/packet by enqueuing the descriptor to the appropriate QMU queue. Because the SDP and the CPRC are pipelined to forward cells and packets, many parts of the forwarding process can be performed concurrently.

Supported External Interfaces

Each CP currently supports (in hardware and C-Ware application microcode) the following external interfaces as shown in Table 9 on page 58.

The programmability feature of the SDPs enables support for many other physical interface types, including various xDSL encapsulations and proprietary protocols.

Each type of physical layer PHY has both a characteristic frequency and a particular configuration for the input and output clocks. To accommodate these needs, each CP has a transmit clock mux (txclk mux) that can select among eight (8) global clocks that are sourced externally, two (2) clocks that are sourced from CPs and driven globally, or a clock that is received locally (that is, for OC-12c).

Table 9 Supported Interfaces & Transmit Clock Mux Selects

Supported Interface	Mux Select	Clock Source (MHz)
T1	1	1.544
E1	2	2.048
E3	3	34.368
T3	4	44.736
10/100Mbit Ethernet (RMII)	5	50

Table 9 Supported Interfaces & Transmit Clock Mux Selects (continued)

Supported Interface	Mux Select	Clock Source (MHz)
1GBit Ethernet (GMII and TBI) and 1.0625GBit FibreChannel (TBI)	6	125
OC-3c	7	155.52
	8	internal_0
	9	internal_1
OC-12 and OC-12c	A	aggr rclk
	B	local rclk (loop timing)
	C - F	not assigned

The SDP can be run synchronously with the C-5 NP clock by driving the C-5 NP's clock signal into the corresponding external clock port to which the SDP is configured.

Up to two (2) receive clock muxes (rxclk mux), *internal_0* and *internal_1*, can also be used as transmit clocks for other CPs. This is intended for Telephony applications where the received clock is considered to be more accurate than the local clock source. The CPs that drive these internal clocks are limited to two fixed locations, CP0 and CP8.

CP I/O requires configuration based on the configuration register that specifies the type of port. In addition, there are controls for the rclk mux and the tclk mux that are also based on the port type.

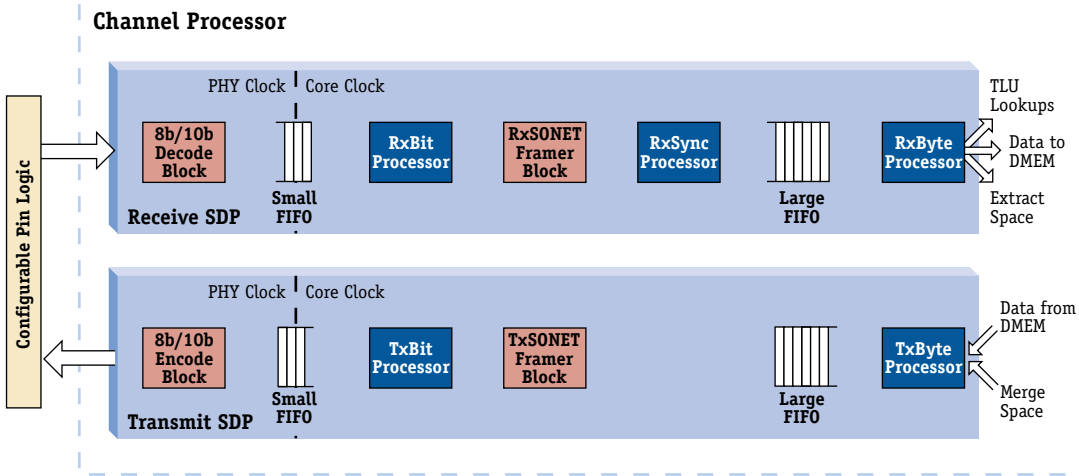
SDPs Functions

The SDP is a pipeline of serially-connected, microcode programmable processors and configurable logic blocks, as shown in Figure 8 on page 60. The data path among these processors and blocks is application configurable. These processors and blocks implement a programmable interface between a port's PHY, where data is serialized, encoded, and encapsulated in protocol-dependent ways, and the CPRC, which expects data to be byte-wide, decoded, delineated, and with header fields naturally aligned.

The receive SDP (RxSDP) accepts serial data from the C-5 NP's physical layer circuitry and performs serial-to-parallel conversion on the data. It delineates frames and cells from the protocol that carries them, performs error and data integrity checks, and compares and extracts fields in the data stream. It provides the contents of the extracted fields to the CPRC in the CPRC's Extract Space. The CPRC's local memory (DMEM) provides buffering for the payload data going to SDRAM.

During data transmission, the CPRC's DMEM buffers the payload data that originates from SDRAM. The transmit SDP (TxSDP) performs field insertion, deletion, and replacement in the outgoing data stream using fields from the CPRC's Merge Space. It performs checksum and CRC generation, frame encapsulation and encoding, and parallel-to-serial conversion on the data. The TxSDP forwards the data to the C-5 NP's external physical layer circuitry.

Figure 8 Rx and Tx SDP Programmable Processors and Configurable Logic Blocks



If a set of CPs is aggregated, those CPs' SDPs are also configured as aggregated. The behavior of aggregated SDPs is described in Appendix B.

SDPs Major Components

The RxSDP and TxSDP features two (2) types of hardware. Refer to Table 10 on page 61.

Table 10 Types of Hardware Features in the RxSDP and TxSDP

Type of Hardware Features	Items	Function
Programmable Processors	RxBit Processor, RxSync Processor, RxByte Processor, TxBit Processor and TxByte Processor. Refer to Figure 8 on page 60.	Each embedded processor has special-purpose hardware such as a dedicated instruction store, a bank of internal registers, ALU, CAM, CRC engines, and so on for performing a unique set of operations. See the *C-Ware Microcode Programming Guide* for more information.
Configurable Logic Blocks	8b/10b Decode Block, RxSONET Framer Block, 8b/10b Encode Block and TxSONET Framer Block. Refer to Figure 8 on page 60.	Each block is optimized to perform one function, and, while not fully programmable is configurable by the application. Examples of these configurable logic blocks are the SONET Framers and the 8b/10b Decode/Encode Blocks used for TBI protocols.

These programmable processors and configurable logic blocks are described in more detail in "RxSDP Detail Operations" on page 64 and "TxSDP Detail Operations" on page 68.

The SDP's processors *are microcode programmable, and SDP's* blocks *are configurable only.*

Each pipeline configuration uses FIFO blocks between certain pairs of processors. The FIFOs allow for some elasticity during relatively short periods of time when one SDP processor's throughput does not exactly match that of another that feeds it or is fed by it.

The run-time path for data traffic through this pipeline of SDP processors and blocks is configured by the application using higher-level calls to the C-Ware Communications Programming Interfaces (CPIs). This configuration takes place under application control as part of initializing the C-5 NP's ports. The available pipeline configurations are determined at the application level. Refer to the *C-Ware Reference Library* document in the *C-Ware Application Development Guide* for more information. These CPI calls configure the RxSDP and TxSDP by setting the appropriate bits in the *SDP_Mode3* (RxSDP) and *SDP_Mode5* (TxSDP) registers.

By distributing the SDP's tasks among a pipeline of embedded programmable processors and configurable logic blocks, the available processing achievable at wire speed can be quite high. For OC-12, Gigabit Ethernet, and FibreChannel, the C-5 NP's SDPs can be configured (via C-Ware CPIs) into aggregated channels that cooperatively achieve an even higher aggregate throughput.

A CP can also be configured to allow *recirculation* of data traffic from the TxSDP to the RxSDP. Refer to "Configuration for Recirculation Operations Using RxSDP and TxSDP" on page 72.

Common Components of the Programmable Processors

Refer to Table 11 on page 62 and Figure 9 on page 62 for the common components of the programmable processors (RxBit, RxSync, RxByte, TxBit and TxByte) inside the SDPs and their functions.

Figure 9 Common Components of Programmable Processors

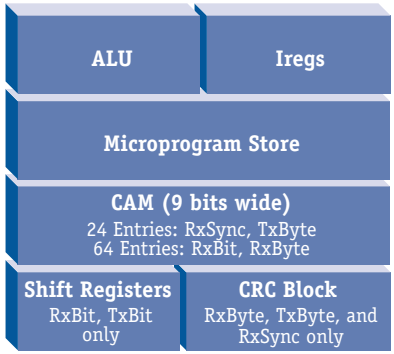


Table 11 Common Components of Programmable Processors and Their Functions

Item	Function
Arithmetic Logic Unit (ALU)	The eight-bit arithmetic and logic unit, where computations take place.
Instruction Registers (Iregs)	Consists of sixteen (16) (Ireg0 to Ireg15) registers for the embedded processor's internal use: • Ireg0 to Ireg3 can serve as general-purpose registers but also can serve as counters (can be incremented directly by microcode). Ireg0 to Ireg3 can be incremented separately as eight-bit registers or as 16bit register pairs (Ireg0/Ireg1 and Ireg2/Ireg3). If Ireg0 to Ireg3 are used as pairs, Ireg1 is the least-significant half of its pair, as is Ireg3. • Ireg4 to Ireg9 are eight-bit, general-purpose registers. • Ireg10 to Ireg12 are currently not implemented. • Ireg13's two least significant bits (LSBs) provide access to the ninth bit of Ireg0 and of Ireg15. • Ireg14 is the Control/Status register. • Ireg15 is the microcode program counter.

Table 11 Common Components of Programmable Processors and Their Functions (continued)

<table>
<tr><th>Item</th><th>Function</th></tr>
<tr><td>Microprogram Store</td><td>Provides storage for the processor's microcode program. It comprises a series of 52bit microprogram words. Its characteristics are shown here:
<table>
<tr><th>Programmable Processors</th><th>Microprogram Store Size *</th></tr>
<tr><td>RxBit</td><td>64 words</td></tr>
<tr><td>RxSync</td><td>64 words</td></tr>
<tr><td>RxByte</td><td>512 words</td></tr>
<tr><td>TxBit</td><td>256 words</td></tr>
<tr><td>TxByte</td><td>384 words</td></tr>
</table>
* 52bit word</td></tr>
<tr><td>Content Addressable Memory (CAM)</td><td>Provides a nine (9) bit wide lookup table. Its characteristics are shown here:
<table>
<tr><th>Programmable Processors</th><th>CAM</th></tr>
<tr><td>RxBit</td><td>64 entries x 9 bits wide</td></tr>
<tr><td>RxSync</td><td>24 entries x 9 bits wide</td></tr>
<tr><td>RxByte</td><td>64 entries x 9 bits wide</td></tr>
<tr><td>TxBit</td><td>None</td></tr>
<tr><td>TxByte</td><td>24 entries x 9 bits wide</td></tr>
</table></td></tr>
<tr><td>Shift Registers</td><td>Supports serial-to-parallel conversion.
Note: Only applies to RxBit and TxBit programmable processors.</td></tr>
<tr><td>Cyclic Redundancy Check (CRC)</td><td>Provides CRC checking, generation, and scrambling.</td></tr>
</table>

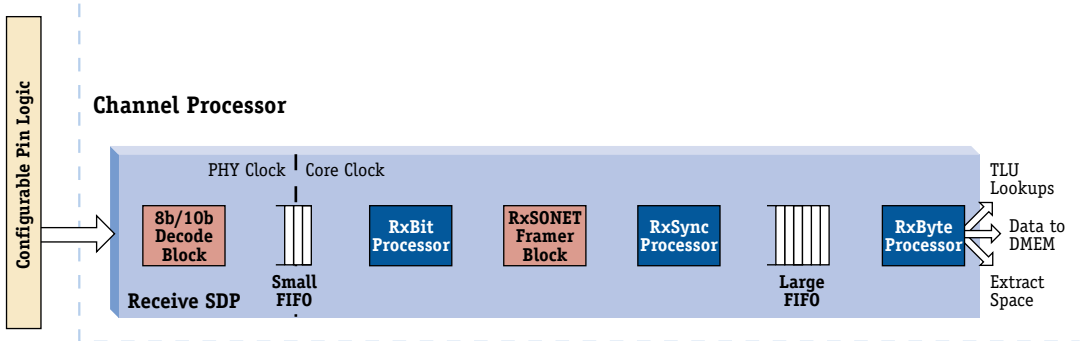
The CPRC restarts each SDP processor by toggling a reset signal. The microprogram stores and CAMs are visible to the CPRC and XP programs only for the purpose of loading these memories. The other registers and counters are not visible to CPRC and XP programs.

For information about the programmability of the SDP's embedded processors, see the *Microcode Programming Guide*.

RxSDP Detail Operations

This section provides more detail information regarding each RxSDP's programmable processors and configurable logic blocks. The individual items are shown in Figure 10 on page 64.

Figure 10 RxSDP Programmable Processors and Configurable Logic Blocks



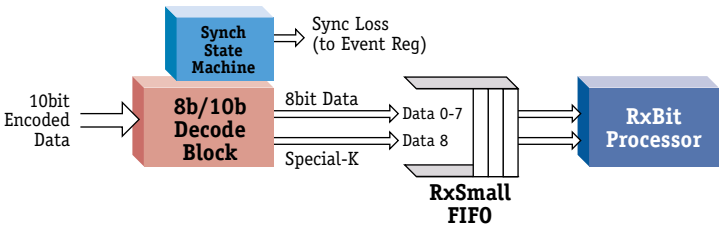
8b/10b Decode Configurable Logic Block

The 8b/10b Decode configurable logic block is located in the RxSDP.

This block contains hardware for encoding and decoding between 8-bit and 10-bit formats. This enables support for protocols that require 8b/10b encoding, such as FibreChannel or Gigabit Ethernet over the Ten Bit Interface (TBI).

As shown in Figure 11, in the RxSDP the 8b/10b Decode block decodes the 10-bit encoded data into its 8-bit equivalent, plus a Special-K signal, which is passed upstream as the ninth bit of the data stream. This bit is used during RxBit CAM lookups to differentiate between control and data characters. Using this decoded data stream, RxBit can be microprogrammed to perform frame delineation and to implement the Physical Coding Sublayer (PCS) state machine, defined in the IEEE 802.3 specification, section 36.

Figure 11 Operation of 8b/10b Decode Configurable Logic Block



Also in the 8b/10b Decode block, a special hardware component implements the synchronization state machine defined in the IEEE 802.3z specification, section 36.2.5.2.6 and in FibreChannel specifications. This component tracks whether comma code groups appear on odd or even positions in the incoming data stream to establish a state of synchronization, after which it tracks the occurrence of running disparity errors. After observing a defined density of disparity errors, the SyncLoss signal is asserted. This signal sets a bit in the *SONET_Event* register which then sets a bit in the *Event0* register that can be programmed to cause a CPRC interrupt, which allows the CPRC the opportunity to take corrective action, such as to initiate a re-establishment of the physical link.

RxSmallFIFO Configurable Logic Block

The RxSmallFIFO configurable logic block provides elasticity for the RxBit Processor's throughput requirements.

The RxSmallFIFO block also bridges the network clock domain with the C-5 NP's internal clock domain. The block has two (2) parts: the internal half's clock runs at the same frequency as the core clock, and the external half's clock runs at the same frequency (or a submultiple) as the external PHY's clock. This allows the input data stream to be converted to the core clock's higher frequency, thereby increasing all SDP throughput.

The RxSmallFIFO is sixteen (16) locations deep and ten (10) bits wide (that is, eight (8) data bits and two (2) control bits).

RxBit Programmable Processor

The RxBit programmable processor performs frame and cell delineation (including serial-to-parallel conversion), pattern matching, and field extraction on serial data up to nine (9) bits at a time. The extracted fields are written to the CP's Extract Space.

The payload output data is nine (9) bits wide.

The RxBit programmable processor can take specific actions in microcode in response to a particular pattern match. The RxBit programmable processor processes the data stream as a function of position. Pattern matches can include "don't cares." Extracted fields and status are written to programmable locations in the CP's Extract Space.

In a sense, the RxBit programmable processor is an intelligent serial-to-parallel converter, in that it is:

- Capable of detecting High-level Data Link Control (HDLC) frames and invalid sequences, as well as removing stuffed zeroes.
- Used to find the Synchronous Transport Signal (STS) frame in an OC-3c data stream.

- Capable of identifying and deleting the preamble of incoming Ethernet frames.
- Used for 1000BASE-X Gigabit Ethernet delimiter recognition.
- Used to implement the receive side of the 1000BASE-X physical convergence sub-layer of IEEE 802.3z.
- Used to delineate FibreChannel frames.

RxSONET Framer Configurable Logic Block

The RxSONET Framer configurable logic block obtains recovered receive clock frame sync (A1 and A2) and eight-bit (8) data from the physical layer interface chip or from the RxBit programmable processor. It descrambles the data, demultiplexes the transport overhead, checks (B1, B2) parity, and writes the overhead octets into the CP's register space. Refer to Figure 10 on page 64. Each STS frame has its own parity checker.

The RxSONET Framer also interprets the STS pointer in order to extract the Synchronous Payload Envelope (SPE). From the payload envelope, it demultiplexes the path overhead, checks (B3) parity, and writes the other overhead bytes to the CP's Extract Space. The remaining payload is passed downstream, which handles concatenated formats only.

The RxSONET Framer does no demultiplexing of the SPE payload. As such, it is only suitable for receiving frames or cells.

The RxSONET framer also provides support for detection and monitoring by software of various SONET/SDH defects such as Loss of Signal (LOS), Loss of Frame (LOF), Loss of Pointer (LOP), Path Remote Defect Indication (RDI-P), Line Alarm Indication Signal (AIS-L) to name just a few. Events of interest can be masked to enable either access via RC interrupt or polling.

RxSync Programmable Processor

RxSync is a general-purpose programmable processor that can be used to perform any generic processing to off-load either the RxBit or RxByte Processors. One example of this is synchronizing on ATM cells.

When configured for ATM, the RxSync programmable processor receives a byte data stream consisting of 53-byte ATM cells. It finds the cell boundaries by applying the Header Error Check (HEC) CRC sequentially to five-byte (5) segments, checking the first four (4) bytes against the fifth HEC byte. If the check fails, the cell delineator repeats the process. When it finds an application-defined number of successful HEC checks in a row, it enters the *in_sync* state. It remains in this state until there has been an application-defined number of consecutive HEC check failures, then it resumes the search.

After the data stream is synchronized, the RxSync programmable processor appends a status byte to the data stream so that the cell could be discarded if the HEC does not check.

The RxSync programmable processor can be programmed to parse the cell header, writing (Virtual Path Identifier (VPI), Virtual Channel Identifier (VCI)) payload_type and cell_loss_priority to locations in the CP's Extract Space.

When configured for Fast or Gigabit Ethernet, the RxSync programmable processor assists in 802.3x pause packet processing, passing the pause time up to the CPRC for processing.

The RxSync programmable processor is also capable of handling the HDLC byte escape sequence for Point-to-Point Protocol (PPP) over SONET.

RxLargeFIFO Configurable Logic Block

The RxLargeFIFO configurable logic block is located in front of the RxByte programmable processor and provides elasticity for the RxByte programmable processor, which typically has the greatest computational responsibility within the RxSDP. Therefore, this block can stage a cell while its VPI/VCI is being looked up by the C-5 NP's TLU.

RxByte Programmable Processor

The RxByte programmable processor performs pattern matching and field extraction. It also detects and parses the Ethernet and IP headers when acting as a pre-processor to the CPRC for switching and routing applications. It can extract fields and launch TLU lookups on fields, stream data to DMEM and to the CP's Extract Space, and has responsibility for interfacing with the CPRC.

The RxByte programmable processor can also perform CRC computations and checks and has the largest amounts of Content Addressable Memory (CAM) and microcode store space. It is expected that the RxByte microcode contains the majority of the application's RxSDP-resident custom functionality.

The RxByte programmable processor can take specific actions as a result of a particular pattern match on nine (9) bit words of data. The nine (9) bit match values are stored in a CAM, which is loaded along with the SDP's microcode. Refer to the *SDP Programming* document in the *C-Ware Application Development Guide* for details about CAMs.

The extracted fields are written to the CP's Extract Space, which is memory-mapped into the CP's Configuration Space. The RxByte programmable processor can also initiate lookups in the C-5 NP's TLU on the extracted fields via the Ring Bus interface.

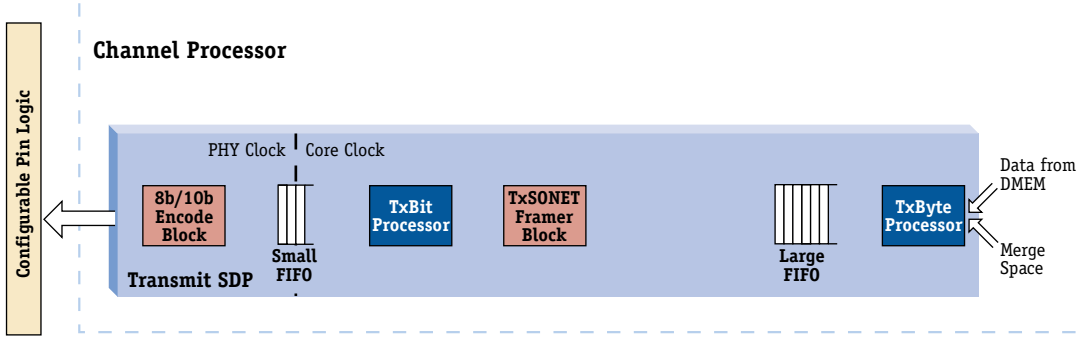
The receive data can pass through the RxByte programmable processor multiple times (via an idle port) by reading the processed data back from local memory and operating on it again via the SDP's recirculation path. Refer to "Configuration for Recirculation Operations Using RxSDP and TxSDP" on page 72 for details.

The RxByte programmable processor implements operations with CRC-16 and CRC-32 only. It can be programmed to check Ethernet and ATM Adaptation Layer 5 (AAL5) Frame Check Sequence (FCS). AAL5 CRC checking is performed with assistance from the TLU.

TxSDP Detail Operations

This section provides more detail information regarding each TxSDP's programmable processors and configurable logic blocks. The individual items are shown in Figure 12 on page 68.

Figure 12 TxSDP Programmable Processors and Configurable Logic Blocks



TxByte Programmable Processor

The TxByte programmable processor can be programmed to read data from DMEM, generate a valid CRC, and insert, delete, and replace fields in the egress traffic data stream. Like the RxByte programmable processor, it is expected that the TxByte microcode contains the majority of the application's TxSDP-resident custom functionality.

The transmit data can pass through the TxByte programmable processor multiple times (via an idle port) by writing the processed data back to local memory and operating on it again via the SDP's recirculation path. Refer to "Configuration for Recirculation Operations Using RxSDP and TxSDP" on page 72 for details.

TxLargeFIFO Configurable Logic Block and Options

The TxLargeFIFO configurable logic block is located after TxByte programmable processor and provides elasticity for the TxByte programmable processor, which typically has the greatest computational responsibility within the TxSDP. This block provides elasticity for field inserts and deletes by the TxByte programmable processor.

This FIFO is one-hundred-twenty-eight (128) locations deep and ten (10) bits wide (eight (8) bits of data and two (2) control bits). The second control bit is not accessible to the microcode and is used only for optional payload scrambling in OC-12 mode. The OC-12 data stream must consist of four (4) OC-3c streams. In addition, the TxLargeFIFO allows two (2) selectable options: Automatic Idle Cell and PPP Flag Insertion, and Transmit FIFO High Water Mark as described here.

Automatic Idle Cell and PPP Flag Insertion Option

For the C-5 NP Version D0, the TxLargeFIFO supports automatically inserting ATM idle cells or PPP Flag characters into the output stream when needed.

For applications running on hardware prior to the C-5 NP Version D0, the TxByte microcode was required to perform this work. That microcode was conservative about when to send idle cells or PPP flags, to ensure that the TxLargeFIFO never emptied completely. This resulted in more idle cells or PPP flags being sent out than necessary, which in turn caused reduced bandwidth over the physical interface.

The new feature for the C-5 NP Version D0 sends idle cells or PPP flags only when the TxLargeFIFO becomes empty. Using this feature ensures that only the minimal number of idle cells or PPP flags are sent between packets or user data cells. It also allows TxLargeFIFO to be filled with real payload and not cluttered with idle characters.

To use the new feature, set the *Idle Insert Enable* bit in the *SDP_MODE4* register. For ATM applications, also set the *Idle Cell Mode* bit in the *SDP_MODE4* register. For PPP flag insertion, make sure that the *Idle Cell Mode* bit is clear. Refer to "SDP_Mode4 Register (CP Mode Configuration Function)" on page 422.

Transmit FIFO High Water Mark Option

Most applications process a protocol header in the SDP's TxByte serial processor prior to streaming payload data. The speed at which it processes a header is typically less than the transmit speed of the physical interface. If the header bytes were popped off the TxLargeFIFO at the rate of the physical interface while TxByte is sending them out at a rate less than that of the physical interface, an under-run condition can occur. An under-run causes corrupted data to be sent out on the interface when no payload data is available.

To work-around this problem prior to the C-5 NP Version C0, applications padded the TxLargeFIFO with enough flag bytes or idle cells to allow TxByte enough cycles to process a header. For all non-SONET applications, TxByte can signal TxBit when the data is ready to send at the physical interface speed. TxByte usually notifies TxBit at some point after it finishes processing the protocol header. In the meantime, TxBit sends out idle bytes to prevent under-running the physical interface.

In the C-5 NP Version D0 a *high water mark* feature has been added to eliminate the need to pad the TxLargeFIFO or coordinate with TxBit. The high water mark is a count of bytes that must be in the TxLargeFIFO for it to appear non-empty to the downstream SDP components that pop data out of it.

The high water mark value is set in the *SDP_MODE4* register. Setting this value to zero results in the same behavior as occurred in the C-5 NP Version C0 chip. Setting *SDP_MODE4* to a non-zero value causes the transmit FIFO to appear empty until the "high water mark" number of bytes are in the TxLargeFIFO. An application typically sets the high water mark value to the number of bytes in the protocol header that TxByte must process at less than the speed of the physical interface. Refer to "SDP_Mode4 Register (CP Mode Configuration Function)" on page 422.

The high water mark depth is tested at the end of each packet. The transmit FIFO uses the ninth bit as the end of packet or cell indication. If the TxLargeFIFO depth is less than the high water mark when the last byte of a packet or cell is unloaded "popped" from the TxLargeFIFO, the FIFO appears empty.

This feature can be used with the *automatic Idle Cell and PPP Flag insertion* logic. This logic inserts idle cells or PPP flags whenever a downstream SDP component unloads "pops" a byte from TxLargeFIFO and the FIFO appears empty.

TxSONET Framer Configurable Logic Block

The TxSONET Framer configurable logic block must receive data from TxLargeFIFO configurable logic block. It obtains most of the transport overhead and path overhead from the *TxSONETOH0* to *TxSONETOH31* registers. This allows the application developer to insert values into the transmit overhead. Refer to Figure 12 on page 68.

The transmit pointer value (bytes H1 and H2 in the SONET frame's Path Overhead section) is fixed.

The TxSONET Framer block generates B1, B2, and B3 bit-interleaved parity, then inserts the OC-12c overhead into the payload data and scrambles it to form either a SONET OC-3c, OC-12, or OC-12c format.

TxBit Programmable Processor

The TxBit programmable processor receives byte-wide data. Under microcode control it inserts, deletes, and replaces fields. Typically, the TxBit programmable processor provides the reverse functions of the RxBit programmable processor. TxBit contains a special-purpose shift register for performing parallel-to-serial conversion.

The TxBit programmable processor can impose minimum inter-frame gaps and monitor PHY status, for instance, in order to detect MAC collisions.

Input to the TxBit programmable processor is nine (9) bits wide and its output can be one, two, four, eight or ten (1, 2, 4, 8, or 10) bits at a time, depending on the type of physical interface.

The TxBit programmable processor can be viewed as an intelligent parallel-to-serial converter, because:

- The TxBit programmable processor modifies the data stream as a function of the data stream.
- The TxBit programmable processor is used for collision detection when configured for half-duplex 10Mbit/100Mbit Ethernet or Gigabit Ethernet.
- The TxBit programmable processor is used to implement the transmit side of the 1000BASE-X physical convergence sub-layer of IEEE 802.3z.

The TxBit programmable processor is capable of inserting the HDLC frame sequence, as well as stuffing zeros into the data stream as appropriate, under microcode control.

TxSmallFIFO Configurable Logic Block

The TxSmallFIFO configurable logic block provides elasticity for the TxBit programmable processor's throughput requirements.

The TxSmallFIFO configurable logic block also bridges the network clock domain with the C-5 NP's internal clock domain. The TxSmallFIFO configurable logic block has two (2) parts: the internal half's clock runs at the same frequency as the C-5 NP's core clock, and the external half's clock runs at the same frequency (or a submultiple) as the external physical layer's clock. This allows the outgoing data stream to be converted from the core clock's frequency to the PHY's clock frequency.

The TxSmallFIFO configurable logic block is sixteen (16) locations deep and ten (10) bits wide (that is, eight (8) data bits and two (2) control bits).

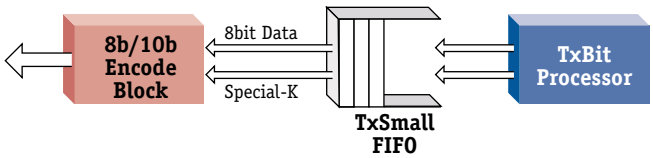
8b/10b Encode Configurable Logic Block

The 8b/10b Encode configurable logic block is located in the TxSDP.

This one (1) block contains hardware for encoding between 8bit and 10bit formats. This enables support for protocols that require 8b/10b encoding, such as FibreChannel or Gigabit Ethernet over the Ten Bit Interface (TBI).

As shown in Figure 13 on page 72, in the TxSDP the 8b/10b Encode block receives as input the 8bit data and Special-K bit (again, in the ninth bit position) and outputs the appropriate 10bit encoded value.

Figure 13 Operation of 8b/10b Encode Configurable Logic Block

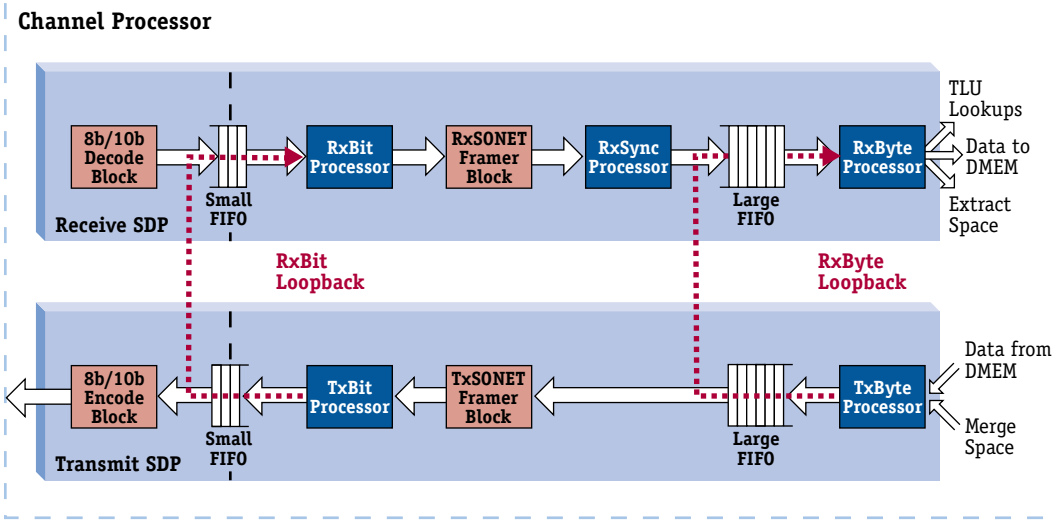


When transmitting idle code groups to the 8b/10b Encode block, the TxBit Processor should be programmed to transmit a series of (K28.5, D31.7). The D31.7 will be converted to the appropriate Dx.y value, either D5.6 or D16.2, depending on the block's internal odd/even state. This relieves TxBit from the burden of tracking the odd/even state of its output stream.

Configuration for Recirculation Operations Using RxSDP and TxSDP

Enabling *recirculation* for an SDP means to configure its RxSDP and TxSDP so that the output from the TxSDP processor is routed to the input of its corresponding RxSDP processor. This method is one way to pipeline your C-5 NP. The C-5 NP Pipeline Channel Mode, allows you to scale processing power for a particular application, the CPs can be linked for pipelined processing on a single data stream. Refer to Figure 14 on page 73.

Figure 14 SDP Recirculation Path Using Both RxBitLoopBack and RxByteLoopBack Bits



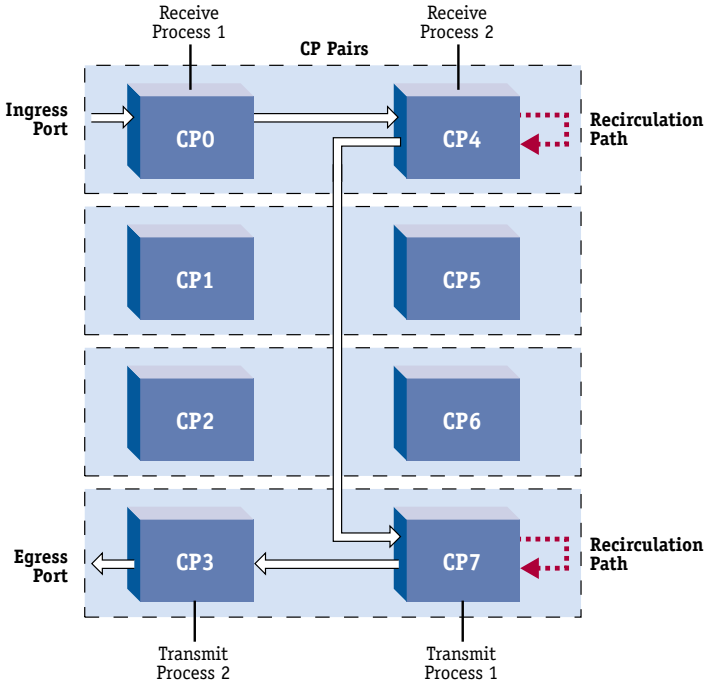
The SDP can be configured to permit recirculation of the data path in either of two (2) ways:

- From the TxByte programmable processor's output to the RxByte programmable processor's input, enabled by setting the *SDP_Mode3* register bit [25] *RxByteLoopBack* field.
- From the TxBit programmable processor's output to RxBit programmable processor's input, enabled by setting the *SDP_Mode3* register bit [24] *RxBitLoopBack* field.

Enabling recirculation for a CP's SDP can be useful in two (2) different ways: pipelining packet processing (during normal operations) and debugging and test.

- During normal operation, the ingress CP performs packet classification then enqueues the packet descriptor. If additional classification is desired, another CP with loopback set can dequeue the packet descriptor and process the packet again. The payload is moved from SDRAM through DMEM in the TxSDP. Using either the byte or bit loopback, the data is brought back through the RxSDP for further classification. The second CP then enqueues the new packet descriptor for either actual transmit or for another stage of loopback processing for classification or forwarding decisions. Figure 15 on page 74 summarizes the data flow during normal operations. The recirculation operation can be used for any application that requires greater degrees of packet processing. For example, with applications such as multichannel HDLC and encryption that use T1 and T3 data rates.

Figure 15 Recirculation Shown for Normal Operations (for Cooperating CPs)



- SDP loopback can be used to debug the output of the TxSDP. Transmit data can be brought back on the chip for traffic analysis or chip test through the RxSDP without ever having to move data off chip.

CP RISC (CPRC) Overview

Each of the sixteen (16) CPs has a Reduced Instruction Set Computer (RISC) Core. The dedicated CPRC in each channel orchestrates cell and packet processing. The CPRC operates at the C-5 NP's core clock rate.

The CPRC contains a 32bit data path and accesses memory using a 32bit physical address. Within the address space, a CPRC can reference its own local memory with zero-wait-state latency in the absence of remote contention from the Global Bus. Memory addresses outside of local memory range refer to remote memory space contained within other CPs and the Executive Processor (XP).

The CP contains memory-mapped, shared control registers used for CPRC-to-SDP communication. Refer to "CP Configuration Space" on page 87. The shared control registers also enable the CPRC to control the Payload and Ring Buses and enable the XP to configure the channel during initialization.

RISC Instruction Set Supported

The CPRC executes a MIPS™1 instruction set (excluding multiply, divide, floating point). The CPRC supports the classes of instructions shown in Table 12 on page 75. The CPRC includes four (4) sets of 32 internal registers. Each associated with a CP's context. These internal registers are used to support *fast context switching*. These internal registers are defined in Table 13 on page 76.

The standard MIPS Coprocessor Zero (CpO) instructions are not supported. However, C-Port provides its own special purpose Coprocessor Zero registers. Refer to Table 14 on page 77.

It is highly recommended that you use the C-Ware Compiler when building your application code. Therefore, refer to the C-Ware Application Development Guide for information on using the C-Port compiler, which supports the CPRC.

Table 12 CPRC Supported Instruction Classes

Class of Instruction	Description
Load and Store	Load immediate values and move data between memory and general registers.
Computational	Perform arithmetic and logical operations for values in registers.
Jump and Branch	Change program control flow.
Coprocessor Interface	Provide standard interfaces to the coprocessors.
Special	Perform miscellaneous tasks.

Table 13 CPRC (32) Internal Registers Definitions

CPRC Internal Register Names	Software Name	Use and Linkage
$0	—	Always has the value of 0.
$at or $1	—	Reserved for the assembler.
$2:$3	v0 to v1	Used for expression evaluations and to hold integer function results. Also used to pass the static link when calling nested procedures.
$4:$7	a0 to a3	Used to pass the first four words of integer type actual arguments. Their values are not preserved across procedure calls.
$8:$15	t0 to t7	Temporary registers used for expression evaluations. Their values are not preserved across procedure calls.
$16:$23	s0 to s7	Saved registers. Their values must be preserved across procedure calls.
$24:$25	t8 to t9	Temporary registers used for expression evaluations. Their values are not preserved across procedure calls.
$26:$27 or $kt0:$kt1	k0 to k1	Used internally by the C-5 NP system services.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30	s8	A saved register (like s0 - s7).
$31	ra	Contains the return address used for expression evaluation.

See the *MIPSpro™ Assembly Language Programmer's Guide* for information about the standard MIPS™1 instruction set. It is available at http://www.mips.com/publications/index.html.

Table 14 C-Ports Coprocessor Zero Register Definitions

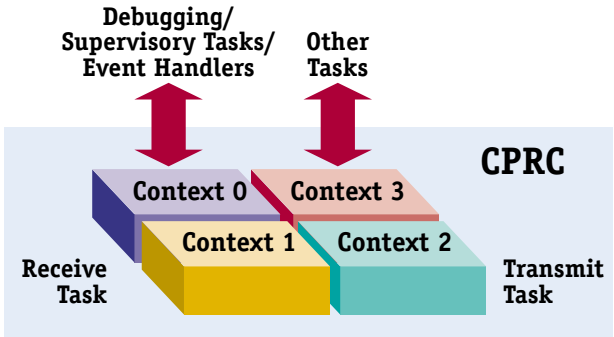
Register	Definition
R0	**Whoami Register** — Contains the DMEM base (hardcoded) for this CPRC.
R1	**Interrupt Table Register** — Contains the vector address for INT 0.
R2	**Break Table Register** — Contains the vector address for break 0.
R3	**Current Context Register** — The two LSBs are the current context register.
R4	**DMEM Comparison Address** — Contains the address at which debug pulse is generated.
R5	**DMEM Comparison Address Mask** — Contains the mask for the DMEM address.
R6	**DMEM Comparison Data** — Contains the data value for which debug pulse is generated.
R7	**DMEM Comparison Data Mask** — Contains the mask for the DMEM data.
R8	**Interrupt Flag** — The LSB in the Interrupt Flag.
R9	**Read/Write Mask** — The two LSBs are the Read mask and the Write mask for R4 to R7.

Fast Context Switching Configuration Using the CPRC

The CPRC and its memory architecture are optimized for the execution of real-time communication tasks, typically multiplexing processing between a number of different tasks (transmit and receive, at a minimum). The CPRC may be configured to incorporate a fast, four-way, context switching feature that replicates the entire CPRC register space four (4) times and can switch from one register set (one context) to another under software control or hardware interrupt.

Thus, actual processing (as opposed to manually saving the contents of one set of registers and then loading another) can begin on a different context in only two cycles. Therefore, these four (4) contexts can be used for debugging, supervisory tasks, event handlers, or other tasks.

Figure 16 CP Context Switching Feature Block Diagram



Fast Context Switching Detail Operations

Using the internal registers, context switching is accomplished two (2) ways:

1 Coprocessor instruction (software):

The software mechanism for executing a context switch is the C-Port Coprocessor Zero instructions. Refer to Table 14 on page 77.

- MTC0 $1 $3
- where $1 specifies the destination context, and where $3 is the source or current context. The contexts have no priority; how they are used is entirely designated by software.

2 Interrupt (hardware):

The hardware interrupt sequence is:

- All interrupts are disabled until an Restore From Exception (RFE) instruction is executed.
- The address of the next instruction to be executed in the interrupted context is saved in K1. Refer to "Interrupts" on page 79.
- Program execution continues with the instruction at the address specified in the interrupt vector.

Interrupts

The CPRC supports four (4) prioritized hardware interrupts, that can be triggered from any bits in the *Event0/Event1* Registers. There are four (4) MIPS-like register sets corresponding to each hardware context, one (1) register of which (K0) is shared by the other contexts.

K1 contains the program counter value and the context number of the interrupted context. These values are used in the execution of the RFE instruction to return to the previously interrupted context.

All interrupts and exceptions transfer control to a location found in the appropriate interrupt or break table. The base address of the interrupt table is specified by the contents of the interrupt table register ($1) in coprocessor zero. The base address of the exception table is specified by the contents of the break table register ($2) in coprocessor zero.

Interrupts are dispatched by a jump to the address equal to ((interrupt number * 8) + (interrupt table register)). Exceptions are dispatched by a jump to the address equal to ((break number * 8) + (break table register)). In addition to the jump, the register context is set to zero and interrupts are disabled. However, exceptions may still occur. Whether a hardware interrupt or an exception, the interrupted routine's register context and its next program counter are saved in K1 of context zero.

The K1 value points at the next instruction to be executed after the interrupt is serviced. RFE is normally used to: (1) resume the instruction flows at this point, (2) restore the proper register context, and (3) restore the Interrupt Enable Flag (IEF) to its value at the time of the interrupt or exception.

Interrupts are not recognized in a branch delay slot. Also, all exceptions fill the delay slot following a change of flow with a NOP instruction.

Interrupts are enabled by setting the IEF which is the LSB of coprocessor zero, Register 8, (Interrupt Flag). Refer to Table 14 on page 77. The IEF is preserved whenever an exception or an interrupt occurs and is restored by the RFE instruction.

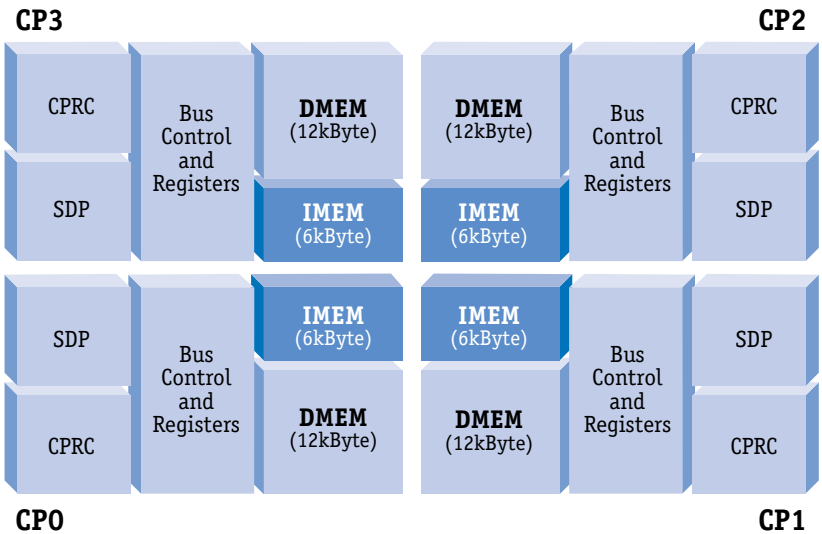
CP Memory (IMEM and DMEM)

The CP has local instruction memory (IMEM) and data memory (DMEM).

Instruction Memory (IMEM)

Each CP shares access to a 24kByte IMEM among a cluster of four (4) adjacent CPs, as shown in Figure 17 on page 80. The IMEM is configured as four (4) sub-arrays, with each CP in the cluster given access to the arrays, one per cycle, in fixed round-robin order. With this simple interleaved scheme, the four (4) adjacent CPRCs can access this memory at nearly full bandwidth.

Figure 17 Local and Shared Memory in a Channel Processor



When adjacent channels are configured to handle similar communication protocols, the large shared memory can contain both CP-specific code and cluster-shared code (such as exception routines).

At initialization time, the 24kByte array can be divided so that each CP gets a dedicated 6kByte sub-array. This array allocation removes all CP contention for IMEM (and also removes the ability to share code among CPs).

CPRC instruction execution outside of the shared local memory space is not supported.

Data Memory (DMEM)

Each CPRC in a cluster has a local 12kByte DMEM and can access the local DMEM of any other CPRC in the cluster with one additional cycle of latency.

The local 12kByte DMEM is organized as 16Byte lines providing 3.2GBps peak bandwidth through a single port. The memory resides in the global address space of the C-5 NP. Local CPRC and Global Bus references use a 4Byte (32bit) access path with zero-wait states in the absence of contention.

The SDP assembles payload data into 16Byte lines and writes it into local DMEM for receive cell/packet processing. For transmit, the SDP reads bytes of payload data from a 16Byte line buffer that is filled from DMEM using a single-cycle, 16Byte access. The Payload Bus controller moves buffers to and from SDRAM through this memory in 64Byte bursts comprised of four (4) consecutive 16Byte accesses.

The SDP and payload transactions have priority over CPRC transactions and use predetermined slots to access DMEM; this provides predictable bandwidth and latency and eliminates the need for extra data buffering. Global references and local CPRC references contend for unused DMEM access slots.

CP Memory Interface Transactions

The CP memory interface transactions are described in Table 15 on page 82.

Table 15 CP Memory Interface Transactions

Transaction Type	Memory Type	Description
Payload Buffer Write	DMEM to SDRAM	The CPRC sets up a Write Control Block (*WrCB0* or *WrCB1*) or Receive Control Block (*RxCB0* or *RxCB1*) and clears the *Avail* bit to cause the bus controller to arbitrate for a payload write. When grant is acquired, the controller transfers 64Bytes in a four-cycle, 16Byte-per-cycle burst from local DMEM directly onto the Payload Bus. Transmission is successful if the receiver acknowledges (ACKs). Otherwise, the bus controller can retry or terminate depending on the programmable controller configuration.
Payload Buffer Read	SDRAM to DMEM	The CPRC sets up a Transmit Control Block (*TxCB0* or *TxCB1*) or Read Control Block (*RdCB0* or *RdCB1*) and clears the *Avail* bit to cause the bus controller to arbitrate for a payload read. When grant is acquired, the controller transfers the read address and makes the request. Transmission is successful if the receiver ACKs. Otherwise, the bus controller can retry or terminate depending on the programmable controller configuration. If the memory controller or queue controller accepts the request, it accesses SDRAM and returns the requested data. Access to DMEM is guaranteed; no acknowledgment is required. The bus controller bus moves 64Bytes in a four-cycle, 16Byte-per-cycle burst directly into the DMEM in consecutive cycles.
Rx SDP Byte Process	External to DMEM	The CPRC sets up a Receive Control Block (*RxCB0* or *RxCB1*) to control the SDP RxByte Processor. When the accumulation buffer fills with byte writes from the RxByte Processor, the 16Byte line is written into the DMEM at the address in the *RxCB0_SDP_Addr register bits [13:0] ByteAddr* field using the next guaranteed Receive access time to DMEM.
Tx SDP Byte Process	DMEM to External	The CPRC sets up a Transmit Control Block (*TxCB0* or *TxCB1*) to control the SDP TxByte Processor. When the TxByte Processor requests a byte read, a 16Byte line buffer is filled from DMEM at the address in the *RxCB0_SDP_Addr* register. Subsequent byte reads from the SDP are serviced from the line buffer. DMEM access uses the next guaranteed Tx access time to DMEM.

Table 15 CP Memory Interface Transactions (continued)

Transaction Type	Memory Type	Description
CPRC Read/Write	DMEM	The CPRC uses word, half-word, and byte loads and stores to access the local DMEM cluster. Local DMEM access for transmit and receive Direct Memory Access (DMA) transactions is guaranteed and takes top priority. CPRC memory references falling within the local DMEM address space receive single-cycle access if memory is available. CPRC memory references falling outside the local DMEM but within the DMEM cluster, take a cycle to arbitrate for the desired DMEM array. When other CPRCs in the cluster have requested a DMEM array, the local CPRC participates in the arbitration. The arbitration scheme ensures that cluster accesses are serviced within the next four cycles that are free of local transmit and receive DMA.
CPRC Read/Write	Global Space	The CPRC uses 32bit word loads and stores to access global memory space. Load operations outside of the cluster DMEM space cause the bus controller to arbitrate for a global transaction. Upon acquisition of grant, the controller drives out the address and request. Transmission of the request is successful if the receiver ACKs. Otherwise, the controller can retry or terminate depending on the programmable controller configuration. Later, the receiver drives back the request data. The CPRC stalls until the load data arrives, so there can only be a single load to global space outstanding per CPRC. Store operations to global memory space dumps address and data into a write buffer in the bus controller. If the write buffer is full, the CPRC process stalls, otherwise the CPRC process continues. A valid write buffer entry causes the bus controller to arbitrate for the global bus. When grant is acquired, the controller drives out address and data. Transmission is successful if the receiver ACKs. Otherwise, the controller can retry or terminate depending on the programmable controller configuration. Since these transactions do not involve the local arrays, DMEM DMA can take place underneath.
CPRC Instruction Fetch	IMEM	Instruction fetch is always local to cluster IMEM. Note: Global memory addresses are not allowed.
Read	IMEM	The read uses the (lwc2) instruction.
Write	IMEM	The write uses the (swc2) instruction.

Table 15 CP Memory Interface Transactions (continued)

Transaction Type	Memory Type	Description
Global Bus Read/Write	DMEM	Global Bus transactions are 32bit word length. From the point-of-view of the target receiving a CPRC Global Memory Space Read/Write, when the bus controller decodes a global read targeted at its local DMEM, it loads a read address latch and either sends back an ACK, if successful, or non-acknowledge (NACK) if the latch is full. The controller arbitrates for DMEM along with cluster DMEM requests. When granted, the controller reads the requested data of the DMEM array into a latch, then arbitrates for the bus. When the bus access is granted, the read data is returned to the requester which must ACK. When the bus control indicates a global write targeted at the local DMEM, the bus controller loads a write address and data latch and either sends back an ACK, if successful, or NACK if the latch is full. The controller arbitrates for DMEM taking the next available cycle to write the data into the array.
CPRC Read/Write	Configuration Space	Global configuration registers maintained on a per CP basis are addressed in global memory space. The CPRC reads and writes the registers over its 32bit data bus using word, half-word, and byte load and store operations. Access is guaranteed since these transactions do not involve the local arrays.
Global Read/Write	Configuration Space	Global access of configuration registers follows the same timing as global access of DMEM.

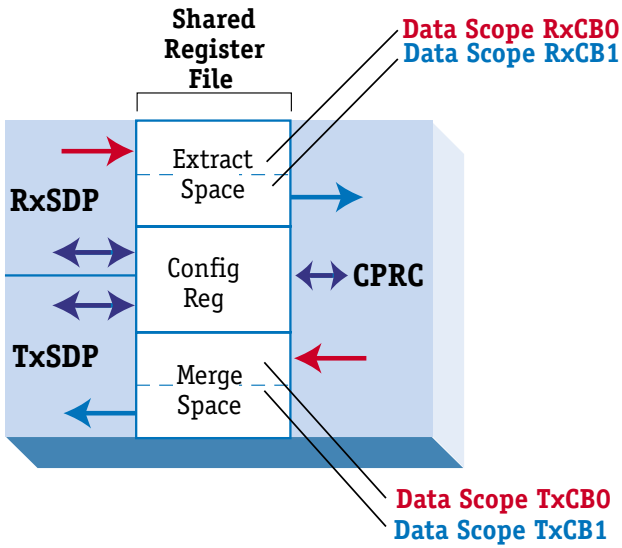
DataScope Purpose

The CP architecture provides access to two (2) sets of packet headers and data fields (*Datascope0* and *Datascope1*) to enable a unique feature called *data scoping*. Data scoping allows overlap of CPRC processing tasks while receiving and transmitting packets. Each data scope provides the CPRC with a coherent view of an individual packet on reception or transmission, including DMA parameters, *Extract* or *Merge* registers, and table lookup results.

The exact contents of the *Extract* and *Merge* registers are determined by microcode. Refer to the *C-Ware Microcode Programming Guide for details*.

Data scopes also eliminate the need for the CPRC program itself to manage the coherency of these disparate operations, allowing the construction of a simple, efficient two-stage software pipeline model. There are a total of four (4) data scopes available, two (2) for receive (Receive Control Blocks *RxCB0* and *RxCB1*) and two (2) for transmit (Transmit Control Blocks *TxCB0* and *TxCB1* registers). A diagram of a CP depicting the receive and transmit data scopes is shown in Figure 18 on page 85.

Figure 18 Four (4) Data Scopes Between the CPRC and SDPs



A hardware *receive* data scope is made up of Extract Space, an SDP Receive status register, and a Receive Control Block (*RxCB*). A hardware *transmit* data scope is made up of Merge Space, an SDP Transmit status register, and a Transmit Control Block (*TxCB*).

Data Scope Detail Operations

Initially, the RxSDP brings in payload, extracts fields and writes them to Extract Space, and launches table lookups on the Ring Bus in *Datascope0*. Hardware directs SDP DMEM writes to *RxCB0*, Extract Space writes to *RxSDP0_Ext0* to *RxSDP0_Ext31*, and status updates to *RxCtl0_Status*. Subsequently, the RxSDP signals that it has finished processing a cell/packet, triggering the hardware to switch to *Datascope1*.

Then the hardware directs SDP DMEM writes to *RxCB1*, Extract Space writes to *RxSDP1_Ext0* to *RxSDP1_Ext31*, and status updates to *RxCtl1_Status*. At the end of this cell/packet, hardware switches back to scope 0. The SDP is required to test the status bits in RxStatus to be sure the new scope is ready before processing the next cell/packet.

The CPRC must monitor both *RxCtln_Status* registers to track SDP processing. After the SDP finishes a scope, the CPRC must:

- Examine and possibly remove relevant data from the associated Extract Space.
- Examine the *RxCBn* to confirm that the payload DMA finished and reprogram the *RxCBn* if necessary.
- Examine and possibly remove relevant data from the Ring Bus response registers and reset the ownership bits.
- Update the *RxCtln_Status* to make the scope available to the SDP.

The TxSDP and CPRC operate in a similar manner to transmit the datascopes.

CP Configuration Space

Each CP has memory-mapped Configuration Space that contains a number of registers. Refer to Table 16 on page 88 for a list of CP registers by function. The CPRC uses these registers to communicate with the SDP, the bus controllers, and the XP.

Address Mapping of the CPs

Since the CP configuration space is duplicated for each CP (CP0 to CP15), the address listed in the memory maps and register descriptions begins with 0x*BC***n***0* where n should be replaced with the appropriate offset for the particular CP. Refer to Chapter 1 for addressing details.

The Configuration Space provides a 1MB block or segment of Configuration Space for each CP. Specific registers are located at offsets from each CP's Configuration Space base address.

Figure 19 CP Configuration Space Memory Map

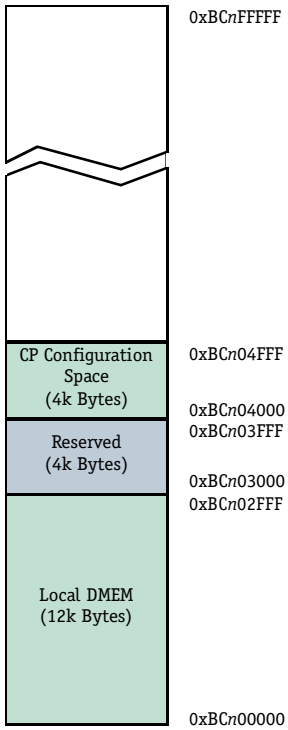


Table 16 CP Registers by Function

CP Function	Specific Registers
DMEM	See "Data Memory (DMEM)" on page 81
Rx Extract	RxSDP0_Ext0 to RxSDP0_Ext15 RxSDP1_Ext0 to RxSDP1_Ext15
Rx Control Blocks	RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr, RxCB0_Status, RxCB1_Sys_Addr, RxCB1_Ctl, RxCB1_DMA_Addr, RxCB1_SDP_Addr, RxCB1_Status
Tx Merge	TxSDP0_Merge0 to TxSDP0_Merge15 TxSDP1_Merge0 to TxSDP1_Merge15
Tx Control Blocks	TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr, TxCB0_Status, TxCB1_Sys_Addr, TxCB1_Ctl, TxCB1_DMA_Addr, TxCB1_SDP_Addr, TxCB1_Status
Wr Control Blocks	WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB1_Sys_Addr, WrCB1_Ctl, WrCB1_DMA_Addr
Rd Control Blocks	RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB1_Sys_Addr, RdCB1_Ctl, RdCB1_DMA_Addr
Ring Bus Tx Message Control	TxMsg0_Ctl, TxMsg1_Ctl, TxMsg2_Ctl, TxMsg3_Ctl, TxMsg0_Data_H, TxMsg0_Data_L, TxMsg1_Data_H, TxMsg1_Data_L, TxMsg2_Data_H, TxMsg2_Data_L, TxMsg3_Data_H, TxMsg3_Data_L
Ring Bus Rx Response Control	RxResp0_Ctl, RxResp1_Ctl, RxResp2_Ctl, RxResp3_Ctl, RxResp4_Ctl, RxResp5_Ctl, RxResp6_Ctl, RxResp7_Ctl, RxResp0_DataH, RxResp0_DataL, RxResp1_DataH, RxResp1_DataL, RxResp2_DataH, RxResp2_DataL, RxResp3_DataH, RxResp3_DataL, RxResp4_DataH, RxResp4_DataL, RxResp5_DataH, RxResp5_DataL, RxResp6_DataH, RxResp6_DataL, RxResp7_DataH, RxResp7_DataL
Ring Bus Rx Message Control	RxMsg_Ctl, RxMsg_FIFO
SONET Rx Control	Rx_SONETH0 to Rx_SONETH31
SONET Tx Control	Tx_SONETH0 to Tx_SONETH31
SDP Rx Control	RxCtl_ByteSeq0, RxCtl_ByteSeq1, RxCtl_SyncSeq, RxCtl_BitSeq0, RxCtl_BitSeq1
SDP Tx Control	TxCtl_ByteSeq0, TxCtl_ByteSeq1, TxCtl_BitSeq0, TxCtl_BitSeq1
CP Mode Configuration	CP_Mode0, CP_Mode1, SDP_Mode2, SDP_Mode3, SDP_Mode4, SDP_Mode5, Debug_Mode, PIN_Mode

Table 16 CP Registers by Function (continued)

CP Function	Specific Registers
Queue Status	Queue_Status0, Queue_Status1, Queue_Status2, Queue_Status3, Queue_Update0, Queue_Update1, Queue_Update2, Queue_Update3
Miscellaneous	Event_Timer, Cycle_Counter_H, Cycle_Counter_L
Event and Interrupt	Event0, Event0, Event_Mask1, Event_Mask1, Event_Access, Mask_Access, Interrupt_Mask0, Interrupt_Mask1, SONET_Event, SONET_Mask

For complete details about specific registers go to their reference. Refer to "CP Registers" on page 378.

Understanding CP Functions

The following is a discussion of the CP functions and the registers associated with each function.

Extract Space

Configuration Space contains 64Bytes of Extract Space per datascope (Datascope0/Datascope1) for passing fields extracted from the receive data stream (by the SDP RxByte programmable processor) to the CPRC. The RxByte programmable processor performs byte-wide write operations to the Extract Space by specifying the configuration register destination commands in microcode.

The RxByte programmable processor cannot read the Extract Space registers.

The CPRC accesses the memory-mapped Extract Space using load and store instructions. The CPRC can write to the Extract Space registers, but only during initialization and test periods when the *SDP_Mode3* register bit [30] *RxEnable* field is cleared.

The data format for the Extract Space is defined by agreement between the CPRC program and the RxByte programmable processor microcode. Refer to Table 17 on page 90 for Extract Space registers.

Table 17 Extract Space Registers

Register Name	Purpose	Address	Details
RxSDP0_Ext0 to RxSDP0_Ext15	Used to pass fields extracted from the receive data stream by the RxSDP to the CPRC. These registers are used only for receive data scope0.	0xBC*n*04000 to 0xBC*n*0403C	See "RxSDP0_Ext0 to RxSDP0_Ext15 Registers (CP Rx Extract Space0 Function)" on page 383
RxSDP1_Ext0 to RxSDP1_Ext15	Same as registers *RxSDP0_Ext0* to *RxSDP0_Ext15*, except for data scope1.	0xBC*n*04200 to 0xBC*n*0423C	See "RxSDP1_Ext0 to RxSDP1_Ext15 Registers (for Datascope1)" on page 383

Merge Space

Configuration Space contains 64Bytes of Merge Space per datascope (Datascope0/Datascope1) for passing fields from the CPRC to the SDP TxByte programmable processor to merge in with the transmit data stream. The CPRC accesses the memory-mapped Merge Space using load and store instructions. The TxByte programmable processor performs byte-wide read operations from the Merge Space by specifying the configuration register source in microcode. The data format for the Merge Space registers is defined by the CPRC process and the SDP firmware. Refer to Table 18 on page 90 for Merge Space registers.

The TxByte programmable processor cannot write to the Merge Space registers.

Table 18 Merge Space Registers

Register Name	Purpose	Address	Details
TxSDP0_Merge0 to TxSDP0_Merge15	Used to pass fields from the CPRC to the TxSDP to merge in with the transmit data stream. These registers are used only for transmit data scope0.	0xBC*n*04100 to 0xBC*n*0413C	See "TxSDP0_Merge0 to TxSDP0_Merge15 Registers (CP Tx Merge Space0 Function)" on page 383
TxSDP1_Merge0 to TxSDP1_Merge15	Same as registers *TxSDP0_Merge0* to *TxSDP0_Merge15*, except for data scope1.	0xBC*n*04300 to 0xBC*n*0433C	See "TxSDP1_Merge0 to TxSDP1_Merge15 Registers (for Datascope1)" on page 384

Control Block Registers

Configuration Space includes eight (8) sets of control registers called Control Blocks (CBs) that govern DMA operations to and from DMEM. The CPRC sets up the control registers to perform four (4) types of DMEM DMA operations:

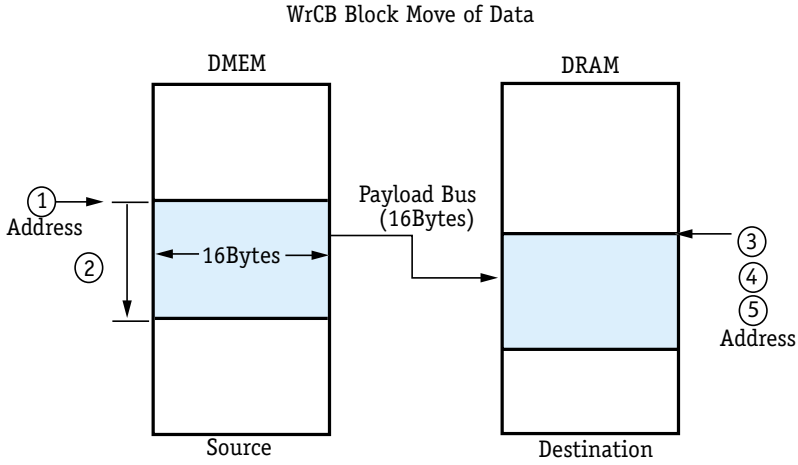
- Write Control Block (WrCB0_ , WrCB1_)
- Read Control Block (RdCB0_ , RdCB1_)
- SDP RxByte Processor Receive Control Block (RxCB0_ , RxCB1_)
- SDP TxByte Processor Transmit Control Block (TxCB0_ , TxCB1_)

Write Control Blocks (WrCB0_ , WrCB1_)

Two Write Control Blocks (WrCB0_ and WrCB1_) provide the capability for general purpose write tasks, such as Buffer Transfers, QMU enqueues and BTag writes. These tasks are DMA operations of programmable length that move data from DMEM over the Payload Bus in bursts of four (4) cycles with 16 bytes per burst.

Figure 20 on page 92 shows a Buffer Transfer. In general, data is moved from DMEM (the source) to SDRAM (the destination). Specifically, moving data starting at the *LineAddr* (1) location inside DMEM with a *Length* (2) to SDRAM beginning at the (*PoolID, BTag* and *Offset* (3,4,5)) location. These individual fields that are used to set up the details of the block move make up parts of these registers: WrCB*n*_Sys_Addr, WrCB*n*_Ctl and WrCB*n*_DMA_Addr.

Figure 20 DMA Operation (Buffer Transfer) Using WrCB*n*_ Registers



Register	Fields	
WrCBn_Sys_Addr	BTag ④	Offset ⑤
WrCBn_Ctl		Length ②
WrCBn_DMA_Addr	PoolID ③	LineAddr ①

Buffer Transfer Setup Using WrCBn_Sys_Addr, WrCBn_Ctl and WrCBn_DMA_Addr:
To set up this single contiguous data transfer, the CPRC writes a system address and a line address for DMEM to the WrCB. The length is written by the CPRC to the desired transfer length.

To perform Buffer Transfers involves setting the bits for WrCB0_Sys_Addr (0xBC*n*04400), WrCB0_Ctl (0xBC*n*04404) and WrCB0_DMA_Addr (0xBC*n*04408).

To set up a general contiguous data transfer, the CPRC process must do the following:

1 Ensure that the WrCB0 is available by testing that *WrCB0_Ctl* bit [31] *Avail* field=1.

2 Write *WrCB0_Sys_Addr* with the system address to be written in the form of:
WrCB0_Sys_Addr bits [31:16] *BTag* field = BTag, and
WrCB0_Sys_Addr bits [15:4] *Offset* field = Offset
(Offset is a 16Byte starting buffer offset, typically equal to 0, or aligned to a 64Byte boundary).

3 Write the *Pool ID* portion of the system address into *WrCB0_DMA_Addr* bits [20:16] *PoolID* field. Write *WrCB0_DMA_Addr* bits [13:0] *LineAddr* field with the location of a buffer in DMEM, typically aligned on a 64Byte boundary. This is the location that the DMA engine uses to begin transferring data out of DMEM to SDRAM.

4 Write *WrCB0_Ctl* with *WrCB0_Ctl* bits [13:4] *Length* field equal to the number of bytes to be transferred, and *WrCB0_Ctl* bit [31] *Avail* field equal to 0, and *WrCB0_Ctl* bit [29] *Modulo64* equal to 0 to cause the *WrCB0_Sys_Addr* bits [15:4] *Offset* field and *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field to increment during the DMA for a contiguous block transfer.

For complete details about specific registers go to their reference. Refer to: "WrCB0_Sys_ Addr Register (CP Wr Control Block0 Function)" on page 390, "WrCB0_Ctl Register (CP Wr Control Block0 Function)" on page 391, and "WrCB0_DMA_Addr Register (CP Wr Control Block0 Function)" on page 392. You also have the following registers available for Control Block1: WrCB1_Sys_Addr, WrCB1_Ctl and WrCB1_DMA_Addr that function in the same manner.

Buffer Transfer Operations Using WrCBn_Sys_Addr, WrCBn_Ctl and WrCBn_DMA_Addr: An availability bit indicates DMA or CPRC control of the block of DMEM. When set, the CPRC controls the block; when clear, the DMA engine controls the block and is free to transfer out of it.

Clearing the availability bit, *WrCB0_Ctl* bit [31] *Avail* field, the CPRC process initiates a data transfer from DMEM beginning at the line addressed by *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field to SDRAM beginning at the system address in *WrCB0_Sys_Addr*. The DMA engine transfers payload out of DMEM in bursts, decrementing *WrCB0_Ctl* bits [13:0] *Length* field by 64Bytes for each burst. Transfer continues until the *WrCB0_Ctl* bits [13:0] *Length* field equals 0 (at which time the DMA engine sets *WrCB_Ctl* bit [31] *Avail* field, thus returning control of the WrCB back to the CPRC process).

Initiating transfers with *WrCB0_Ctl* bits [13:0] *Length* field equal to 0 causes a single 64Byte transfer. If a 4-cycle, 64Byte transfer is started with *WrCB0_Ctl* bits [13:0] *Length* field less than 64 bytes, only the number of 16Byte lines needed to transmit the whole length actually get written into SDRAM and the *Length* field is set to 0 after the burst.

If the *WrCB0_Sys_Addr* bits [15:4] *Offset* field is aligned to a 64Byte boundary, a contiguous 64Byte block of DRAM is written. If the *WrCB0_Sys_Addr* bits [15:4] *Offset* is 64Byte unaligned, the DRAM block is written in a wrapped fashion which is typically not useful. Typically, contiguous transfers start with aligned offsets.

WrCB0_DMA_Addr bits [13:4] *LineAddr* field increments for each of the four 16Byte-per-cycle transfers to provide an address into DMEM. Typically, *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field starts at a 64Byte aligned address. *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field can start unaligned, but the resulting wrap behavior is not useful. Transfers with *Offset* unaligned make the most sense if the *Length* and *Offset* fields are set so that the resulting SDRAM completes a block of SDRAM, but does not wrap. For example, if *WrCB0_Sys_Addr* bits [15:4] *Offset* field= 0x0010 and *WrCB0_Ctl* bits [13:4] *Length* field= 0x0030, then the DMA moves three 16Byte lines from DMEM[1:3] to SDRAM[1:3]. The burst wraps to DMEM[0] and SDRAM[0], but the write is inhibited. If *WrCB0_Ctl* bit [29] *Modulo64* field equal to 0, *WrCB0_DMA_Addr* bits [13:4] *LineAddr field* and *WrCB0_Sys_Addr* bits [15:4] *Offset* field increment by 64 for each 64Byte burst.

Initiating transfers with the modulo64, W*rCB0_Ctl* bit [29] *Modulo64* field, equal to 1prevents an update of the *WrCB0_Sys_Addr* bits [15:4] *Offset* field and causes *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field to increment modulo 64Bytes, effectively returning *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field to wrap back to the starting value after a 4-cycle burst. This feature is useful for writes to the QMU or BMU when the system address contains a command, not an address. The WrCB can be used again without resetting the *WrCB0_Sys_Addr* bits [15:4] *Offset* field and *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field.

The CPRC process can read the state of the DMA machine from *WrCB0_Ctl* bits [17:16] *State* field. The *WrCB0_Ctl* word generally should not be written by the CPRC process when hardware is operating (that is, *WrCB_Ctl* bit [31] *Avail* field= 0). On a write to *WrCB0_Ctl*, bits [17:16] *State* field is only updated if bit [31] *Avail* field =1.

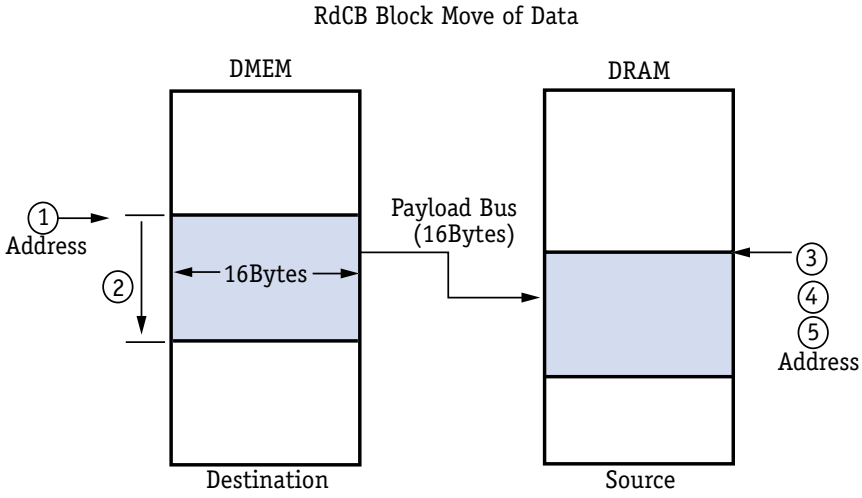
When DMA transaction requests receive a no-acknowledge (NACK) on the Payload Bus, the bus controller retries the request up to 16 (maximum) times before reporting a bus error. The bus error sets status in *WrCB0_Ctl* bits [27:24] *Error* field with an encoding, this generates an event for the *Event0* or *Event1* register, and immediately terminates the transfer by setting *WrCB0_Ctl* bit [31] *Avail* field. When the *WrCB0_Ctl* bit [30] *NoRetry* field is set, the bus controller does not retry, and reports the bus NACK immediately as a bus error.

Read Control Blocks (RdCB0_ , RdCB1_)

Two Read Control Blocks (RdCB0_ and RdCB1_) provide the capability for general purpose read tasks, such as Buffer Transfers, QMU dequeues and BTag allocates. These tasks move data across the Payload Bus into DMEM in bursts of four (4) cycles with 16Bytes per cycle.

Figure 21 on page 95 shows a Buffer Transfer. In general, you are moving data from SDRAM (the source) to DMEM (the destination). Specifically, you are moving data starting at the (*PoolID*, *BTag* and *Offset* (3,4,5)) location inside DRAM to DMEM at the *LineAddr* (1) location with a *Length* (2). These individual fields that are used to set up the details of the block move make up parts of these registers: RdCB*n*_Sys_Addr, RdCB*n*_Ctl and RdCB*n*_DMA_Addr.

Figure 21 DMA Operation (Buffer Transfer) Using RdCB*n*_ Registers



Register	Fields	
RdCBn_Sys_Addr	BTag (4)	Offset (5)
RdCBn_Ctl		Length (2)
RdCBn_DMA_Addr	PoolID (3)	LineAddr (1)

Buffer Transfer Setup Using RdCBn_Sys_Addr, RdCBn_Ctl and RdCBn_DMA_Addr:
To set up this single, contiguous data transfer, the CPRC writes a system address (consisting of a *PoolID*, *BTag*, and *Offset* for buffer memory in SDRAM), a line address for DMEM, to the RdCB. The length is written by the CPRC to be the desired transfer length.

To perform Buffer Transfers involves setting the bits for RdCB0_Sys_Addr (0xBC*n*04420), RdCB0_Ctl (0xBC*n*04424) and RdCB0_DMA_Addr (0xBC*n*04428).

To set up a general contiguous data transfer, the CPRC process must do the following:

1. Ensure that the RdCB0 is available by reading *RdCB0_Ctl* bit [31] *Avail* field =1.
2. Write *RdCB0_Sys_Addr* with the system address to be read, in the form of:
   RdCB0_Sys_Addr bits [31:6] BTag field = BTag, and
   RdCB0_Sys_Addr bits [15:4] Offset field = Offset
   (Offset is a 16Byte starting buffer offset, typically equal to 0, or aligned to a 64Byte boundary.
3. Write the *PoolID* portion of the system address to be read into *RdCB0_DMA_Addr* bits [20:16] *PoolID* field. Write *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field with the location of a buffer in DMEM, typically aligned on a 64Byte boundary. This is the DMEM location that the DMA engine uses to begin writing data from SDRAM.
4. Write *RdCB0_Ctl* with *RdCB0_Ctl* bits [13:0] *Length* field equal to the number of bytes to be transferred, *RdCB0_Ctl* bit [31] *Avail* field equal to 0, and *RdCB0_Ctl.* bit [29] *Modulo64* equal to 0 to cause the *RdCB0_Sys_Addr* bits [15:4] *Offset* field and *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field to increment during the DMA for a contiguous block register.

For complete details about specific registers go to their reference. Refer to: "RdCB0_Sys_Addr Register (CP Rd Control Block0 Function)" *on page 393*, "RdCB0_Ctl Register (CP Rd Control Block0 Function)" *on page 394*, and "RdCB0_DMA_Addr Register (CP Rd Control Block0 Function)" *on page 395*. *You also have the following registers available for Control Block: RdCB1_Sys_Addr, RdCB1_Ctl and RdCB1_DMA_Addr that function in the same manner.*

Buffer Transfer Operations Using RdCB*n*_Sys_Addr, RdCB*n*_Ctl and RdCB*n*_DMA_Addr: An availability bit indicates that the block of DMEM is controller by to either DMA or CPRC. When set, the CPRC controls the block; when clear, the DMA engine controls the block and is free to transfer into it.

Clearing the available bit, *RdCB0_Ctl* bit [31] *Avail* field, the CPRC process initiates a 64Byte data transfer from SDRAM beginning at the system address consisting of (*RdCB0_Sys_Addr* bits [31:16] *BTag* field, *RdCB0_Sys_Addr* bits [15:4] *Offset* field, *and RdCB0_DMA_Addr* bits [20:16] *PoolID* field to DMEM beginning at the 16Byte line addressed by *RdCB0_DMA_Addr* bits [15:4] *LineAddr* field. The SDRAM DMA engine transfers payload out of SDRAM in a 4-cycle, 16Byte-per-cycle burst, decrementing *RdCB0_Ctl* bits [13:0] *Length* field by 64 for each burst.

Transfer continues until *RdCB0_Ctl* bits [13:0] *Length* field equals 0 (at which time the DMA engine sets *RdCB0_Ctl* bit [31] *Avail* field, thus returning control of the RdCB back to the CPRC process).

Initiating transfers with *RdCB0_Ctl* bits [13:0] *Length* field equal to 0 causes a single 64Byte transfer. If a 4-cycle, 64Byte transfer is started with *RdCB0_Ctl* bits [13:4] *Length* field < 64Bytes, only the number of 16Byte lines needed to satisfy the whole length according to the *Length* field actually get read from SDRAM. Unpredictable data completes the full 64Bytes returned, and the Length field after the burst is set to 0.

If the *RdCB0_Sys_Addr* bits [13:4] *Offset* field is aligned to a 64Byte boundary, a contiguous 64Byte block of SDRAM is read. If the Offset is 64Byte unaligned, the SDRAM block is read in a wrapped fashion which is generally not useful. Typically, contiguous transfers start with aligned offsets.

RdCB0_DMA_Addr bits [15:4] *LineAddr* field increments for each of the four 16Byte-per-cycle transfers to provide an address into DMEM. Typically, R*dCB0_DMA_Addr* bits [13:4] *LineAddr* field starts at a 64Byte aligned address. *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field can start unaligned, but the resulting wrap behavior is not useful. The burst read from SDRAM always returns 64Bytes. *RdCB0_Sys_Addr* bits [15:4] *Offset* field increments by 64 each 64Byte burst.

Initiating transfers with the modulo64, *RdCB0_Ctl* bit [29] *Modulo64,* equal to 1prevents updates of the *RdCB0_Sys_Addr* bits [15:4] *Offset* field and causes *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field to increment modulo 64Bytes, effectively returning the *LineAddr* to wrap back to the starting value. This feature is useful for reads from the QMU or BMU where the system address contains a command, not an address. The RdCB can be used again without resetting the *Offset* and *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field.

The CPRC process can read the state of the DMA machine at any time from *RdCB0_Ctl* bits [17:16] *State* field. The *RdCB0_Ctl* word generally should not be written by the CPRC process when hardware is operating (that is, *RdCB0_Ctl* bit [31] *Avail* field=0). On a write to *RdCB0_Ctl* bits [17:16] *State* field are only updated if bit [31] *Avail* field=1.

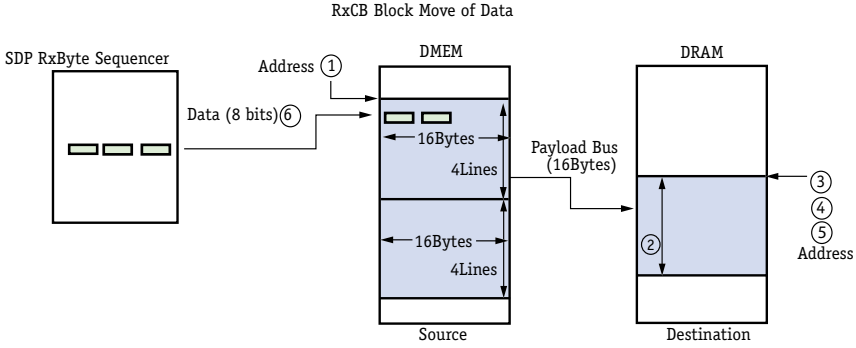
When DMA transaction requests receive a no-acknowledge (NACK) on the Payload Bus, the bus controller retries the request up to 16 times before reporting a bus error. The bus error sets status in *RdCB0_Ctl* bits [27:24] *Error* field, generates an event for the Event register, and immediately terminates the transfer by setting *RdCB0_Ctl* bit [31] *Avail* field. When the *RdCB0_Ctl bit [30] NoRetry* field is set, the bus controller does not retry, and reports the bus NACK immediately as a bus error.

SDP RxByte Processor Receive Control Blocks (RxCB0_, RxCB1_)

The CPRC controls receive operations using the two (2) Receive Control Blocks (RxCB0_ and RxCB1_) that handle the payload write operation much like the (WrCB0 and WrCB1), but add the capability to control SDP RxByte Processor writes to DMEM. The SDP RxByte Processor directs the incoming data stream into DMEM a byte at a time. In hardware, the byte stream is accumulated into 16Byte lines that are written to DMEM in a single cycle. Using a RxCB, the CPRC can set up a payload receive path from the RxByte Processor to DMEM to SDRAM. Payload data movement happens in hardware with no further CPRC control.

Figure 22 on page 98 shows a Buffer Transfer. In general, you are moving data from SDP (the source) to SDRAM (the destination). Specifically, you are moving data from the SDP RxByte Sequencer in 8bit units using *ByteAddr* (6) and counting the bytes moved with a *Length* (2) starting at the *LineAddr* (1) location inside DMEM to SDRAM at the (*PoolID, BTag* and *Offset* (3,4,5)) location. These individual fields are used to setup the details of the block move and make up parts of these registers: RxCB*n*_Sys_Addr, RxCB*n*_Ctl, RxCB*n*_DMA_Addr, and RxCB*n*_SDP_Addr.

Figure 22 DMA Operation (Buffer Transfer) Using RxCB*n* Registers



Register	Fields	
RdCBn_Sys_Addr	BTag 4	Offset 5
RdCBn_Ctl		Length 2 Counts the Bytes moved from the sequencer)
RdCBn_DMA_Addr	PoolID 3	LineAddr 1
RxCBn_SDP_Addr		ByteAddr 6

Buffer Transfer Setup Using RxCB*n*_Sys_Addr, RxCB*n*_Ctl RxCB*n*_DMA_Addr and RxCB*n*_SDP_Addr:
To perform Buffer Transfers involves setting the bits for RxCB0_Sys_Addr (0xBC*n*04080), RxCB0_Ctl (0xBC*n*4084), RxCB0_DMA_Addr (0xBC*n*04088), and RxCB0_SDP_Addr (0xBC*n*0408C).

To set up a typical single receive operation, the CPRC must do the following:

1. Ensure that the RxCB0 is available by testing that *RxCB0_Ctl* bit [31] *Avail* field = 1.
2. Write *RxCB0_Sys_Addr* with the system address to be written in the form of:
   RxCB0_Sys_Addr bits [31:16] *BTag* field = *BTag*, and
   RxCB0_Sy_Addr bits [15:4] *Offset* field = Offset
   (Offset is a 16Byte starting buffer offset, typically aligned to a 64Byte boundary).
3. Write the Pool ID portion of the system address to be written into *RxCB0_DMA_Addr* bits [20:16] *PoolID* field. Write *RxCB0_DMA_Addr* bit [13:4] *LineAddr* field with a 16Byte address in DMEM, typically aligned on a 128Byte boundary. This is the location that the DMA engine uses to begin transferring data out of DMEM to SDRAM.
4. Write *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field with a 16Byte address in DMEM, typically the same value (buffer) as *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field. This is the location that the SDP RxByte Processor uses to begin transferring bytes into DMEM.
5. Write the *RxCB0_Ctl* bits [15:0] *RxLength* field to zero, clear *RxCB0_Ctl* bit [23] *Own1* field and *RxCB0_Ctl* bit [22] *Own0* field to give ownership of the double buffer to the SDP DMA (rather than the SDRAM DMA) engine, and clear *RxCB0_Ctl* bit [31] *Avail* field that starts the SDRAM DMA engine.

For complete details about specific registers go to their reference. Refer to: "RxCB0_Sys_Addr Register (CP Rx Control Block0 Function)" on page 384, "RxCB0_Ctl Register (CP Rx Control Block0 Function)" on page 385, "RxCB0_DMA_Addr Register (CP Rx Control Block0 Function)" on page 388 and "RxCB0_SDP_Addr Register (CP Rx Control Block0 Function)" on page 389. *You also have the following registers available for Control Block1: RxCB1_Sys_Addr, RxCB1_Ctl, RxCB1_DMA_Addr and RxCB1_SDP_Addr that function in the same manner.*

Buffer Transfer Operations Using RxCB*n*_Sys_Addr, RxCB*n*_Ctl RxCB*n*_DMA_Addr and RxCB*n*_SDP_Addr:

The DMA engine always uses 128Bytes double buffering (that is, two (2) sequential 64Byte DMEM buffers (16bytes wide x 4lines high)) to handle payloads of arbitrary length. The transfer of these buffers can be individually controlled by the CPRC. Typically, both buffers are enabled by the CPRC via the *RxCB0_Ctl* bit [23] *Own1* field and *RxCB0_Ctl* bit [22] *Own0* field and the DMA initiates transfers whenever the next 64Byte block within the buffer becomes available. Ownership bits track the status of the two contiguous 64Byte blocks in DMEM.

RxCB0_DMA_Addr bits [13:4] *LineAddr* field and *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field typically point to the 128Byte aligned buffer. Increments of *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field and *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field are done modulo 128 so that writing or reading the last line in the buffer causes the pointers to wrap back to the start of the buffer.

RxCB0_Ctl bit [23] *Own1* field and *RxCB0_Ctl* bit [22] *Own0* field track the ownership of the two (2) 64Byte blocks in the 128Byte buffer. By clearing the ownership bits initially, the CPRC allows the SDP RxByte Processor to write into the DMEM buffers. When *RxCB0_SDP Addr* bits [15:0] *ByteAddr* field reaches a 64Byte boundary, the hardware sets the corresponding ownership bit to indicate that the SDRAM DMA engine now owns the block. It initiates a 64Byte transfer to SDRAM as soon as possible, incrementing *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field by 16 for each of the four (4), 16Byte-per-cycle transfers to provide an address into DMEM. It also adds 64 to *RxCB0_Sys_Addr* bits [15:4] *Offset* field to update the SDRAM address. When the DMA is complete, the *RxCB0_Ctl* bit [23] *Own1* field or *RxCB0_Ctl* bit [22] *Own0* field is cleared to allow the SDP to reuse that half of the double buffer.

Thus, *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field and *RxCB0_DMA_Addr* bits [13:4] *LineAddr field* act as a pair, following one another through a payload transfer. *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field leads as the SDP RxByte Processor fills DMEM with payload bytes. When a 64Byte buffer is full, an SDRAM DMA transaction uses the lagging *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field to move the buffer to SDRAM. The SDP RxByte Processor forces a line write when signaling end-of-frame by setting *RxCtl0_Status* bit [31] *Avail* field, which must happen exclusive of an SDP byte write. Unwritten bytes at the end of the 16Byte line are undefined. This clears *RxCB0_SDP_Addr* bits [6:0] within the *ByteAddr* field, clears *RxCB0_DMA_Addr* bits [6:4] within the *LineAddr* field, and sets *RxCB0_Ctl* bit [31] *Avail* field to realign the double buffer and to return control of the RxCB to the CPRC.

In hardware, an accumulation buffer assembles sequential SDP RxByte Processor writes until a 16Byte DMEM line boundary is crossed. This triggers a line write to DMEM at the

address in *RxCB0_SDP_Addr* bits [15:4] *ByteAddr* field if the associated ownership bit allows it. There are no hardware interlocks that assure the RxCB is configured before accepting SDP byte writes. The CPRC process must be sure to configure *RxCB0_SDP_Addr* bits [15:0] *ByteAddr* field and both *Own0* field bit [22] and *Own1* field bit [23] before passing the datascope to the SDP. For some applications, the CPRC process can choose to write *RxCB0_DMA_Addr*, *RxCB0_Sys_Addr*, and *RxCB0_Ctl* bit [31] *Avail* field (using a byte operation later). For each byte transferred, hardware increments the *RxCB0_Ctl* bits [15:0] *RxLength* field to reflect the total number of bytes in the receive payload.

The RxCB can be used with *RxCB0_Sys_Addr* bits [15:4] *Offset* field and *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field pointing to 16Byte addresses that are not 64Byte aligned. In this case, the SDRAM DMA inhibits any writes that wrap within the SDRAM block. This can be used to transfer partial blocks from DMEM to SDRAM in assembly operations. After a 4-cycle burst, the *RxCB0_Sys_Addr* bits [15:4] *Offset* field is always set to the next aligned 64Byte block.

There are eight (8) bits in each of the Out-Of-Band fields (OOB). Refer toTable 19 on page 101.They (OOB) are located in the *TxCB0_SDP_Addr* register, bits [31:24] are for OOB0 and bits [23:16] are for OOB1. Eight (8) Out-Of-Band bits are transferred to SDRAM along with every 64Byte payload transfer. The 7th Bit of OOB*n* indicates that the SDP encountered an error receiving this frame. The 6th Bit of OOB*n* indicates that this block of 64Bytes contains the End-of-Packet (EOP), and when the 6th Bit is set, the remaining six bits in the OOB*n* field indicate the position of the last byte. These (OOB*n*) bits get transferred to SDRAM automatically, based on SDP error signals and *RxCB0_Ctl* bits [15:0] *RxLength* field. During test, the RxCB can be used to write the OOB bits. Writing a buffer in DMEM and setting up *RxCB0_Sys_Addr*, *RxCB0_DMA_Addr* bits [13:4] *LineAddr* field, *RxCB0_Ctl* bit [29] *EOP* field, and *RxCB0_Ctl* bits [15:0] *RxLength* field determine what payload and OOB*n* bits get written to SDRAM.

Test software can force the transfer by clearing *RxCB0_Ctl* bit [31] *Avail* field and setting the appropriate *RxCB0_Ctl* bits [23:22] *Own1* or *Own0* bit.

Table 19 Out-of-Band Bits and Functions

Side-Car Bits	Function
7	Packet Error (also asserted for double ECC errors)
6	End of Packet (EOP)
5:0	Encoded Value (for valid Bytes) Legal Range= 0 to 63

The CPRC process can read the state of both DMA engines and the EOP status at any time from *RxCB0_Ctl* bit [19] *SDP State* field and the *RxCB0_Ctl* bits [17:16] *DMA State* field. The *RxCB0_Ctl* word generally should not be written by the CPRC process when hardware is operating (that is, when *RxCB0_Ctl* bit [31] *Avail* field=0). On a write to *RxCB0_Ctl* bit [19] *SDP State* field, bits [17:16] *DMA State*, and bit [29] *EOP* fields are only updated if bit [31] Avail=1. Additionally, *RxCB0_Ctl* bit [29] *EOP* field is not updated if bit [28] *Protect_EOP*=1.

When DMA transaction requests receive no-acknowledge (NACK) on the Payload Bus, the bus controller retries the request up to 16 (maximum) times before reporting a bus error. The bus error sets status in *RxCB0_Ctl* bits [27:24] *Error* field, generates an event for the *Event 1*, and immediately terminates the SDRAM transfer by setting *RxCB0_Ctl* bit [31] *Avail* field. When the *RxCB0_Ctl* bit [30] *NoRetry* field is set, the bus controller does not retry, and reports the bus NACK immediately as a bus error.

Receive payload can be recycled back through the SDP RxByte Processor using a configuration option and explicit CPRC control. A path can be set up for payload bytes to travel from the SDP to DMEM, back to the SDP to DMEM and then to SDRAM.

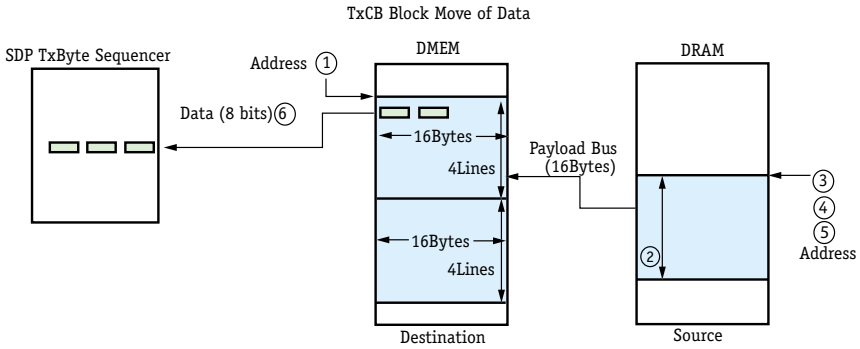
The NP supports two (2) near-end loopbacks for the purposes of recirculation. The first connects the output of the Large Transmit FIFO to the input of the Large Receive FIFO, the second connects the output of the Small Transmit FIFO to the input of the Small Receive FIFO. For more information about recirculation, see "Configuration for Recirculation Operations Using RxSDP and TxSDP" on page 72.

SDP TxByte Processor Transmit Control Block (TxCB0_, TxCB1_)

The CPRC controls transmit operations using the two (2) Transmit Control Blocks (TxCB0_ and TxCB1_). The TxCBs handle the payload read operation much like the (RdCB0 and RdCB1), but add the capability to control TxByte Processor reads from DMEM. Using a TxCB, the CPRC can set up a payload transmit path from SDRAM to DMEM, and from DMEM to the TxByte Processor. Payload data movement happens in hardware with no further CPRC control.

Figure 23 on page 103 shows a Buffer Transfer. In general, you are moving data from SDRAM (the source) to SDP (the destination). Specifically, you are moving data starting at the (*PoolID*, *BTag* and *Offset* (3,4,5)) location inside the SDRAM to DMEM at the *LineAddr* (1) location and moving 8bit units of data from the DMEM into the RxByte Sequencer using *ByteAddr* (6) and counting the bytes moved with a *Length* (2). These individual fields that are used to set up the details of the block move make up parts of these registers: TxCB*n*_Sys_Addr, TxCB*n*_Ctl, TxCB*n*_DMA_Addr and TxCB*n*_SDP_Addr.

Figure 23 DMA Operation (Buffer Transfer) Using TxCB*n*_ Registers



Register	Fields	
TxCBn_Sys_Addr	BTag 4	Offset 5
TxCBn_Ctl		Length 2 Counts the Bytes moved from the sequencer)
TxCBn_DMA_Addr	PoolID 3	LineAddr 1
TxCBn_SDP_Addr		ByteAddr 6

Buffer Transfer Setup Using TxCB*n*_Sys_Addr, TxCB*n*_Ctl TxCB*n*_DMA_Addr and TxCB*n*_SDP_Addr:
To perform Buffer Transfers involves setting the bits for TxCB0_Sys_Addr (0xBC*n*04180), TxCB0_Ctl (0xBC*n*4184), TxCB0_DMA_Addr (0xBC*n*04188), and TxCB0_SDP_Addr (0xBC*n*0418C).

To set up a typical single transmit operation for > 64Bytes of data, the CPRC must do the following:

1 Ensure that the TxCB0 is available by reading *TxCB0_Ctl* and testing that *TxCB0_Ctl* bit [31] *Avail* field=1.

2 Write *TxCB0_Sys* with the system address to be written in the form of:
TxCB0_Sys_Addr bits [31:16] *BTag* field = *BTag*, and
TxCB0_Sys_Addr bits [15:4] *Offset* field = Offset
(Offset is a 16Byte starting buffer offset, typically aligned to a 64Byte boundary).

3 Write the Pool ID portion of the system address to be written into *TxCB0_DMA_Addr* bits [20:16] *PoolID* field. *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field with the location of a 64Byte buffer in DMEM, typically aligned on a 128Byte boundary. This is the DMEM location that the DMA engine uses to begin reading data from SDRAM.

4 Write *TxCB0_SDP_Addr* bits [15:0] *ByteAddr* field with a 16Byte address in DMEM, typically the same value (buffer) as *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field. This is the location that the SDP TxByte Processor uses to begin transferring bytes out of DMEM.

5 Write the *TxCB0_Ctl* register, to initialize the *TxCB0_Ctl* bits [15:0] *TxLength* field with the number of bytes to transfer, clear *TxCB0_Ctl* bit [23] *Own1* field and *TxCB0_Ctl* bit [22] *Own0* to give ownership of the buffer to the SDRAM (rather than the SDP) DMA engine, enabling the prefetch of 128Bytes of payload, clear TxCB0_Ctl bit [31] *Avail* field to start the SDRAM DMA engine.

For complete details about specific registers go to their reference. Refer to: "TxCB0_Sys _Addr Register (CP Tx Control Block0 Function)" on page 396, "TxCB0_Ctl Register (CP Tx Control Block0 Function)" on page 397, "TxCB0_DMA_Addr Register (CP Tx Control Block0 Function)" on page 398 and "TxCB0_SDP_Addr Register (CP Tx Control Block0 Function)" on page 399. *You also have the following registers available for Control Block1: TxCB1_Sys_Addr, TxCB1_Ctl, TxCB1_DMA_Addr and TxCB1_SDP_Addr that function in the same manner.*

Buffer Transfer Operations Using TxCB*n*_Sys_Addr, TxCB*n*_Ctl TxCB*n*_DMA_Addr and TxCB*n*_SDP_Addr:
The CPRC can set up a payload transfer of arbitrary length using double buffering. *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field and *TxCB0_SDP_Addr* bits [15:0] *ByteAddr* field typically point to a 128Byte aligned buffer. Increments of *TxCB_SDP* bits [15:6] *ByteAddr* field and *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field are done to modulo 128 so that writing or reading the last line in the buffer causes the pointers to wrap back to the start of the buffer.

TxCB0_Ctl bit [22] *Own0* field and *TxCB0_Ctl* bit [23] *Own1* field track the ownership of the two 64Byte blocks in the 128Byte double buffer. By clearing the ownership bits initially, the CPRC allows the SDRAM DMA engine to prefetch payload into the DMEM buffers. *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field increments by 16 for each of the four 16Byte-per-cycle transfers to provide an address into DMEM. It also adds 64 to *TxCB0_Sys_Addr* bits [15:4] *Offset* field to update the SDRAM address.

If the payload length is ≤64 bytes, only *TxCB0_Sys_Addr* bit [22] *Own0* field should be cleared and *TxCB0_Sys_Addr* bit [23] *Own1* field set to keep the DMA engine from wasting bandwidth by prefetching an extra block. If the payload length is >64 bytes, *TxCB0_Sys_Addr* bit [22] *Own0* field and *TxCB0_Sys_Addr* bit [23] *Own1* field must be cleared to prefetch the first two blocks of payload. When the blocks of payload arrive from SDRAM, the DMA engine sets the corresponding ownership bit to indicate that the SDP DMA engine now owns the block.

In hardware, setting *Own* causes the SDP TxByte Processor to read and buffer a 16Byte line of DMEM pointed to by *TxCB0_SDP_Addr* bits [15:6] *ByteAddr* field. The SDP TxByte Processor byte reads are serviced from this read buffer. For each byte transferred, the address in *TxCB0_SDP_Addr bits* [15:0] *ByteAddr* field is incremented. Crossing a 16Byte DMEM line triggers another line read from DMEM from the address in *TxCB0_SDP_Addr* bits [15:6] *ByteAddr* field. When the last line of a 64Byte block of payload has been read out of DMEM, the SDP DMA engine clears the corresponding *Own* bit to allow the SDRAM DMA engine to reuse that half of the buffer.

Thus, *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field and *TxCB0_SDP_Addr* bits [15:0] *ByteAddr* field act as a pair, following one another through a payload transfer. *TxCB0_DMA_Addr* bits [13:4] *LineAddr* field leads as the DMA engine fills DMEM with payload from SDRAM. When a 64Byte buffer is full, the SDP TxByte Processor uses the lagging *TxCB0_SDP_Addr* bits [15:0] *ByteAddr* field to read bytes of payload from DMEM. When the TxCB0_Ctl bits [15:0] *TxLength* field equals 0, the hardware clears the *TxCB0_Sys_Addr* bit [22] *Own0* field and the *TxCB0_Sys_Addr* bit [23] *Own1* field and signals the SDP that the last byte was transmitted. The SDP TxByte Processor signals end-of-frame by setting *TxCtl_Status* bit [31] *Avail* field. This clears *TxCB0_SDP_Addr* bits [6:0] *within the ByteAddr* field, clears *TxCB0_DMA_Addr* bits [6:4] with the *LineAddr* field and sets *TxCB0_Ctl* bit [31] *Avail* field to realign to the double buffer and to return control of the TxCB to the CPRC.

There are eight (8) bits in each of the Out-Of-Band field (OOB). The OOB are located in the *TxCB0_SDP_Addr* register. Bits [31:24] are for OOB0 and bits [23:16] are for OOB1. Eight (8) OOB bits are transferred to SDRAM along with every 64Byte payload transfer. The 7th Bit of the OOB*n* indicates that the SDP encountered an error receiving this frame.

The 6th Bit of the OOB*n* indicates that this block of 64Bytes contains the last byte of the payload, and when the 6th Bit of the (OOB*n*) is set, the remaining six bits indicate the position of the last byte. These (OOB*n*) bits get transferred to *TxCB0_SDP_Addr* bits [31:24] *OutOfBand0* and *TxCB0_SDP_Addr* bits [23:16] *OutOfBand1* field for every payload read. Based on the *TxCB0_Ctl* bit [28] *OOB* field, the hardware uses either the *TxCB0_Ctl* bits [15:0] *TxLength* field or the *TxCB0_SDP_Addr OutOfBand* field to determine the last payload byte. Hardware decrements the *TxCB0_Ctl* bits [15:0] *TxLength* field and the appropriate *TxCB0_SDP_Addr OutOfBandn* field for each byte transferred. When *TxCB0_Ctl* bit [28] *OOB* field is clear, the *TxLength* equals 0 indicates the payload transfer is finished. When the *TxCB0_Ctl* bit [28] *OOB* is set, and the appropriate *TxCB0_SDP_Addr OutOfBandn* field indicates last byte and the position equals 0, the payload transfer is finished.

During test, the TxCB can be used to read the SDRAM data and OOB bits. Setting up *TxCB0_Sys_Addr* and *TxCB0_DMA _Addr* bits [13:4] *LineAddr* field determines where payload gets read into DMEM. Software can force the transfer by clearing *TxCB0_Ctl* bit [31] *Avail* field and setting the appropriate *TxCB0_Ctl* bit [23 or 22] *Ownn* bit. When the transfer finishes, the OOB bits can be read from TxCB0_SDP_Addr OutOfBandn field.

The CPRC process can read the state of both DMA engines and the EOP status at any time from *TxCB0_Ctl* bits [19:18] *SDP State* field and *TxCB0_Ctl* bits [17:16] *DMA State* field. The *TxCB0_Ctl* word generally should not be written by the CPRC process when hardware is operating (that is, when *TxCB0_Ctl* bit [31] *Avail=0*). On writes to *TxCB0_Ctl* bits [17:16] *DMA State* field, bits [19:18] *SDP State* field, and bit [29] *EOP* field are only updated if bit [31] *Avail*=1.

When DMA transaction requests receive no-acknowledge (NACK) on the payload bus, the bus controller retries the request up to 16 times before reporting a bus error. The bus error sets status in *TxCB0_Ctl* bits [27:24] *Error* field, generates an event for the *Event1* register, and immediately terminates the SDRAM transfer by setting *TxCB0_Ctl* bit [31] *Avail* field. When the *TxCB_Ctl* bit [30] *NoRetry* field is set, the bus controller does not retry, and reports the bus NACK immediately as a bus error.

Transmit payload can be recycled back through DMEM and retransmitted using a configuration option and explicit CPRC control. A path can be set up for the payload to travel from the SDRAM to DMEM to the SDP to DMEM to the SDP (a process call recirculation). For more information about recirculation, see "Configuration for Recirculation Operations Using RxSDP and TxSDP" on page 72.

Ring Bus Registers

Configuration Space contains registers to control the Ring Bus, including transmitting messages, receiving messages, and receiving responses.

Ring Bus Transmit (Tx) Messages Registers

Configuration Space includes four (4) sets of registers used to transmit messages on the Ring Bus. The four (4) consist of:
TxMsg0_Ctl, *TxMsg0_Data_H*, and *TxMsg0_Data_L;* T*xMsg1_Ctl*, *TxMsg1_Data_H*, and *TxMsg1_Data_L*; *TxMsg2_Ct*l,*TxMsg2_Data_H*, and *TxMsg2_Data_L*; *TxMsg3_Ctl, TxMsg3_Data_H*, and *TxMsg3_Data_L*. The CPRC has access to all four (4) sets. The SDP RxByte and TxByte Processors have access to only sets zero and one, (*TxMsg0_Ctl, TxMsg0_Data_H*, and *TxMsg0_Data_L;* T*xMsg1_Ctl, TxMsg1_Data_H*, and *TxMsg1_Data_L)*.

Refer to the SDP Programming document in the *C-Ware Application Development Guide* for SDP addressing of these registers.

When programming, mutual exclusivity among users of each TxMsgn_Ctl must be maintained.

The 8Byte data portion of a single-slot Ring Bus message is written into two (2) 4Byte *TxMsg0_Data_H* and *TxMsg0_Data_L* registers. The control portion of a Ring Bus message is written into the *TxMsg0_Ctl* register bits [23, 19:0] in the exact format to be sent directly out on the Ring Bus control wires. Clearing the *TxMsg0_Ctl* bit [31] *Avail* transfers ownership of the transmit message registers to the Ring Bus control, effectively giving the send command. The Ring Bus controller then puts the 21bits of control from the *TxMsg0_Ctl* register and the 8Bytes of data from the *TxMsg0_Data_L* registers out on the Ring Bus. The Ring Bus controller sets the *TxMsg0_Ctl* bit [31] *Avail* when the message has gone out, indicating to the CPRC that the transmit message register set is available to send subsequent messages. Four (4) messages of 8Byte data length can be sent independently using the four (4) sets of transmit message registers. Transmit message register sets can also be combined to send messages of 16Bytes and 32Bytes length (two and four Ring Bus slots). Multiple slot messages may begin with any of the transmit message register sets. The additional data is placed in sequential, wrapped *TxMsg0_Data* registers. The beginning *TxMsg0_Ctl* register must contain the appropriate slot length. The sequential *TxMsg0_Ctl* registers that match participating sequential *TxMsg0_Data* registers must have the *Avail* bit [31] set, that is, must not be in use for another transmit, but otherwise have no effect on the transaction.

Ring Bus (Rx) Receive Message Registers

Configuration Space includes a set of registers used to receive unsolicited messages consisting of *RxMsg_Ctl*, and *RxMsg_FIFO*.

Unsolicited messages are of type: indication, confirmation, or request. These incoming messages enter a 4-entry x 8-Byte FIFO in the Ring Bus controller. The CPRC process uses load instructions to read the head of the FIFO from the *RxMsg_Ctl* and *RxMsg_FIFO* registers. When set, *RxMsg_Ctl* bit [31] *State* indicates that a complete, valid message resides in the FIFO. *RxMsg_Ctl* bits [23,14:10,4:0] contain the control portion of the message as received off the Ring Bus. The *Dst* field [9:5] which must be this channel's Ring Bus ID is not reported.

When the FIFO contains a valid message, the CPRC reads *RxMsg_FIFO* to obtain the first 4Bytes of the data portion of the Ring Bus message. The CPRC continues to read from the *RxMsg_FIFO* register to empty the complete message out of the FIFO. The CPRC process must track the message length given by initial *RxMsg_Ctl* bits [17:15] *Len* to know how many times to read the *RxMsg_FIFO* to obtain the complete message. When the *RxMsg_Ctl* indicates a message is valid, the entire data portion of the message is available through *RxMsg_FIFO*; there is no need for the CPRC process to check intermediate data status.

Ring Bus Receive (Rx) Response Registers

Messages initiated by the CPRC as a request type expect to receive a subsequent response type message, for example TLU requests. Configuration space includes eight (8) sets of registers used to receive responses. The eight (8) consist of: *RxResp0_Ctl*, *RxResp0_Data_H*, *RxResp0_Data_L*; *RxResp1_Ctl*, *RxResp1_Data_H*, *RxResp1_Data_L*; *RxResp2_Ctl*, *RxResp2_Data_H*, *RxResp2_Data_L*; *RxResp3_Ctl*, *RxResp3_Data_H*, *RxResp3_Data_L*; *RxResp4_Ctl*, *RxResp4_Data_H*, *RxResp4_Data_L*; *RxResp5_Ctl*, *RxResp5_Data_H*, *RxResp5_Data_L*; *RxResp6_Ctl*, *RxResp6_Data_H*, *RxResp6_Data_L*; *RxResp7_Ctl*, *RxResp7_Data_H*, *RxResp7_Data_L*.

The control field of a Ring Bus response is moved into a *RxRespn_Ctl* register and the data field of a Ring Bus response slot is moved into a *RxRespn_Data_H*/*RxRespn_Data_L* register pair. Responses are directed to the specific one of eight register sets based on the *sequence* bits [12:10] of the incoming Ring Bus control field. The sequence field is merely an echo of the sequence field that was sent in the control field of the request message that triggered this response. *Sequence* field bits [14:13] have no effect on hardware and can be used by software for additional ordering information.

When set, *RxResp0_Ctl* bit [31] *Avail* indicates that a complete, valid response has been received. The *Dst* field [9:5] which must be this channel's Ring Bus ID, the *Type* field bits [19:18] which must be type response, and the *Length* field bits [17:15] which must be known by requesting software, are not reported.

Eight (8) responses of 8Byte data length can be received independently using the eight (8) sets of receive response registers. Receive response register sets can also be combined to receive responses of 16Byte and 32Byte length (two or four Ring Bus slots). Multiple slot responses begin with the receive response register set specified by the *sequence* bits. The additional data is placed in sequential, wrapped *RxResp0_Data* registers. The beginning *RxResp0_Ctl* register contains the Ring Bus control field. The sequential *RxResp0_Ctl* registers that match participating sequential *RxResp0_Data* registers are not updated.

While receiving cells/packets, the SDPRxByte Processor uses its access to the Ring Bus transmit message registers to initiate lookup requests for the TLU based on various fields (such as the destination address) extracted from the incoming header. The TLU responses to the lookup requests are received and interpreted by the CPRC.

For complete details about specific registers go to their reference. Refer to: "TxMsg0_Ctl Register (CP Ring Bus Tx Message Control Function)" on page 401, "TxMsg0_Data_H Register (CP Ring Bus Tx Message Control Function)" on page 403, "TxMsg0_Data_L Register (CP Ring Bus Tx Message Control Function)" on page 403, "RxMsg_Ctl Register (CP Ring Bus Rx Message Control Function)" on page 406, "RxMsg_FIFO Register (CP Ring Bus Rx Message Control Function)" on page 407, "RxResp0_Ctl Register (CP Ring Bus Rx Response Control Function)" on page 404, "RxResp0_Data_H Register (CP Ring Bus Rx Response Control Function)" on page 405, and "RxResp0_Data_L Register (CP Ring Bus Rx Response Control Function)" on page 405. *You also have the other registers available that comprise the sets, and function in the same manner.*

SDP Control and Status Registers

Configuration Space includes a number of general purpose registers for passing control and status information between the CPRC and the SDP Processors.

Five (5) control registers (*RxCtl_ByteSeq0*, *RxCtl_ByteSeq1*, *RxCtl_SyncSeq*, *RxCtl_BitSeq0* and *RxCtl_BitSeq1*) are allocated to communicating with the RxByte, RxBit, and RxSync Processors. The Rx processors perform byte-wide read and write operations from and to these registers under microcode control.

For complete details about specific registers go to their reference. Refer to: "RxCtl_ByteSeq0 Register (CP SDP Rx Control Function)" on page 408, Table 124 on page 408, "RxCtl_SyncSeq Register (CP SDP Rx Control Function)" on page 409, "RxCtl_BitSeq0 Register (CP SDP Rx Control Function)" on page 409, and Table 125 on page 409.

Four (4) control registers (*TxCtl_ByteSeq0*, *TxCtl_ByteSeq1*, *TxCtl_BitSeq0* and *TxCtl_BitSeq1)* are allocated to communicating with the TxByte and TxBit Processors. The Tx processors perform byte-wide read and write operations from and to these registers under microcode control.

For complete details about specific registers go to their reference. Refer to: "TxCtl_ByteSeq0 Register (CP SDP Tx Control Function)" on page 410, Table 126 on page 410, "TxCtl_BitSeq0 Register (CP SDP Tx Control Function)" on page 410, and Table 127 on page 410.

In addition, four (4) status registers (*RxCtl0_Status*, *RxCtl1_Status*,*TxCtl0_Status, and TxCtl1_Status),* two (2) each for the RxSDP and the TxSDP, contain predefined status bits used by the CPRC to track the SDP progress through cell/packet processing and vice-versa.

The CPRC process accesses all of these memory-mapped SDP control registers using load and store instructions. These registers have two (2) read and two (2) write ports, allowing CPRC and SDP access at all times. The CPRC process and SDP firmware must cooperate to ensure data integrity. For both receive and transmit, the SDP microcode sets the *Avail* bit [31] to signal end-of-frame, and thereby switch data scopes. For more information about data scopes. Refer to "Data Scope Detail Operations" on page 86.

For complete details about specific registers go to their reference. Refer to: "RxCtl0_Status Register (XP DMEM#24 Transfer Rx Control Block0 Function)" on page 486, Table 116 on page 400, "TxCtl0_Status Register (CP Tx Control Block0 Function)" on page 400, and Table 116 on page 400.

Miscellaneous Control Registers

Configuration Space includes miscellaneous control registers.

Event Registers

There are a number of events that can occur in a C-5 NP that are asynchronous, and that the CPRC must be able to process. These events must be recognized either by polling for them, or via interrupt notification. Refer to "Interrupt Access" on page 113 for more information. The C-5 NP has the capabilities to reduce the processing time required to respond to an asynchronous event. This event handling mechanism in the CPRC has the following properties:

- Software can identify events and dispatch them to their corresponding processing routines very quickly.
- Software can dynamically prioritize events.
- Software can choose which events will generate interrupts (if any), and which events it will poll.

Each of sixty-four (64) events in the CP is assigned an event number, and a corresponding bit in one of the two (2) 32bit event registers (*Event0* and *Event1*). When an event occurs in the CP (that is, the signal transitions from 0 to 1), it sets the corresponding bit in event registers.

*Most of the bits in the event registers can be interrogated and cleared independently of other state in Configuration Space. However, Event0 register bit [50] and Event1 register bit [21] are exceptions; these bits are not edge sensitive and cannot be cleared directly. Bit [21] represents the logical OR of the current bits in all of the four Queue Status registers (*Queue_Status0 *to* Queue_Status3*). Clearing the Queue Status registers clears Event1 register bit [21]. Similarly, bit [50] SONET event represents the logical OR of the masked bits in the SONET_Event register. Clearing or masking off all SONET events clears Event0 register bit [50].*

The Event Registers comprise two (2) words in the CP (*Event0* and *Event1*), and those words can be read by the CPRC and written with value 1 to clear bits. The normal mechanism for accessing the event status uses the "Event_Access Register (CP Event and Interrupt Function)" on page 439.

For complete details about specific registers go to their reference. Refer to: "Event0 Register (CP Event and Interrupt Function)" on page 435, *and* "Event1 Register (CP Event and Interrupt Function)" on page 437.

Event Access registers are a set of four (4) registers used to provide access to the *Event0* and *Event1* registers. The Event Access registers consist of: *Event_Mask0, Event_Mask1*, *Event_Access*, and *Mask_Access* register.

The *Event_Mask* defines which events the *Event_Access* responds to. It comprises two (2) 32bit registers in the CP. The event number in *Event0* and *Event1* registers is active if its corresponding bit is set in the *Event_Mask0* or *Event_Mask1* registers. This can be done at initialization time or dynamically. Individual bits can be set or cleared in *Event_Mask* by using the *Mask_Access*.

The *Event_Access* returns the logical AND and the logical NOR of the bits from *Event0/Event1* that are active. When the CP reads the value of *Event_Access*, it gets a value of 1 in bit [31] *All* field if all of the bits in the *Event0/Event1* that are set in *Event_Mask* are on. If any of the bits in the *Event0/Event1* that are active in the Event Access registers are reading, *Event_Access* returns 0 in bit [31] *All* field. This allows a program to check whether all interesting events have occurred. If no events are active in the Event Access registers, that is, the *Event_Mask*=0, reading *Event_Access* returns 1 in bit [31] *All* field. When the CP reads the value of *Event_Access*, it gets a value of 0 in bit [15] *None* field if any of the bits in the *Event0/Event1* that are set in *Event_Mask* are on. If all of these bits are 0, reading *Event_Access* returns 1 in bit [15] *None* field. This allows a program to check whether any interesting events have occurred.

The *Event_Access* also provides access to the events in the *Event_Mask0/Event_Mask1* registers, one at a time in a highest-to-lowest event number order. When a program reads *Event_Access*, bits [7:2] *EventNumber* field denotes the "highest numbered active bit", which is set in *Event0/Event1*. The *Event_Access*, bits [7:2] *EventNumber* field is positioned to allow software to use the read value directly as a word index. If *Event_Mask* & *Event0/Event1* = 0, indicating that none of the events active has occurred, reading *Event_Access* returns 0x8000 in bits [15:0] field.

While many of the bits in the *Event0/Event1* correspond to other bits in the CP, they are not directly linked to those bits. When an asynchronous event occurs in the CP, such as the *Avail* bit [31] being set in Receive Control Block 0 (RxCB0_), the corresponding bit gets set in the *Event0/Event1*. Clearing the bit in the *Event0/Event1* does not clear the *Avail* bit [31] in the *RxCB0_Ctl* register.

To clear a particular bit in *Event0/Event1*, a program writes the particular bit number into *Event_Access bits [7:2] ClearBit* field. This lets a program clear an event bit (after processing the event) by writing the same value to *Event_Access*.

To set a particular bit in *Event0/Event1*, a program writes the particular bit number into *Event_Access bits* [23:18] field. This provides a mechanism for setting a software event and having it recognized later in the event polling loop.

Another way to clear one or more bits in *Event0/Event1* is to write a mask value containing the bits to be cleared into the appropriate words of *Event0/Event1* directly. Bits in the *Event0/Event1* are "write 1 to clear".

Single bits in the *Event_Mask* can be set and cleared using the *Mask_Access*, which provides a decode mechanism similar to the one for the *Event0/Event1*. This allows an event dispatcher to dynamically change the events that are interesting to a program as the program modules progress from one stage to the next.

To clear a particular bit in *Event_Mask*, a program writes the particular bit number into *Mask_Access bits [7:2] ClearBit field*. To set a particular bit in *Event_Mask*, a program writes the particular bit number into *Mask_Access bits [23:18] SetBit* field. *Event_Mask0* and *Event_Mask1* registers are also directly writable.

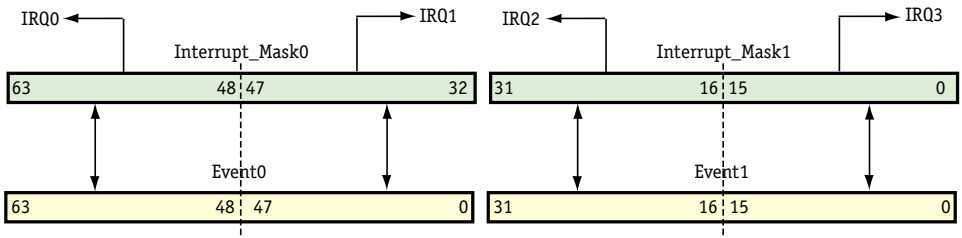
For complete details about specific registers go to their reference. Refer to: "Event_Mask0 Register (CP Event and Interrupt Function)" on page 439, "Event_Mask1 Register (for Mask1)" on page 439, "Event_Access Register (CP Event and Interrupt Function)" on page 439 and "Mask_Access Register (CP Event and Interrupt Function)" on page 441.

Interrupt Access

The CPRC implements four prioritized hardware interrupts, IRQ0-IRQ3. *Interrupt_Mask0* and *Interrupt_Mask1* registers provide a means for software to configure which events in the event register cause interrupts.

An interrupt is requested whenever a bit in the *Event0* or *Event1* register is set and its corresponding bit in the *Interrupt_Maskn* is set. Bits [63:48] in *Event0* register correspond to bits [63:48] in the *Interrupt_Mask0* register that also corresponds to IRQ0. This same type of relationship applies for IRQ1, IRQ2 and IRQ3. Refer to Figure 24 on page 113.

Figure 24 Relationship Between Interruput_Mask0, IRQ0 and Event0 Registers



For complete details about specific registers go to their reference. Refer to: "Interrupt_Mask0 Register (CP Event and Interrupt Function)" on page 441, and Table 134 on page 441.

Queue Status Registers

Queue status from the Queue Management Unit (QMU) is regularly broadcast on a side band of the C-5 NP buses. This status is automatically loaded into the four (4) queue status registers (*Queue_Status0*, *Queue_Status1*, *Queue_Status2 and Queue_Status3)* where it can be read by the CPRC. The CPRC can set bits in the queue status registers by accessing them through the update addresses. The logical OR of the bits in the status registers, provides a level-sensitive event for input to *Event1* register bit [21].

For complete details about specific registers go to their reference. Refer to: "Queue_Status0 Register (CP Queue Status Function)" on page 433 and "Queue_Statusn Registers (for Queue Status 1, 2 and 3)" on page 433.

Cycle Counter

A 64bit Cycle counter is provided in Configuration Space (*Cycle_Count_H* and *Cycle_Count_L*). The cycle counter initializes to 0 during reset and runs freely when reset is released. Thus, the cycle counters in each of the channels are synchronized. The full counter value is readable atomically by the CPRC reading two (2) registers. A copy of the top word is updated whenever the bottom word is read. Only the frozen copy of the top word can be read. For atomic access to the 64bit value, the bottom 32bit word should be read first, then the frozen top 32bit word.

For complete details about specific registers go to their reference. Refer to: "Cycle_Count_H Register (CP Miscellaneous Control Function)" on page 434, and "Cycle_Count_L Register (CP Miscellaneous Control Function)" on page 434.

Event Timer

One event timer register is provided in the Configuration Space (*Event_Timer*). The timer initializes to 0 during reset. After reset, the value in the timer always decrements once per core clock cycle. During the cycle that the timer decrements through 0, a timer event is recorded in the *Event0* register bit [52] *Time-out* field. The timer value can also be read by the CPRC. Applications can write a value into this register that decrements in the same fashion.

For complete details about specific registers go to their reference. Refer to: "Event_Timer Register (CP Miscellaneous Control Function)" on page 434.

Understanding Block Moves of Data

Block moves are used to move data from/to the CPs to/from the BMU, or from/to the CPs to/from the QMU across the Payload Bus.This is done by using Wr, Rx, Rd and Tx features to achieve many different functions. Therefore, to use this feature you should have a basic understanding of the Payload process as described in the following sections.

Payload handling is divided into two (2) types:

- External, a data stream that is received from outside the C-5 NP and transmitted outside the C-5 NP using the Rx and Tx functions, and
- Internal, a movement of data within the C-5 NP using the Wr and Rd functions.

External Handling Overview

This is a general overview of the data movement coming into the C-5 NP. Refer to: "SDP RxByte Processor Receive Control Blocks (RxCB0_, RxCB1_)" on page 98 for more details of the Rx side, and "SDP TxByte Processor Transmit Control Block (TxCB0_, TxCB1_)" on page 102 for details of the Tx side.

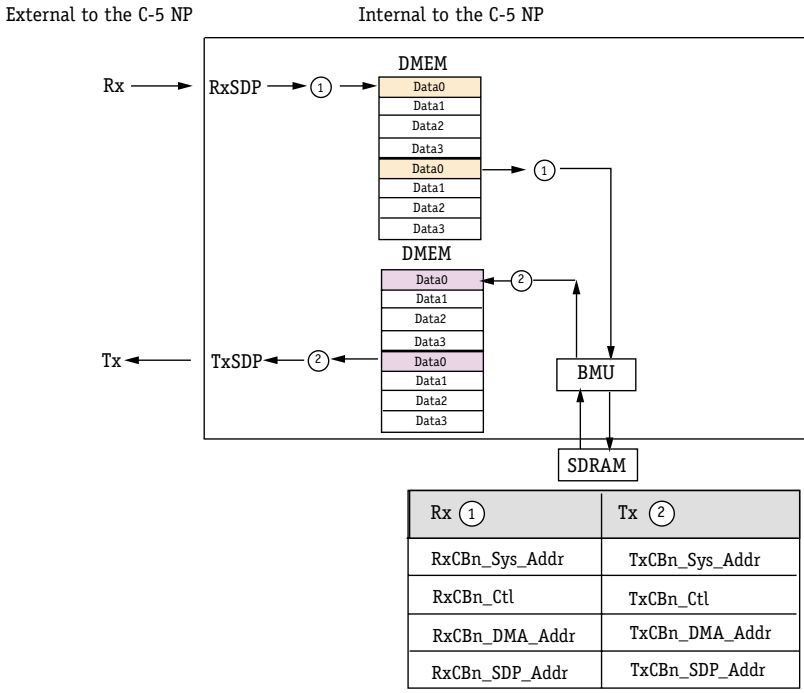
1 Payload handling process, the Rx side:

The flow of the payload handling process starts with a Rx of data from outside the C-5 NP that is processed by the RxSDP, then using the following registers (RxCB*n*_Sys_Addr, RxCB*n*_Ctl, RxCB*n*_DMA_Addr, and RxCB*n*_SDP_Addr) places data (data0) into a location inside the DMEM. The data inside DMEM is then written through the BMU, into the SDRAM for storage. This process explains why an external Rx is associated with an internal Wr. The Rx Control Blocks are used to provide block data moves across the Payload Bus from the CPs to SDRAM. Refer to Figure 25 on page 116.

2 Payload handling process, the Tx side:

The flow of the payload starts with a Tx of data from inside the C-5 NP using certain registers (TxCB*n*_Sys_Addr, TxCB*n*_Ctl, TxCB*n*_DMA_Addr and TxCB*n*_SDP_Addr) that reads the data stored in the SDRAM, through the BMU, then places the data (data0) into the DMEM that is then processed by the TxSDP to outside the C-5 NP. This process explains why an external Tx is associated with an internal Rd. The Tx Control Blocks are used to provide block data moves across the Payload Bus from the SDRAM to the CPs. Refer to Figure 25 on page 116.

Figure 25 Rx and TxCB*n*_ Handling Process Overview (for External Flow)



Internal Handling Overview

This is a general overview of the data movement inside the C-5 NP. Refer to: "Write Control Blocks (WrCB0_ , WrCB1_)" on page 91 for more details of the Wr side, and "Read Control Blocks (RdCB0_ , RdCB1_)" on page 95 for more details of the Rd side.

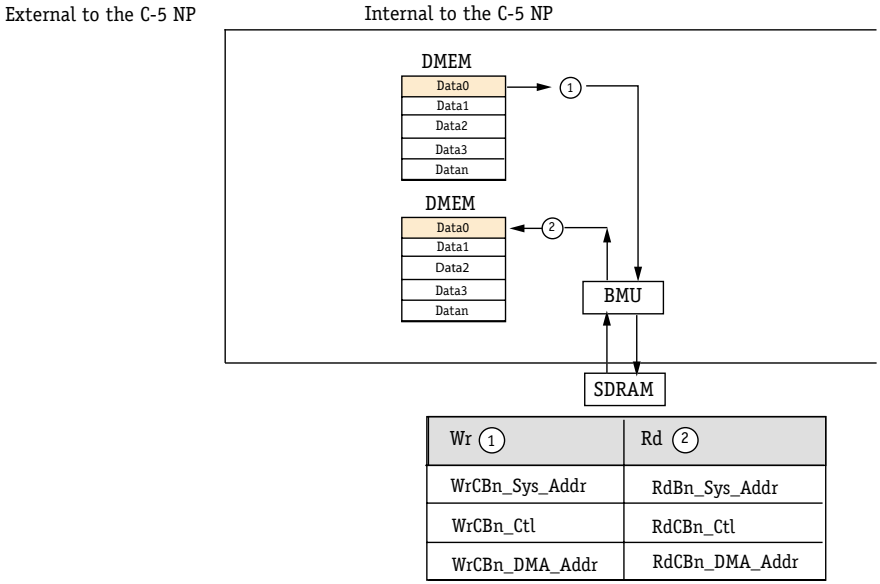
1 Payload handling process, the Wr side:

The flow of the payload handling process starts with a Wr of data from inside the C-5 NP using (WrCB*n*_Sys_Addr, WrCB*n*_Ctl, and WrCB*n*_DMA_Addr) that takes data (data0) from the DMEM and then writes it through the BMU into the SDRAM for storage. The Wr Control Blocks are used to provide block data moves across the Payload Bus from the SDRAM, QMU or BMU to the CPs. Refer to Figure 26 on page 117.

2 Payload handling process, the Rd side:

The flow of the payload handling process starts with a Rd of data from inside the C-5 NP using (RdCB*n*_Sys_Addr, RdCB*n*_Ctl, and RdCB*n*_DMA_Addr) that reads the data stored in the SDRAM, through the BMU then places the data (data0) into the DMEM. The Rd Control Blocks are used to provide block data moves across the Payload Bus from SDRAM to the CPs. Refer to Figure 26 on page 117.

Figure 26 Wr and RdCB*n*_ Handling Process Overview (for Internal Flow)



Wr (1)	Rd (2)
WrCBn_Sys_Addr	RdBn_Sys_Addr
WrCBn_Ctl	RdCBn_Ctl
WrCBn_DMA_Addr	RdCBn_DMA_Addr

Using Multi-Use Control Blocks to Achieve Different Functions

The Multi-Use Control Blocks (WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB0_SDP_Addr; RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr; RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB0_SDP_Addr; and TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr) can be programmed to make data moves to/from SDRAM, the BMU, or the QMU. All of these registers physically reside in the CP memory map at their respective addresses.

The individual fields of these registers are used to perform different functions. Refer to Table 20 on page 118. Detail examples of each, including actual field bit values, are shown in other locations of this manual as noted in this table.

Table 20 Multi-Use Control Blocks (for Wr, Rx, Rd and Tx)

Mode	Category	Function	Fields Used	Details
CP to/from BMU	Memory Transactions	Buffer Memory Transfer Operation	PoolID, BTag, Offset	See "Write Control Blocks (WrCB0_ , WrCB1_)" on page 91 See "SDP RxByte Processor Receive Control Blocks (RxCB0_, RxCB1_)" on page 98 See "Read Control Blocks (RdCB0_ , RdCB1_)" on page 95 See "SDP TxByte Processor Transmit Control Block (TxCB0_, TxCB1_)" on page 102
CP to/from BMU	BTag Management Transactions	Initializing BTags	PoolID, BTag, Command, Pool	See "BTag Initialization Operation" on page 226.
		Allocating BTags		See "BTag Allocation Operation" on page 229.
		Deallocating BTags		See "BTag Deallocation Operation" on page 231.
	Multi-Use Management Transactions	Allocating (Multi-Use Counter)		See "MUC Allocation Operation" on page 234.
		Decrementing (Multi-Use Counter)		See "MUC Decrement Operation" on page 236.
		Reading (Multi-Use Counter)		See "MUC Read Operation" on page 238.
CP to/from QMU	Queue Management Transactions	Configure Queue	Mail Box#, Queue#, Command, PoolID	See "Configure Queue Operation" on page 340.
		Queue Status		See "Queue Status Operation" on page 342.
		Unicast Enqueue		See "Unicast Enqueue Operation" on page 344.
		Multicast Enqueue	Mail Box#, QueueLevel#, Command, PoolID	See "Multicast Enqueue Operation" on page 346.
		Dequeue	Mail Box#, Queue#, Command, PoolID	See "Dequeue Operation" on page 348.

Chapter 3

Executive Processor

Chapter Overview

This chapter covers the following topics:

Executive Processor (XP) Overview

The XP serves as a centralized computing resource for the C-5 NP and manages the system interfaces. One of the system interfaces it manages is the PCI bus, which is generally used for communication to an external *host processor*. If present, a host processor can provide device-wide coordination (for example, between multiple C-5 NPs), network management, signaling, and could possibly build all routing tables for the device of which the C-5 NP is a part. The XP can also perform many of these functions by itself. The XP has access to the internal Global, Ring, and Payload buses.

Typical XP functions include:

- Chip initialization and code download
- Routing/Switching table maintenance (either building tables or importing updates from the host)
- Statistics harvesting from CP DMEM and the TLU
- Fault detection/recovery
- Non-critical-path forwarding functions

XP Major Components

The major components of the XP are listed in Table 21 on page 122. In addition, Figure 27 on page 124 shows the XP Block Diagram.

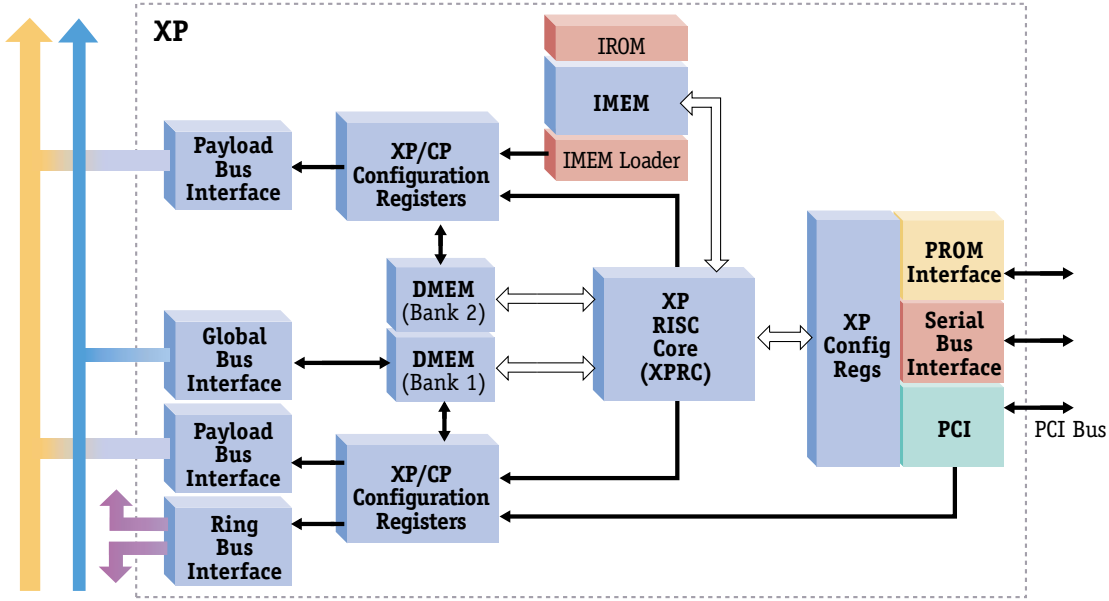
Table 21 Major Components of the XP and Their Function

Item	Function
XP RISC Core (XPRC)	Performs conventional supervisory tasks in the C-5 NP, including: • Reset and initialization of the C-5 NP • Program loading and control of CPs • Centralized exception handling • Management of a host interface through the PCI • Management of system interfaces (PCI, PROM, Serial Bus) This general purpose CPU implements a subset of the MIPS 1 instruction set (multiply, divide, floating point, and CP0 instructions are not supported) with its own dedicated code and data store. The XPRC has Global Bus access to all CP configuration registers and DMEMs. In addition, the XPRC has Ring Bus access for table lookup operations. A 16-word Instruction ROM (IROM) is dedicated to the XPRC. Refer to "XP RISC (XPRC) Overview" on page 125.

Table 21 Major Components of the XP and Their Function (continued)

Item	Function
Memory	Two (2) types of memory are available: IMEM and DMEM. • The XP has 32kBytes of IMEM that contains the RISC Instructions in RAM. It is organized as two (2) 16kBytes banks for sharing within the XP. • The XP has 32kBytes of local non-cached data memory (DMEM) for storage of data. It is organized as two (2) 16kBytes banks. In addition, the DMEM can also be accessed as remote memory by CPs via the Global Bus. Refer to "XP Memory (IMEM and DMEM)" on page 130.
PCI	Provides an industry standard 32bit 33/66MHz PCI channel used for chip-level shared resources. The PCI has both *initiator* and *target* capabilities. A host is optional, but when present, it is capable of: • Requesting the Global Bus (which provides access to all CP configuration registers and DMEMs) • Requesting the Ring Bus (which provides access to table lookup operations) • Requesting XP processing and communicating with the XP for additional services • Supporting C-5 NP initialization Refer to "PCI Bus Interface" on page 132.
PROM Interface	Allows the XP to boot from an external PROM. The PROM interface is a low-speed, serial I/O interface that requires external glue logic to interface to an external PROM up to 4MB in size. Refer to "PROM Interface" on page 134.
Serial Bus Interface	Consists of a general purpose bi-directional, two-wire serial bus and I/O port. It allows the C-5 NP to control external logic with either of two (2) standard protocols. • The high-speed protocol uses a 16bit data format with 10bits of addressing, and supports transfers up to 25MHz. • The low-speed protocol uses an 8bit data format followed by an acknowledge bit and supports transfers at up to 400kbps. The bus supports a single master hierarchy that can operate as either a receiver or a transmitter. The bus also supports an integrated addressing and data-transfer protocol. Refer to "Serial Bus Interface" on page 136.
Configuration Space	This area of the XP contains a number of registers used to communicate with the SDP and the bus controllers (Payload Bus and Global Bus). The XP's registers can also be accessed by other components of the C-5 NP.(CPs via the Global Bus). Refer to "XP Configuration Space" on page 140.

Figure 27 Executive Processor Block Diagram



XP RISC (XPRC) Overview

The XPRC is a general purpose Central Processing Unit (CPU) founded on the same RISC Core used for the CP. Operating at the C-5 NP's core clock rate, the XPRC provides about.85 instructions per cycle (IPC) when executing out of local memory. The IPC and frequency targets offer about 190MIPS per channel on non-blocking code.

The XPRC contains a 32bit data path and accesses memory using a 32bit physical address. It has two (2) banks of local data memory (DMEM); references to memory within Bank 2 (also referred to as DMEM 25) occur with zero wait states; accesses to Bank 1 (also referred to as DMEM 24) incur one core clock cycle latency. Memory addresses outside of local memory range refer to remote memory (that is, the memory contained within the CPs, SDRAM, or I/O devices).

The XP contains memory-mapped control registers (blocks) used for DMA between DMEM and SDRAM, between PCI and SDRAM (via DMEM 24), as well as between SDRAM and IMEM (via DMEM 25). In addition, Configuration Registers enable the XPRC (and PCI interface) access to Payload, Global, and Ring Buses.

XPRC Instruction Set

The XPRC processor uses a MIPS™1 instruction subset (mul, div, floating point, and CPO instructions are not supported). Its 128 registers support fast context switching. See the *C-Ware Application Development Guide* for information on using the compiler that supports the XPRC. Also see the *MIPSpro™ Assembly Language Programmer's Guide* (available over the Internet at http://www.mips.com/publications/index.html) for information about the standard MIPS1 instruction set.

XPRC Registers

The set of internal XPRC registers is defined in Table 22.

Table 22 Internal XPRC Register Definitions

Register Name	Software Name	Use and Linkage
$0	—	Always has the value of 0.
$at or $1	—	Reserved for the assembler.
$2:$3	v0 to v1	Used for expression evaluations and for hold integer function results. Also used to pass the static link when calling nested procedures.
$4:$7	a0 to a3	Used to pass the first four words of integer type actual arguments. Their values are not preserved across procedure calls.
$8:$15	t0 to t7	Temporary registers used for expression evaluations, Their values are not preserved across procedure calls.

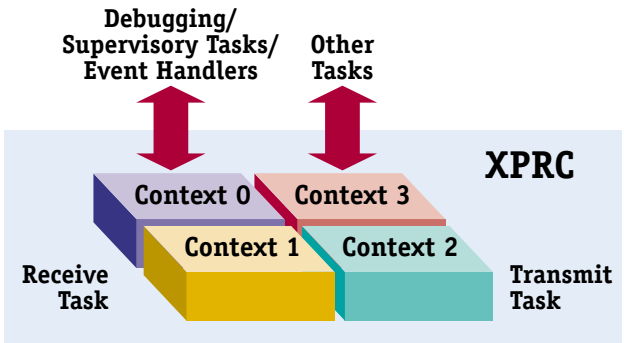
Table 22 Internal XPRC Register Definitions (continued)

Register Name	Software Name	Use and Linkage
$16:$23	s0 to s7	Saved registers. Their values must be preserved across procedure calls.
$24:$25	t8 to t9	Temporary registers used for expression evaluations. Their values are not preserved across procedure calls.
$26:$27 or $kt0:$kt1	k0 to k1	Used internally by the C-5 NP system services.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30	s8	A saved register (like s0 - s7).
$31	ra	Contains the return address used for expression evaluation.

Context Switching

The XPRC incorporates a fast, four-way, context switching facility that replicates the entire XPRC register space four times and can switch from one register set (one context) to another under software control or hardware interrupt. Thus, actual processing (as opposed to manually saving the contents of one set of registers and then loading another) can begin on a different context in only two cycles. Therefore, you can use these four contexts for debugging, supervisory tasks, event handlers, or other tasks.

Figure 28 Executive Processor Context Switching



The XPRC includes four sets of 32 internal registers. Each register set is associated with a processor context. The set of registers are defined in Table 22.

Context switching is accomplished two (2) ways:

- Coprocessor instruction (software)
- Interrupt (hardware)

The software mechanism for executing a context switch is the MIPS MTC0 instruction:

```
MTC0 $1 $3
```

where $1 specifies the destination context. The contexts have no priority; how they are used is entirely designated by software.

The hardware interrupt sequence is:

- All interrupts are disabled until an RFE instruction is executed.
- The address of the next instruction to be executed in the interrupted context is saved in K1 (see "Interrupts").
- Program execution continues with the instruction at the address specified in the interrupt vector.

Interrupts

The XPRC supports four prioritized hardware interrupts, that can be triggered from any bits in the Event Register. There are four MIPS-like register sets corresponding to each hardware context, one register of which (K0) is shared between the other contexts.

K1 contains the program counter value and the context number of the interrupted context. These values are used in the execution of the RFE instruction to return to the previously interrupted context.

All interrupts and exceptions transfer control to a location found in the appropriate interrupt or break table. The base address of the interrupt table is specified by the contents of the interrupt table register ($1) in coprocessor zero. The base address of the exception table is specified by the contents of the break table register ($2) in coprocessor zero.

Interrupts are dispatched by a jump to the address equal to ((interrupt number * 8) + (interrupt table register)). Exceptions are dispatched by a jump to the address equal to ((break number * 8) + (break table register)). In addition to the jump, the register context is set to zero and interrupts are disabled. However, exceptions may still occur. Whether a hardware interrupt or an exception, the interrupted routine's register context and its next program counter are saved in K1 of context zero.

The K1 value points at the next instruction to be executed after the interrupt is serviced. RFE is normally used to: (1) resume the instruction flows at this point, (2) restore the proper register context, and (3) restore the Interrupt Enable Flag to its value at the time of the interrupt or exception.

Note that interrupts are not recognized in a branch delay slot. ALso note that all exceptions fill the delay slot following a change of flow with a NOP instruction.

Interrupts are enable by setting the Interrupt Enable Flag (IEF) which is the LSB of coprocessor zero, Register 8 (see Table 23). The IEF is preserved whenever an exception or an interrupt occurs and is restored by the RFE instruction.

Table 23 Coprocessor Zero Register Definitions

Register	Definition
R0	**Whoami Register** — Contains the DMEM base (hardcoded) for this XPRC.
R1	**Interrupt Table Register** — Contains the vector address for INT 0.
R2	**Break Table Register** — Contains the vector address for break 0.
R3	**Current Context Register** — The two LSBs are the current context register. Set by setting ictxt in decoder.v.
R4	**DMEM Comparison Address** — Contains the address at which debug pulse is generated.
R5	**DMEM Comparison Address Mask** — Contains the mask for the DMEM address.
R6	**DMEM Comparison Data** — Contains the data value for which debug pulse is generated.
R7	**DMEM Comparison Data Mask** — Contains the mask for the DMEM data.
R8	**Interrupt Flag** — The LSB in the Interrupt Flag.
R9	**Read/Write Mask** — The two LSBs are the Read mask and the Write mask for R4 to R7.

Hardware Programming Resources

In addition to fast Context Switching, the XPRC contains resources to aid in efficient program design. These include:

- **Event Registers** — Each bit indicates that the corresponding event has or has not occurred since last set. Centralizing status monitoring into a single register allows for efficient event-driven software design. Many bits are pre-defined, providing high-speed reporting of events between on-chip subsystems (for example, data available on QMU queue). Other bits are software programmable.
- **Cycle Counter** — This 64bit counter is set to 0 when the chip is reset, and increments every core clock cycle thereafter. at overflow, the counter wraps to 0.
- **Countdown Timer** — Applications can set this timer to a value that decrements. When the timer reaches 0, it generates an event for the application.

Event Registers

There are a number of events that can occur in a C-5 NP that are asynchronous, and that the XPRC must be able to recognize and process. These events must be recognized either by polling for them, or via interrupt notification. To reduce the processing time required to respond to an asynchronous event (and hence to improve latency and reduce the chance of losing an event), this event handling mechanism in the XPRC has the following properties:

- Software can identify events and dispatch to their corresponding processing routines very quickly, on the order of 5 to 10 cycles.
- Software can dynamically prioritize events.
- Software can choose which events will generate interrupts (if any), and which it will process via polling.

Each of 64 events in the XP is assigned an event number, and a corresponding bit in one of the two (2) 32bit event registers (*Event0* and *Event1*). When an event occurs in the XP (that is, the signal transitions from 0 to 1), it sets the corresponding bit in event registers. The normal mechanism for accessing the event status uses the Event Access Control Block.

Most of the bits in the event registers can be interrogated and cleared independently of other state in Configuration Space. However, Event0 *register bits [55:52] are an exception; these bits are not edge sensitive and cannot be cleared directly. They represent the logical OR of the current bits in each of the Queue Status registers (*Queue_Status0 to Queue_Status3*). Clearing the Queue Status registers clears these* Event0 *register bits [55:52]. Refer to "*Executive Processor (XP) Configuration Registers*" on page 446.*

XP Memory (IMEM and DMEM)

The XP has both local instruction memory (IMEM) and local data memory (DMEM). These are local memory, not a level 1 cache. In addition, it has the capability to bring in overlays from SDRAM to either IMEM or DMEM, using DMA under program control. The XP also has a local Instruction ROM (IROM).

Instruction Memory

The XP has 32kByte IMEM, configured as two (2) sub-arrays. This memory is shared two (2) ways between the XP and the IMEM loader. The IMEM loader is a logical block that moves code overlays from SDRAM to IMEM. Using the Code Overlay Transfer Control Block, the IMEM loader can DMA code from SDRAM into IMEM via an intermediate buffer in DMEM bank 2 (DMEM #25).

XPRC instruction references outside of the local memory space are not supported. Similarly, the XP IMEM is not visible to any other processors on the chip or to the PCI interface.

Data Memory

The XP has a local 32kByte DMEM. This is organized into two 16kByte banks; bank 2 (DMEM #25) is accessed with zero latency; bank 1 (DMEM #24) is accessed with one additional cycle of latency.

DMEM is organized as 16Byte lines providing 3.2GBps peak bandwidth through a single port. It is accessed via a 4Byte (32bit) access path. The memory resides in the global address space of the C-5 NP; however, only Bank 1 (DMEM24) is accessible by CPs; DMEM Bank 2 (DMEM #25) is not visible to processors outside of the XP. Bank 2 does, however, interface to the Payload bus for data and code transfers, as well as the PCI Bus. Refer to "Executive Processor (XP) Configuration Registers" on page 446.

SDRAM

The XP has DMA access to SDRAM to support data transfers to/from the PCI, IMEM code overlays, and DMEM data overlays. All DMA is controlled using Control Blocks (WrCB0_, RdCB0_, RxCB0_, TxCB0_). SDRAM is not addressable in the global address space. The XP's control blocks provide the following types of SDRAM access:

- DMA to/from DMEM Bank 1 and SDRAM for data overlays.
  [control block RdCB/WrCB #24]
- DMA to/from DMEM Bank 2 and SDRAM for data overlays.
  [control block RdCB/WrCB #25]
- DMA to/from the PCI bus and SDRAM (via buffer in DMEM bank 1).
  [control block TxCB/RxCB #24]

The transfer control block presents transaction requests to the XP Outbound Transaction State Machine, which competes for access to the PCI Master in the XP Outbound Transaction Arbiter. The PCI address and transfer count information for the DMA transfer are provided via additional configuration registers in the XP Configuration Register Block.

- DMA from SDRAM to IMEM (via buffer in DMEM bank 2) for code overlays. [control block TxCB #25]
- DMA from a constant zero data to SDRAM. This is used to initialize SDRAM. [control block RxCB #25]

The transfer control block presents transactions to the IMEM Loader that interfaces directly into the IMEM. IMEM target address information for the DMA transfer is provided via additional configuration registers in the XP Configuration Register Block.

IROM The IROM provides the first instructions when the chip is initialized. It is only accessible by the XPRC. See "C-5 NP Interface Options for Initialization" on page 137 for more information.

XP Supported Interfaces

The XP manages the supervisory controls for the network interfaces as well as the set of pins that provide interfaces to other components in the system that are not memories or network interfaces. The XP supports three (3) system interfaces:

- 32bit PCI Interface (33MHz or 66MHz)
- PROM Interface
- Serial Bus Interface

PCI Bus Interface

Host communication to the C-5 NP is provided through the PCI interface. A host is optional, but when present, it is capable of requesting the Global Bus through the PCI interface. Using the PCI interface, a host can request XP processing through the PCI mailbox registers and communicate with the XP for additional services. A host is capable of supporting C-5 NP initialization without a ROM.

The XP can be configured to support a 32bit PCI interface capable of operating at either 33MHz or 66MHz. The PCI interface on the C-5 NP is fully compliant with the PCI Specification Revision 2.1. The C-5 NP PCI interface includes the following functions:

- Initiation of PCI transactions as a PCI Bus Initiator including:
  - Memory Reads and Writes
  - Internal DMA engines capable of transferring blocks of data between the C-5 NP's SDRAM and the PCI Bus under XP control
- Processing PCI transactions as a PCI Bus Target including:
  - Memory Writes
  - Memory Reads, Memory Read Line, and Memory Read Multiple
  - Configuration Read and Write
  - Single Delayed Transaction
  - Medium DEVSEL timing
  - Configurable via the PCI Interface and/or internal bus accesses from the XP
  - 32bit Addressing
  - 32bit Transfers
  - 33MHz or 66MHz operation

– Support for a single PCI interrupt line (*PCI_INTA*)

The PCI Bus interface does *not* include support for the following functions:

- Exclusive accesses controlled by the *PCI_LOCK_N* signal as either an Initiator or a Target (all requirements for access exclusion to memory space within the C-5 NP are assumed to be handled through software semaphores)
- Special cycles
- PCI cache support (all memory space within the C-5 NP is NOT cacheable to an external processor)
- JTAG (IEEE 1149.1)
- Power management
- Bus arbitration logic (an external PCI Central Resource is required to support this function)

PCI Access to C-5 NP Physical Address Space

An external PCI Initiator can access C-5 NP physical address space through two 1MByte windows in PCI address space. The System Interface Configuration Space contains two standard PCI Base Address Registers (BARs) each defining a 1MB prefetchable memory region.

While the regions are defined as prefetchable, software is responsible for properly handling any read side effects that may occur within the C-5 NP.

For each of these BARs, there is a corresponding address translation register indicating the 1MByte page in C-5 NP physical address space that is to be accessed.

C-5 NP Access to PCI Address Space

The XP can access PCI address space through eight programmable windows in the C-5 NP's physical address space. Each window is controlled by an XP BAR and a PCI address translation register. The BAR controls the location and size of the window in C-5 NP physical address space. The programmable window sizes are: 16kBytes, 32kBytes, 64kBytes, 128kBytes, 256kBytes, 512kBytes, 1MByte, or 2MByte. Each window can be up to 2MByte in size, but the windows can be programmed as any combination of the specified sizes (for example, there could be eight 2MByte windows, or four 128kByte, three 1MByte, and one 256kByte windows). The *PCI Address Translation* register controls the window's view into the PCI address space.

The C-5 NP provides an optional byte swapping mode for moving data between the PCI Bus Little Endian environment and the C-5 NP Big Endian environment. Refer to "PCI Byte Swapping Overview" on page 624.

PCI Registers

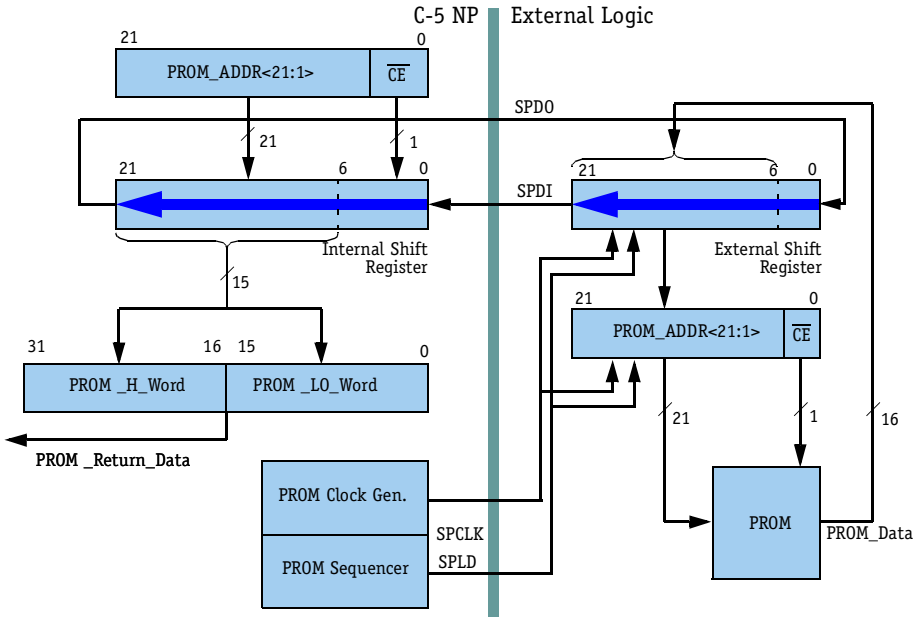
Table 135 on page 447 shows all the PCI Configuration registers. Refer to "Executive Processor (XP) Configuration Registers" on page 446.

PROM Interface

The PROM interface is a low-speed serial I/O interface that allows the C-5 NP to read from an external PROM. The PROM interface clock is created internally in the C-5 NP by dividing the core clock. The clock divider is programmable via the *XP Miscellaneous Control* register and can be set to values ranging from 2 to 16. The maximum PROM size addressable is 4MB and must be provided by the your application in a "by 16" configuration.

The external glue logic (which must be provided by the application) is illustrated in Figure 29 along with the internal mechanisms of the C-5 NP PROM interface. The glue logic consists of an external 22bit shift register with parallel-in and parallel-out capabilities, and a 22bit parallel-in/parallel-out register. Both registers must be positive edge triggered by the PROM interface clock, and perform a synchronous parallel load whenever SPLD is asserted high. For all other cycles, SPLD is asserted low, and the shift register should shift and the parallel register should hold.

Figure 29 PROM Interface



The PROM interface operates in the following manner. Whenever the XPRC, or an inbound transaction being serviced by the PCI target, requests a read from address 0xBFC00000 through 0xBFFFFFFC, the PROM interface initiates the following sequence (which accesses the 16bit wide external PROM to return a 32bit result). Note that two accesses are pipelined together to execute one 32bit fetch:

1. The PROM_ADDR is loaded into the C-5 NP internal shift register.
2. The PROM_ADDR is shifted into the external shift register for 22 SPCLK cycles.
3. SPLD is asserted for one SPCLK cycle, loading the PROM_ADDR into the external presentation register.
4. SPLD is deasserted for 22 SPCLK cycles. The PROM presents the first 16bit PROM_DATA. At the same time, the next PROM_ADDR is shifted into the external shift register.
5. SPLD is asserted for one SPCLK cycle, loading the PROM_ADDR into the external presentation register and the first PROM_DATA into the external shift register.
6. SPLD is deasserted for 22 SPCLK cycles, shifting the first PROM_DATA into the C-5 NP internal shift register.

7 SPLD is asserted for one SPCLK cycle, loading the first PROM_DATA into the C-5 NP PROM_RETURN_DATA register and the second PROM_DATA into the external shift register.

8 SPLD is deasserted for 22 SPCLK cycles, shifting the second PROM_DATA into the C-5 NP internal shift register.

9 SPLD is asserted for one SPCLK cycle, loading the second PROM_DATA into the C-5 NP PROM_RETURN_DATA register.

Serial Bus Interface

The Serial Bus interface is a general purpose bi-directional, two-wire serial bus and I/O port. It allows the C-5 NP to control external logic with either of two standard protocols. The high-speed protocol (MDIO) uses a 16bit data format with 10bits of addressing, and supports transfers up to 25MHz. The low-speed protocol uses an 8bit data format followed by an acknowledge bit and supports transfers at up to 400kbps. Software can select which protocol to use by setting the appropriate bits in the Serial Bus Configuration Register. When a serial bus transfer is active, an external pin is driven by the C-5 NP to indicate which protocol is being used (SPLD=0 indicates high-speed protocol, SPLD=1 indicates low-speed protocol).

The bus only supports a single master hierarchy that can operate as either a receiver or a transmitter. The bus also supports collision detection and arbitration, and an integrated addressing and data-transfer protocol.

Both SIDA and SICL are bi-directional lines that are connected (via a pull-up resistor) to the positive supply voltage. When the bus is free, both lines are HIGH. The output stages of the devices connected to the bus must have either an open-drain or open-collector in order to perform the wired-AND function required for its arbitration mechanism. Refer to "Serial Bus Configuration Register (XP Miscellaneous Control Function)" on page 467, and "Serial Bus Data Register (XP Miscellaneous Control Function)" on page 468.

C-5 NP Interface Options for Initialization

Typically, you use either the PCI or PROM interface to initialize the C-5 NP. Upon initialization, the XP begins executing at the first word of the 16-word IROM. The IROM uses the contents of location BD808300h as a pointer to a formatted boot image, copies the code from that image to the XP IMEM, and begins execution at the code's start address. Unless modified by an external system, the reset value in the boot image pointer is 0xBFC000000, which is the standard address for the boot PROM.

Using the PCI Interface Initialization Option

If you use the PCI to initialize the C-5 NP, you would normally use the C-5 NP as an intelligent peripheral to a host processor. Upon deassertion of the C-5 NP reset, all of the internal CPs and the XP continue to be held in a reset state and the external host processor is responsible for initialization.

The external system contains a C-5 NP boot image that is accessible to the XP via the PCI Bus. This image can be in a boot ROM or in any other memory region accessible via the PCI bus. The external host processor sets up the configuration registers in the System PCI to give the XP access to the boot image, sets the boot pointer to the address of the image in C-5 NP address space, and then releases the XP to begin fetching code over the PCI bus.

Using the PROM Interface Initialization Option

You would use the PROM interface to initialize the C-5 NP if the C-5 NP is used as a stand-alone processor in a single-C-5 NP system. Upon deassertion of reset, the XP immediately begins to fetch code from the PROM.

Other XP Interfaces

In addition, the XP has access to:

- PCI interface with both Initiator and Target capabilities.
- Global Bus access to all CP configuration registers and DMEMs from both the PCI target and the XPRC.
- Ring Bus access from both the PCI target and the XPRC.
- Payload Bus access from both the PCI target and the XPRC via Control Blocks.

All CP configuration registers and DMEMs are accessible via the Global Bus from the XPRC and PCI. However, CPs cannot access XP configuration registers or the PCI bus. Also, CPs can only access one bank of XP DMEM (Bank 1) via the Global Bus; Bank 2 is not visible. In addition, the PCI and XPRC have the same access to all resources with the exception of the IMEM and IROM.

Table 24 lists the accessibility of XP initiated data transactions to various C-5 NP Resources.

Table 24 Accessibility of XP Initiated Data Transactions to C-5 NP Resources

		Transaction Initiator*						
		XPRC	PCI Target	CPs via Global Bus	TxCB/ RxCB #24	TxCB/ RxCB #25	RdCB/ WrCB #24	RdCB/ WrCB #25
Resources	IROM	W	none	none	none	none	none	none
	XP Specific Configuration Registers	W, H, B	W, H, B	none	none	none	none	none
	XP CP-like Configuration Registers†	W, H, B	W, H, B	none	none	none	none	none
	External Serial Bus	H, B	H, B	none	none	none	none	none
	PROM	W	W	none	none	none	none	none
	Ring Bus	Yes	Yes	none	none	none	none	none
	CPs via Global Bus	W	W	none	none	none	none	none
	SDRAM (Payload Only)	none	none	none	16Bytes	16Bytes	16Bytes	16Bytes
	IMEM	W	none	none	none	16Bytes	none	none
	DMEM #24	W, H, B (1 stall)	W, H, B	W	16Bytes	none	16Bytes	none
	DMEM #25	W, H, B (no stall)	W, H, B	none	none	16Bytes	none	16Bytes
	PCI	W, H, B	W, H, B	none	16Bytes	none	none	none

* The table entries indicate accessibility in terms of byte (B), half-word (H), 32bit word (W), and larger transfer operations.

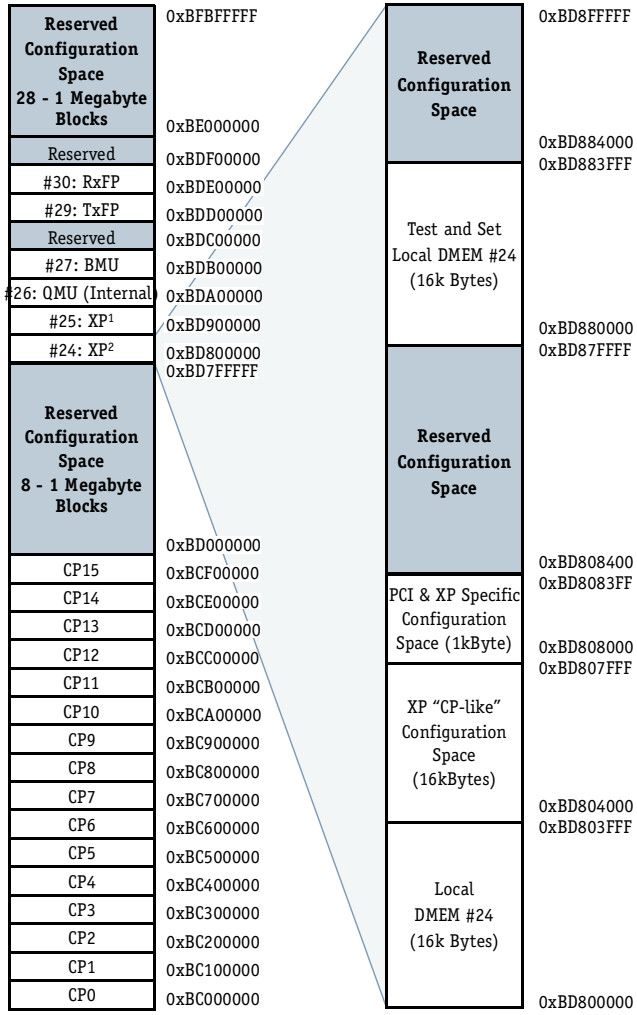
† All control blocks

XP Configuration Space

The Executive Processor (XP) has two areas of memory, referred to as XPSlot 24 and XPSlot 25. Both XPSlot 24 and XPSlot 25 has 1MByte of memory allocated to each for its use. Only the DMEM part of XPSlot 24 can be accessed by all Channel Processor (CPs). In contrast, no CP can access the XPSlot 25 area, however, the XP has full access to the XPSlot 25 area. The memory maps for the XPSlot 24, XP Slot 25, and PCI, XP and other miscellaneous registers are shown in Figure 30 on page 141, Figure 31 on page 142, and Figure 32 on page 143.

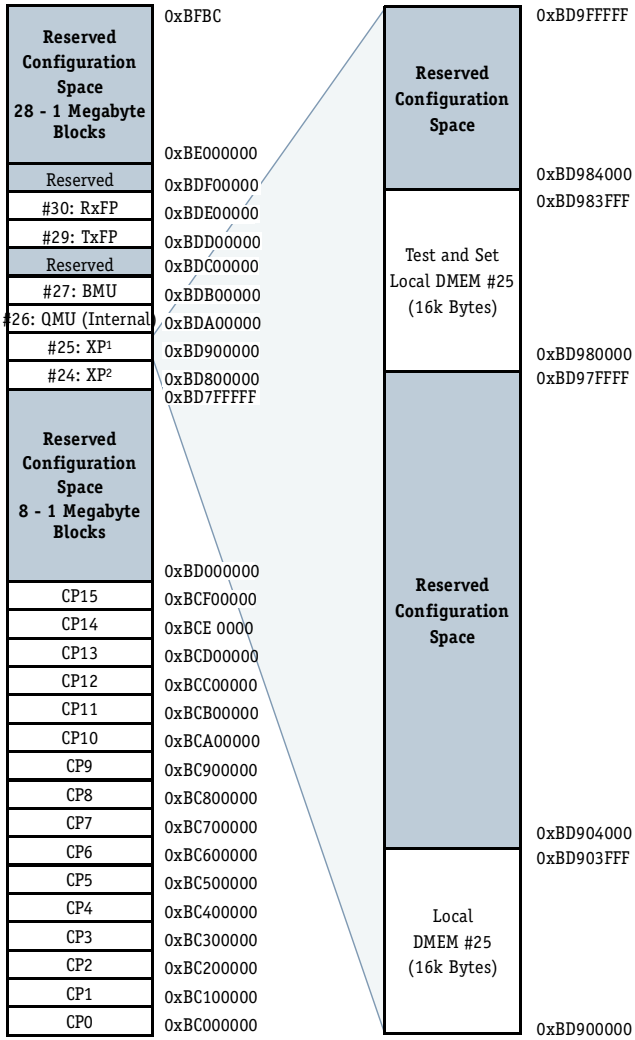
Although specific ranges of memory are allocated to specific functions, the entire area may not be used.

Figure 30 XP Configuration Space (Slot #24)



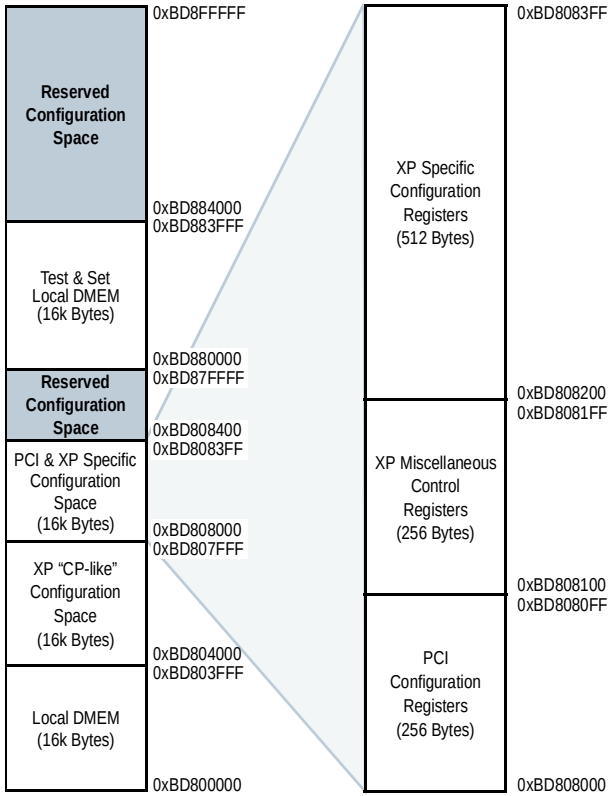
1: XP #25 can only be accessed by the XP, it is not visible to CPs.
2: The CPs can only access DMEM in XP #24.

Figure 31 XP Configuration Space (Slot #25)



1: XP #25 can only be accessed by the XP, it is not visible to CPs.
2: The CPs can only access DMEM in XP #24.

Figure 32 XP Slot #24 Configuration Space for PCI, XP and Miscellaneous Registers



For complete details about specific registers go to their reference. Refer to "Executive Processor (XP) Configuration Registers" on page 446.

Chapter 4

Fabric Processor

Chapter Overview

This chapter covers the following topics:

- Fabric Processor (FP) Overview
- Fabric Processor Transmit (FPTx)
- Fabric Processor Receive (FPRx)
- FP Functionality
- Fabric Interface Configuration and Operation

Fabric Processor (FP) Overview

The FP provides a high-bandwidth port for the segmentation and reassembly of PDUs at up to OC-48 speeds. It behaves like a high-speed network interface port (up to 110MHz for two 32bit data paths) with advanced functionality that allows the C-5 NP to interface to an application-specific switching solution or a switching fabric. The FP can be configured to conform to the UTOPIA-1, -2, and -3, PRIZMA, and Power X interfaces. The programming flexibility of the FP allows it to support standards-based and customer proprietary switch fabric cell formats.

The FP performs flow mapping and management to and from the switching fabric. It can receive up to 159 flows concurrently, and supports transmission of up to 128 prioritized, simultaneous flows arranged as either a 32-port matrix with four priority levels or a 16-port matrix with eight priority levels. These flows support:

- Unicast and multicast topologies
- C-5 NP-to-fabric link-level flow control
- End-to-end congestion management and flow control
- Segmentation and reassembly (SAR) of Protocol Data Units (PDUs) to and from configurable uniform fabric cells for the purposes of higher fabric utilization and Quality of Service (QoS) based arbitration

You can think of the FP as a very high performance CP with limited programmability. It uses the same bus interfaces and data path constructs as a CP. The receive and transmit parts of the FP can operate both autonomously and asynchronously. Because it is microcode programmable using the same instruction architecture as the SDP Byte Processors, the FP can adapt to customer proprietary fabric header formats.

Terminology

For definitions of terminology commonly associated with fabric, refer to the “Glossary” on page 633, at the end of this book.

The word “segment” used in this chapter corresponds to the following:

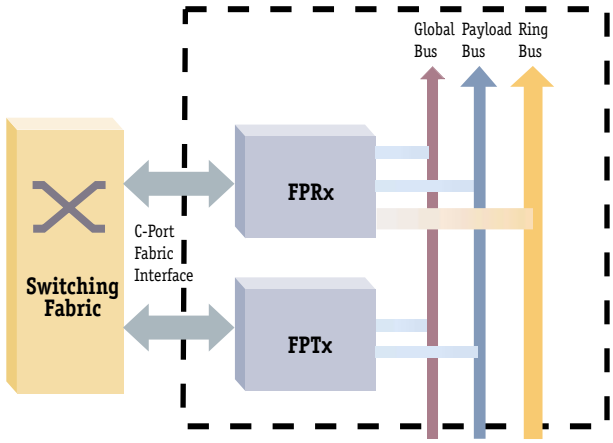
Table 25 Protocol-Specific Nomenclature

Protocol	Nomenclature
Utopia	Cell
PRIZMA	Packet
PowerX	Frame

FP Block Diagram

Figure 33 shows a high-level diagram of the FP.

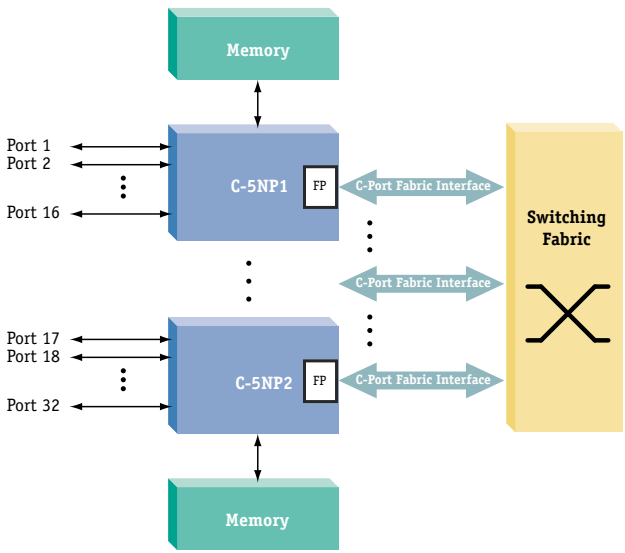
Figure 33 Fabric Processor Block Diagram



Multiple C-5 NP Configurations

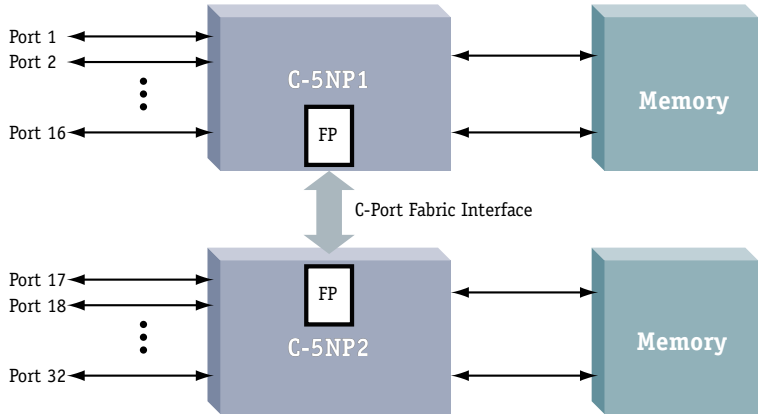
A switching fabric is used when more than two C-5 NPs are required in a system. The switching solution has two or more FP-type ports and provides a mechanism for switching cell or packet-based data from one C-5 NP to another. A homogenous, multi-C-5 NP application is shown in Figure 34.

Figure 34 Multiple C-5 NPs with Switching Port



The FP is designed with symmetric receive (Rx) and transmit (Tx) interfaces. Therefore, the C-5 NP can use the FP to provide a two chip solution, as shown in Figure 35.

Figure 35 Two C-5 NP Application



General FP specifications

- Fabric interface frequency of up to 110MHz
- Separate transmit and receive data buses of 8, 16, or 32bits
- Supported protocols: PowerX, PRIZMA, Utopia 1, 2, 3 ATM and PHY, (except 8bitPHY)
- Segment size: minimum of 40Bytes, maximum of 204Bytes. The size must be a multiple of 4
- PDU size: 5Bytes minimum, 64K-1Bytes maximum

Fabric Processor Transmit (FPTx)

The FP Tx performs essentially the same transmit function as a Channel Processor (CP), but with a high performance and mostly hard-wired implementation. It services a number of QMU queues, using descriptors to identify the PDUs in the BMU buffers. The FP Tx segments the PDUs and places a header at the beginning of each segment before transmitting onto the external fabric interface. The FP Tx can actively transmit segments in a round-robin fashion from up to 8 PDUs. As many as 128 queues can be serviced.

Figure 36 illustrates the main components of the FP Tx. The FP Tx functionality is described below. For a description of functions which span the FP Rx and FP Tx such as flow control, please refer to the FP Functionality section.

The basic flow of a PDU through the FP Tx is as follows.

1. Whenever there is something to transmit (as indicated by the QMU), the FP Tx makes a dequeue request to the QMU via a dedicated FP-to-QMU interface.
2. The descriptor returns from QMU via payload bus.
3. The BTag, pool, PDU length, and multicast flag are extracted from descriptor.
4. The current queue length, which was returned along with the descriptor in step 2, is saved.
5. The payload is read from the BMU buffer pointed to by BTag/pool.
6. The microcode generates headers for segments.
7. Payload and header merge to form segments.
8. The segment CRC is generated.
9. Segment data is adjusted according to endianness.
10. Segments are transmitted via FP Tx interface until entire PDU (as indicated by PDU length) has been transmitted. The segments are interleaved with segments of other PDUs in a round-robin fashion.
11. If a multicast flag was set, the multicast counter associated with pool/BTag is decremented, otherwise the BTag is deallocated back to the pool.

Figure 36 FP Tx Block Diagram

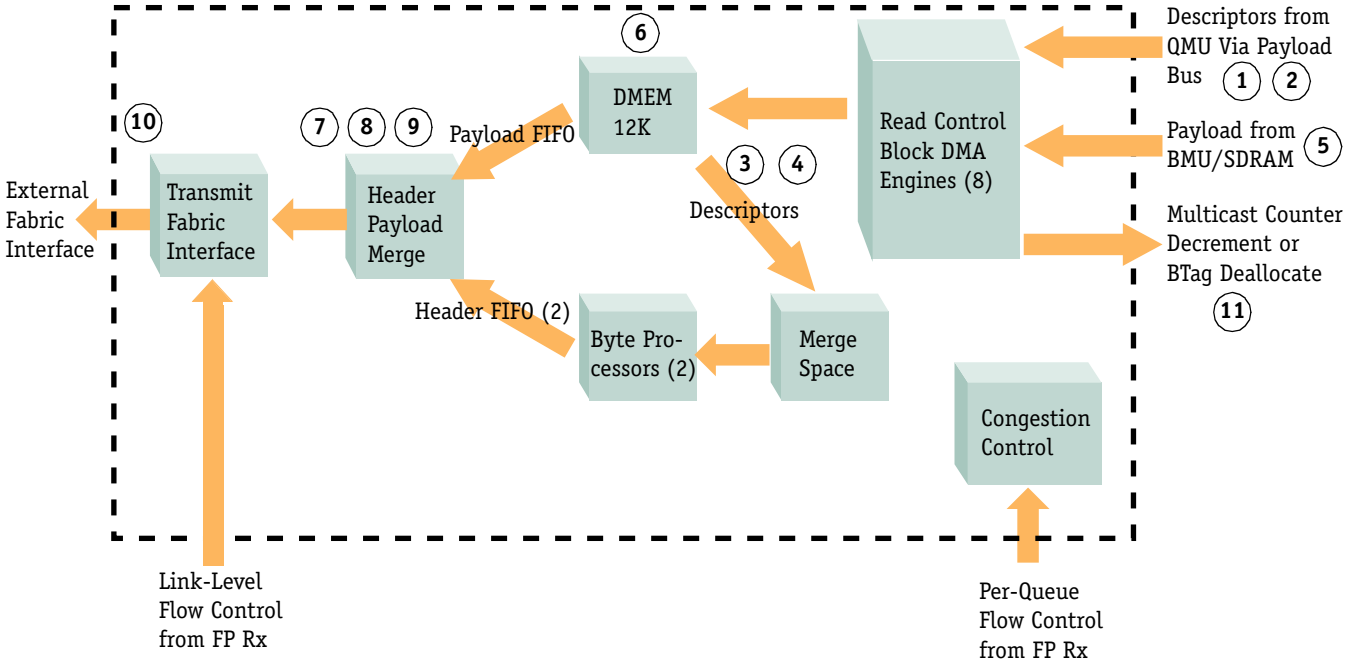
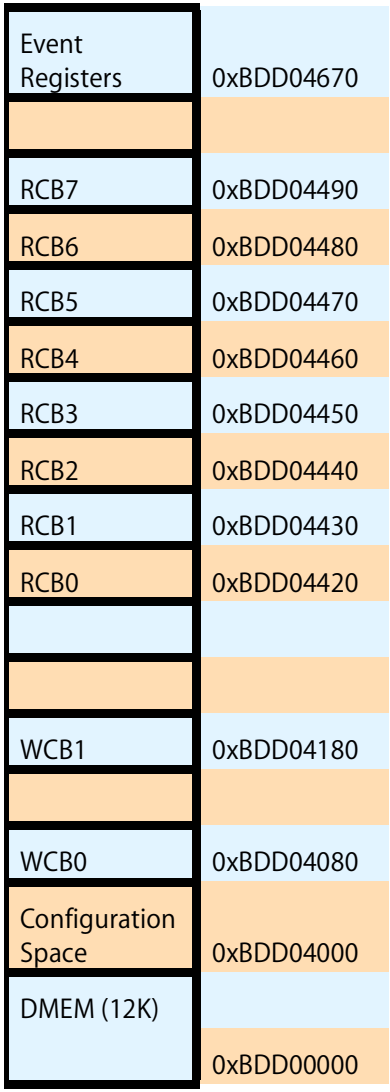


Figure 37 FPTx Memory Map



Transmission Sequencing

The FP Tx only begins dequeuing descriptors and transmitting segments after it has received a queue ready notification from the QMU via the global bus. After that, it continues transmitting PDUs from that queue until the queue length returned with a descriptor is zero; indicating that the queue is empty.

The FP Tx can service up to 128 queues. Its base queue is configured via the Queue Offset field of the *TxSysConfig* register and must be configured to be the same queue as the QMU configuration specifies. The FP Tx has no knowledge of the number of queues assigned to it. It will service any queue for which it receives a queue ready notification from the QMU. For details about the specific register, see "TxSysConfig Register (FP Tx Configuration Function)" on page 534 in Appendix A.

If more than one queue is non-empty, the FP Tx transmits segments in an interleaved fashion, with up to eight queues being transmitted simultaneously.

The FP Tx will always complete transmission of one PDU from a queue before beginning another PDU from that queue, thus ensuring in-order transmission from queues.

The FP Tx can be configured to cause its 128 queues to be organized as 32 ports with 4 priorities per port or 16 ports with 8 priorities per port. The organization is selected via the Queue Depth field of the *TxFCE Configuration* register. Even if fewer than 128 FP Tx queues are used, the port organization is the same. The FP Tx does not know how many queues are used.

Ports are contiguous sets of queues with the highest priority queue being the lowest numbered queue. For example, with a 32x4 organization, queues 0 to 3 are part of port 0, with queue 0 being the highest priority queue within that port and queue 3 being the lowest priority queue within that port. When multiple queues are non-empty, the FP Tx uses the algorithm shown in "Weighting Algorithm" on page 159 to select the next queue from which to transmit.

Descriptor Format

Every descriptor that the FP Tx dequeues must contain:

- A 5bit pool and 16bit BTag that point to the buffer to be transmitted
- A 16bit PDU length indicating the amount of data in that buffer which should be transmitted
- A 1bit multicast flag

The FP Tx can be configured to extract these parameters from different locations within the descriptor with a few constraints. This configuration is done using the *TxDescInfo* register.

For multibit fields (length, pool, and BTag), the "position" indicates the position within the descriptor of the least significant bit of the field.

Bit positions within the descriptor are numbered so that bit position 0 is the least significant bit of the first 32bit word of the descriptor, bit position 32 is the least significant bit of the second 32bit word of the descriptor, etc.

Multibit fields must be positioned so that they do not cross 32bit boundaries. For example, the 5bit pool field cannot be positioned at bit 30 of the descriptor because it would require some bits in the first 32bit word of the descriptor and some bits in the second 32bit word. For details about the specific register, see "TxDescInfo Register (FP Tx Configuration Function)" on page 532 in Appendix A.

Reading the Payload

When the FP Tx begins transmitting a PDU from a buffer, it always transmits from the beginning (offset 0) of that buffer. It is not possible for the FP Tx to begin transmission from any offset inside the buffer.

The FP Tx ignores the out-of-band bits returned from the BMU and relies on the length passed to it in the descriptor to determine what portion of the buffer to transmit.

Microcode Generation of Headers

Because header formats vary from fabric protocol to fabric protocol and application to application, the FP Tx includes a microcoded Byte Processor, similar to those in the Channel Processors, to generate the header for each segment. In fact, there are two Byte Processors in the FP Tx. Headers for consecutive segments are generated alternately by one of the two.

Typically, more information needs to be included in the header of the first segment of a PDU than is required in subsequent segments. For example, the header of the first segment typically must include information about the length of the PDU and the destination queue on the receiving network Processor. This information needn't be conveyed in subsequent segments for that PDU.

For this reason, the FP Tx supports two different header sizes; one which is used for first or only segments and another which is used for middle or last segments. Because the FP Tx will always append as much payload as possible after the header, this allows more payload to be transmitted with each middle and last segment, thus increasing the efficiency of the transmission. The header sizes are configured via the Header Length fields of the *TxDM Header/Payload Delimiter* register. Headers for all cells can be configured to be the same size. FP Tx microcode must generate headers of exactly the sizes configured. Header sizes must be non-zero and multiples of 4. For details about the specific register, see "TxDM_Header/Payload Delimiter Register (FP Tx Configuration Function)" on page 532 in Appendix A.

FP Tx Microcoding

In order for the the FP Tx Microcode to operate properly, the minimum requirements for FP Tx microcode are that it:

- Wait for datascope ownership
- Wait for the header FIFO to be empty using the header FIFO empty test condition
- Build the header by writing out the header Bytes in sequence
- Flip ownership for that datascope

Generally, FP Tx microcoding is done much like microcoding for an SDP Byte Processor. The FP Tx Byte Processor is capable of the same sequencing and ALU operations as the SDP Byte Processor, with 64 control store entries and 24 CAM entries. The unique aspects of the FP Tx Byte Processor, compared to an SDP are:

- The external test conditions
- Inputs
- No input FIFO

External test conditions

The external test conditions available to the FP Tx Byte Processor are:

0 Not used

1 Header FIFO empty

2 Not used

3 Not used

4 Not used

5 Not used

6 Not used

7 Not used

The header FIFO empty test condition is true when the header FIFO for the Byte Processor is empty.

datascope

Because the FP Tx can transmit up to eight queues at a time, it provides context or "datascope" for one of eight PDUs at a time to the Byte Processor. There are times during

which the FP Tx hardware is updating this context and so the datascope is not ready for processing. Because of this, microcode must wait until it is granted ownership of a datascope before it begins constructing a header. Microcode tests for datascope availability using the Ownership bit in the *TxStatus* register.

When FP Tx microcode has finished generating a header, it passes ownership for the datascope to hardware by setting the Ownership bit. This indicates to the FP Tx hardware that the full header is constructed and ready to be merged with payload to form a segment.

Performance Requirement

A segment will not begin transmission until both its payload and header are ready. For optimal performance, FP Tx microcode should be constructed to complete header building at a rate faster than segments can be transmitted on the interface, otherwise, bandwidth will be wasted.

Header Inputs

The Byte Processor can read a number of different things to construct a header. These are:

- Current payload length

  This value represents the number of Bytes of payload appended to the header to form the segment. Typically this is only useful for applications such as PowerX where the segment size is included in the fabric header. See "Pay_Len [7:0]" on page 158.

If the CRC is enabled, the payload length includes an additional 4Bytes of CRC.

- Current segment type (bits 1:0 in the TxStatus register). Encoding:

Table 26 Segment Types

Encoded Value	Segment Type
00	Middle
01	Last
10	First
11	Only

For details about the specific registers see "TxByte Processor Registers" on page 156.

- FP queue

  This value represents the FP queue number (offset from the FP base queue) from which the current segment's PDU came. Typically, this would be used to form the PDU ID or fabric address. See "Src_Queue [6:0]" on page 158.

- General Purpose Configuration Registers

  There are 8Bytes of general purpose registers that can be initialized with global writes and read by either Byte Processor. For example, one of these Bytes might be initialized to contain a unique Network Processor ID (for a multiple Network Processor system) that could then be incorporated into a PDU ID in the header used by the FP Rx for reassembly. Both Byte Processors read the same value from these registers. There are no restrictions for when the two Byte Processors can read these registers; that is, they can both read any Byte any time, including different Bytes at the same time. The Byte Processors cannot write to these registers. See "General Purpose Registers" on page 176.

- Descriptor contents

  Typically, headers contain at least some portion of the descriptor that the FP Tx dequeued. All Bytes of the current descriptor are made available to the Byte Processor through an internal memory known as *Merge Spac*e.

- Information from FP Rx Byte Processors

There are 17 bits of information that the FP Rx sends to the FP Tx hardware for per-queue flow control, as will be described later (see "Fabric to C-5 NP Per-Queue Flow Control" on page 193). These bits can also be read by the FP Tx Byte Processors as a general purpose communication mechanism from the FP Rx Byte Processors.

If used for this purpose, disable flow control. For details about the specific register, see "TxFI_Configuration Register (FP Tx Configuration Function)" on page 530 in Appendix A. Refer also to the registers beginning "Pool0_CFG0" on page 69.

- Literals

TxByte Processor Registers

The TxByte Processor Register Block is composed of three sets of registers:

- **Merge Space** — 64, 32bit Merge registers (256Bytes) organized as eight datascopes of 32Bytes.

For details about accessing merge space, see "TxMergeAddr (FPTx Debug Function)" on page 542, "TxMergeData (FPTx Debug Function)" on page 543 and "TxFDP_Mrg0 - TxFDP_Mrg63" on page 543 in Appendix A.

- **Control Space** — control information unique to each Byte Processor.
- **TxByte General Purpose Registers** — two 4Byte registers shared by both Processors

While all but the TxByte General Purpose (GP) registers are used to pass information between the TxByte Processor and associated FP hardware, all of these registers are mapped to Global Address Space for debug purposes. Figure 38 on page 158 shows a TxByte Processor memory map and Table 27 on page 158 provides a summary of the Extract registers.

Merge Space

The Merge space is globally accessed via the TxMergeAddr and TxMergeData registers shown below. The *TxMergeAddr* register is used to index into the Merge Block and read/write data via the *TxMergeData* Register.

Merge space contains 64, four Byte Merge registers used for passing fields to be inserted as part of the Segment Header by the FDP TxByte Processor. During normal operation the FDP TxByte Byte Processor performs Byte-width reads and the FPTx hardware writes the Merge registers with the entire Internal Descriptor. Byte 0 of the Descriptor is written to Merge[0], Byte *n* of the descriptor is written to Merge[n], and so on. The FDP TxByte Byte Processor cannot write these registers and is restricted to one descriptor (up to 32Bytes at a time dependent upon datascope).

The Merged information is prepended as part of the Segment Header and formatted to the fabric destination descriptor format. There are eight descriptors of either 32 or 16Bytes in length as configured by the "TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535 in Appendix A.

Figure 38 TxByte Processor Memory Map

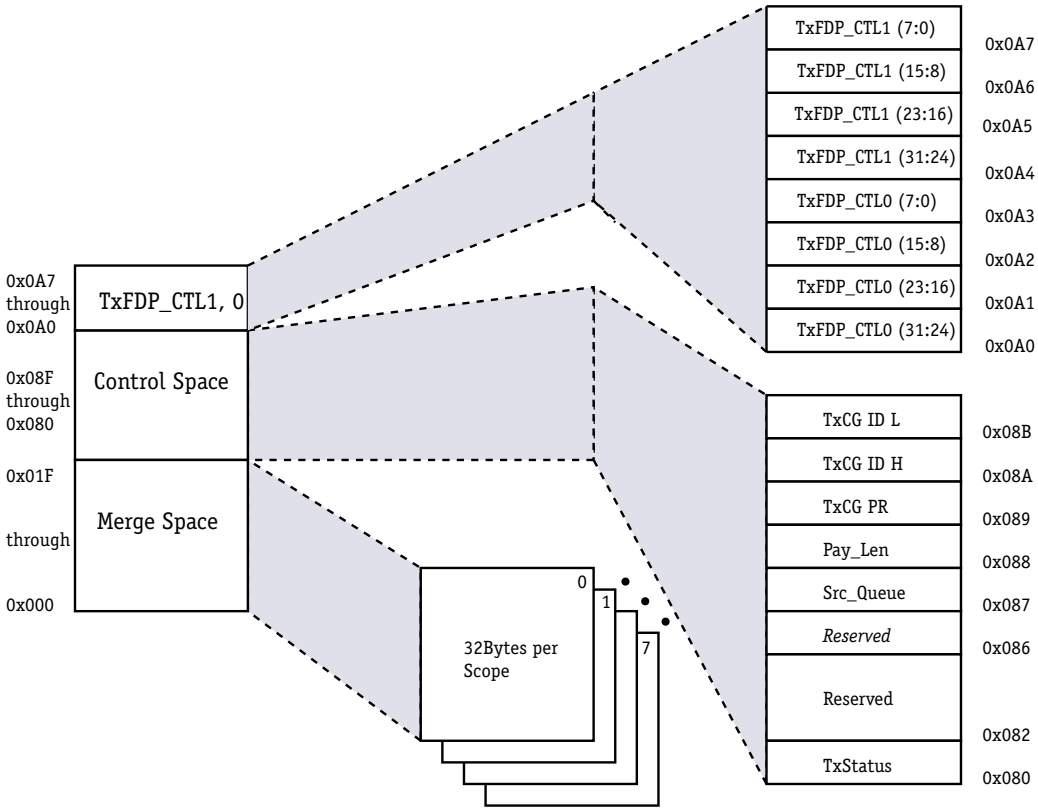


Table 27 TxByte Processor Registers Summary

Fabric Data Processor Offset	Register Name	Register Description	Access	
			Global Bus	Fabric Data Processor
0x000 - 0x01F	Merge0 - Merge63	64 32bit merge registers used for descriptor data, organized as eight groups of 32Bytes. The datascope number selects the group and the FDP offset selects the Byte.	R/W[1]	R
0x0A0 - 0x0A7	TxByte0 Ctl0 - Ctl7	Shared general purpose registers	R/W	R
0x080	TxStatus [7:0]	Status register	—	R/W
0x087	Src_Queue [6:0]	Cell Source Queue # (0 - 127)	—	R
0x088	Pay_Len [7:0]	Payload Length within cell	—	R

Table 27 TxByte Processor Registers Summary (continued)

Fabric Data Processor Offset	Register Name	Register Description	Access	
			Global Bus	Fabric Data Processor
0x089	TxCG PR [0]	Congestion register Pause/Resume bit (bit 0)	—	R
0x08A	TXCG ID H	Congestion Flow ID High [15:8]	—	R
0x08B	TXCG ID L	Congestion Flow ID Low [7:0]	—	R

1 Accessed via Merge Address/Data registers.

The FP TxByte Processor is analogous to the RxSync Processor in the SDP.

Weighting Algorithm

A weighting algorithm can be used to balance the number of PDUs which get transmitted from each of the 128 queues. For applications which use only one queue, the algorithm has no effect. It is most useful for applications with fixed-size PDUs; in such a case the algorithm effectively acts as a bandwidth allocator.

Using the *TxQueueWeight* register, each queue can be configured with a static 4bit weight value. If not explicitly configured, all queues will default to a weight of 1. To configure a weight value to something other than the default, write the queue number and desired weight to the *TxQueueWeight* register, while also setting the Write bit in that register. When all of the weights have been set, deassert the Write bit.

Each queue also has a dynamic 4bit field which specifies PDUs remaining in the current round for this port. This count initially equals the weight value. As a PDU is completely transmitted for a queue, the count is decremented.

When deciding which queue to transmit the next PDU from, the FP Tx advances to the next port, and then choose the highest priority queue (within that port) which satisfies 2 conditions: (1) it is non-empty, and (2) the PDUs remaining in this rounds' count is nonzero. The same algorithm is applied to each port in a round-robin fashion. When the counts reach zero for all nonempty queues within a port, the counts for all queues in that port are reset to the initial weight value.

Example 1:

Queue Organization: 32 ports with 4 priorities per port (that is, q0 is the most significant queue for port0, and q3 is the least significant queue for port0)

All 128 queue weights equal 1

Presume all queues have something to transmit (non-empty)

PDU transmission order: q0, q4, q8….q124, q1, q5, q9….125, q2, q6, q10….q126, q3, q7, q11….q127 (repeat)

Example 2:

Queue organization: 16 ports with 8 priorities per port (port0 is represented by q0-q7)

The application uses 16 queues; 8 on port0, 8 on port1, with the rest unused (always empty)

The 8 weights for each port are configured to be: 3, 2, 1, 1, 1, 1, 1, 1.

Presume all 16 queues remain non-empty

PDU transmission order: q0, q8, q0, q8, q0, q8, q1, q9, q1, q9, q2, q10, q3, q11, q4, q12, q5, q13, q6, q14, q7, q15 (repeat)

The weight value of 3 on queue 0 causes 3 PDUs to be transmitted from that queue for every port0 round. The FP Tx alternates ports in a round-robin fashion. In this example ports 2 to 15 have nothing to transmit, so the queues are chosen alternately from ports 0 & 1.

Error Reporting and Interrupts

The following four errors are detected by the FP Tx and logged in bits [31:28] of the TxFCE Configuration register. In addition to being logged, these errors will cause an interrupt to be sent to the XP (if the Interrupt Enable bit in the TxFCE Configuration register is asserted). For details about the specific register, see "TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535 in Appendix A.

The register bits for each error remains asserted until the Interrupt Acknowledge bit [26] (again, in the "TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535) is written to a 1. When writing the register, be careful not to change the other configuration fields, such as Descriptor Size. The Interrupt Acknowledge bit must then be set to a 0 again. When writing the register, be careful not to change the other configuration fields, such as Descriptor Size.

These topics are covered in the following sections.

- Descriptor (QMU) Parity Error
- Buffer (BMU) Read Error
- Write (BMU) Error

- Dequeue (QMU) Failure

Descriptor (QMU) Parity Error

Indicates that a parity error occurred when the QMU sent a descriptor. This error will only be logged if the QMU Parity Error Enable bit ("TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535, bit [27]) is set. In this case, the error has a separate enable in addition to the interrupt enable.

Descriptor parity errors are not detected and reported by the QMU if 12Byte descriptors are used and are only detected and reported for some words of the descriptor for 24Byte descriptors.

If the error occurs (and is enabled), no additional PDUs will be transmitted for the queue in question. Further, no PDUs will be transmitted for the lower priority queues in the same port.

Buffer (BMU) Read Error

A buffer read error may occur for a number of different reasons.

- **ECC error** - BMU detects an ECC error when it reads the buffer from SDRAM
- **Retry timeout** - BMU is unable to satisfy the buffer read request because it is too busy.
- **Non-existent memory error** - this would only occur if the BMU were misconfigured to be storing buffers in a non-existent memory location.

Whatever data is transferred from the BMU will be transmitted.

Write (BMU) Error

- **Retry Timeout** - A BTag deallocate operation or a multi-use counter (MUC) operation could fail because the BMU is too busy to service the request or because the pool ID used for the operation is invalid. The invalid pool ID errors only occurs if the pool value in a descriptor passed to the FP Tx is invalid or the FP Rx was illegally configured to use an invalid pool. The BMU should never be too busy to service these operations.
- **Multi-use Counter Decrement Error** - Multiuse counter decrement operations can also fail if the multi-use counter does not exist in the BMU. This could only occur in the event of a CP/XP software error; for example, if it failed to set up the counter, initialized it to a value which was too small, or decremented a counter more than once after transmitting.

If one of these write errors occurs, the FP Tx will not retry the operation and the buffer will effectively be leaked because its BTag will never be freed.

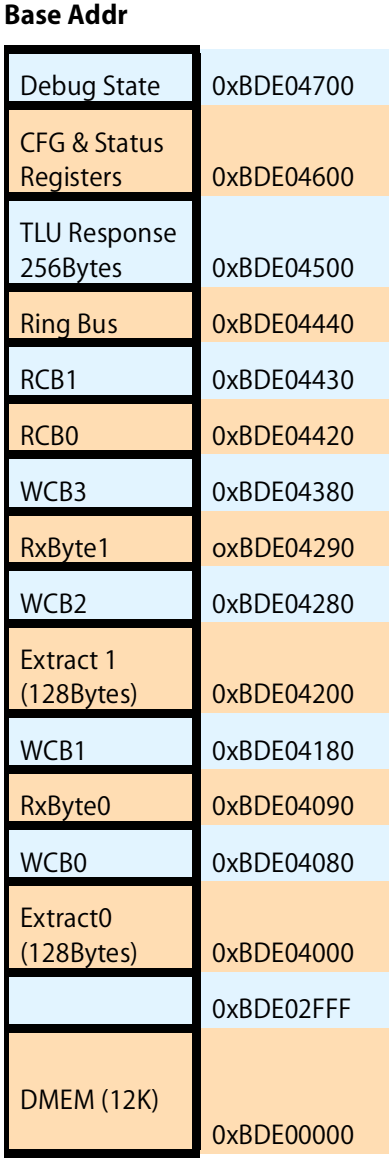
Dequeue (QMU) Failure

A dequeue operation will fail if the queue is empty when the request is made. A dequeue from an empty queue should only occur in the event of a CP/XP programming error where a CP or XP removes something from a queue belonging to the FP Tx.

Fabric Processor Receive (FPRx)

The FP Rx performs essentially the same receive function that a CP does, but with a high-performance and mostly hard-wired implementation. The FP Rx receives segments and writes them to BMU buffers, reassembling them into PDUs, while building and enqueuing the associated descriptors.

Figure 39 FPRx Memory Map



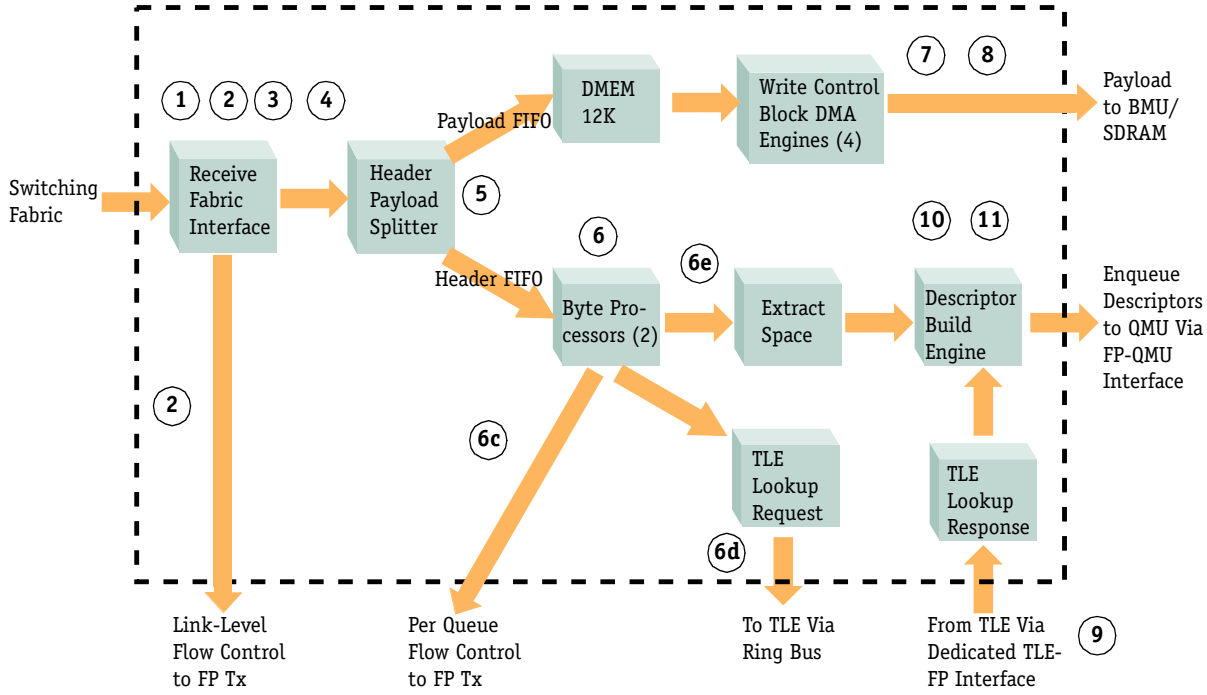
The FP Rx can reassemble segments from up to 159 interleaved PDUs.

Figure 40 illustrates the main components of the FP Rx. The FP Rx functionality is described here. For a description of functions which span the FP Rx and FP Tx such as flow control, please refer to "FP Functionality" on page 187.

The basic flow of a segment through the FP Rx is as follows.

1. A segment arrives at fabric interface.
2. The in-band link-level flow control information extracted from the header (PRIZMA mode).
3. The segment data is adjusted according to endianness. See "Endianness (Byte and Bit Ordering)" on page 195.
4. The segment CRC is checked.
5. The segment header is sent to header FIFO of one Byte Processor, the payload is sent to payload FIFO.
6. The Byte Processor microcode processes the header.
   a. It extracts PDU ID and segment type.
   b. It extracts PDU length (optional).
   c. It processes in-band per-queue flow control (optional).
   d. It launches TLU lookup (optional).
   e. It saves header content for descriptor building (optional).
7. If this is the first segment of a PDU, then a buffer is selected for payload based on PDU length.
8. The payload is written to the BMU buffer.
9. The TLU response is returned (optional).
10. If this is the first segment of a PDU, the descriptor is built.
11. If this is the last segment of a PDU, the descriptor is enqueued.

Figure 40 FP Rx Block Diagram



Header and Payload Splitting

Because different segment types can have different sized header and payload regions, the FP Rx supports splitting header and payload for up to three different types of segment. The segment type must be identifiable by checking the masked value of some single Byte of the segment. Which Byte to check, the mask, and the comparison value are all configurable via the change registers. For details about the specific registers, see "RxDS_Header_Change1 Register (FP Rx Configuration Function)" on page 553 and "RxDS_Header_Change2 Register (FP Rx Configuration Function)" on page 554 in Appendix A. Segment type checking is done in a prioritized order with the first match specifying the type. If neither of the change registers matches, a configurable default splitting is done.

Delimiter0 register must always be used and properly configured. Delimiters 1 and 2 are optional. C-Port recommends that if you use more than 1 delimiter, use the #1 delimiter before the #2 delimiter.

```
If (change reg 2 enabled AND (segment[change 2 reg index] &
   change reg 2 mask) ==
   change reg 2 value))
   Use delimiter reg 2
Else If (change reg 1 enabled AND (segment[change 1 reg
   index] & change reg 1 mask) == change reg 1 value))
   Use delimiter reg 1
Else
   Use delimiter reg 0
```

In the equation above, "&" means "bitwise".

For each segment type, the number of segment Bytes to be directed to the header FIFO of a Byte Processor for header processing is configurable. Headers are always presumed to begin with the first Byte of the segment (Byte offset is zero). The portion of the segment which will be directed to the payload FIFO, ultimately to be stored in a BMU buffer, is also configurable. Header/payload regions are configured via the delimiter registers. The delimiter registers allow you to specify a header and payload which overlap, allowing as much or all of the header to go to the payload or leaving a gap between header and payload. For details about the specific registers, see the pool config registers starting with "RxDS_Header/Payload_Delimiter0 Register (FP Rx Configuration Function)" on page 554 in Appendix A.

The Payload Last FP Rx index must have the bottom two bits (LSBs) set (that is, 0xmmmmmm11) as C-5 NP only supports 4Byte multiples for both payload and header. Correspondingly, the Payload First Index must have the two LSBs cleared (that is, 0xmmmmmm00).

The payload cannot end on Byte 3 of the cell, so the Payload Last index must be greater than 3.

As segments arrive, their headers are directed alternately to the header FIFOs belonging to the two Byte Processors. All payload is directed into a single payload FIFO.

Buffer Pool Configuration, BTag Allocation, and Buffer

As the FP Rx receives PDUs it must determine which BMU buffer they will be stored in. The FP Rx can be configured to use buffers in up to 4 of the BMU's 30 pools. Which of the 30 BMU pools the FP Rx will use is configurable.

To prepare for incoming PDUs, the FP Rx keeps a store of BTags for each pool that it is using. It can store a maximum of 256 BTags (8 blocks of 32 BTags) per pool. When the FP Rx is enabled, it will request BTags from the BMU to fill its store for each pool up to a configured maximum for each pool. As PDUs are received and BTags used, the FP Rx BTag stores will be depleted. When the number of BTags in a store drops below a configurable

threshold, the FP Rx will begin requesting more BTags from the BMU and will continue to do so until it has filled to its configurable maximum. If the FP Rx requests BTags from the BMU and the BMU cannot satisfy that request, a statistics register will be incremented and the "No BTags available from BMU" interrupt error (if enabled) will be sent to the XP. The FP Rx will continue to request until the request is satisfied. If the request cannot be completed after 16 attempts, a "BTag allocation timeout" interrupt error (if enabled) will be generated.

FP Rx buffer pool configuration is done using the Pool Configuration registers. For details about the specific registers, see "Buffer Pools" on page 101 in Appendix A.

When the first segment of a PDU arrives, a Byte Processor will extract the PDU length from the header. This length will be used by the FP Rx to select a BMU buffer for the PDU which is big enough for it. The FP Rx will compare this length to the size buffers for the four system pools that it has been configured to use. It checks the pools in order (from FP Rx pool 0 to FP Rx pool 3) and when the first match is found, it will try to use a buffer from that pool. If there are no BTags available for that pool, that PDU will be dropped.

Because the pools will be checked in sequential order and the PDU assigned based on the first fit, the FP Rx should be configured to use buffer pools with monotonically increasing sized buffers. For example, FP Rx pools 0-3 might be assigned to buffer pools with buffers of size 64B, 256KB, 2KB, and 64KB respectively.

The BMU does not support a buffer size of 128B.

If the length of the PDU cannot be extracted from the first segment, the FP Rx can be configured to use a default PDU length. If this default length is used, all PDUs will be assumed to be this length for purposes of buffer selection. As PDU payload is received, it will be stored in a buffer until a last segment for the PDU is received.

The entire contents of that last segment will be stored in the buffer because the FP Rx has no way of knowing how much of the segment's payload is valid. The use of default PDU length is configured via the RxFCE Configuration1 register. For details about the specific register, see "RxFCE_Configuration1 Register (FP Rx Configuration Function)" on page 561 in Appendix A.

The PDU length check must also be disabled if this feature is used.

Storing the Payload

It is required that PDU segments be delivered to the FP Rx interface in order. Payload Bytes for a PDU are always stored in a BMU buffer in the order that they are received on the FP

interface beginning at offset 0 in the buffer. When writing payload to a BMU buffer, the FP does not set the out-of-band bits of the BMU buffer to meaningful values, so they cannot be used by any transmitting CP.

Microcode Processing of Headers

Because header formats vary from fabric to fabric and application to application, the FP Rx includes a microcoded Byte Processor, similar to those in the Channel Processors, to process the header for each segment. In fact, there are two Byte Processors in the FP Rx. Headers for consecutive segments are processed alternately by one of the two. Only a single microcode program can be loaded for both Processors, however there are ways microcode execution can be made different for each Processor. The memory map of the registers accessible by each Byte Processor is shown in Figure 41. Refer to the C-Ware Microcode Programming Guide, Part number: 4-017, Most recent publication date: April 19, 2001

The FP RxByte Processor is analogous to the RxSync Processor in the SDP.

Figure 41 RxByte Processor Memory Map

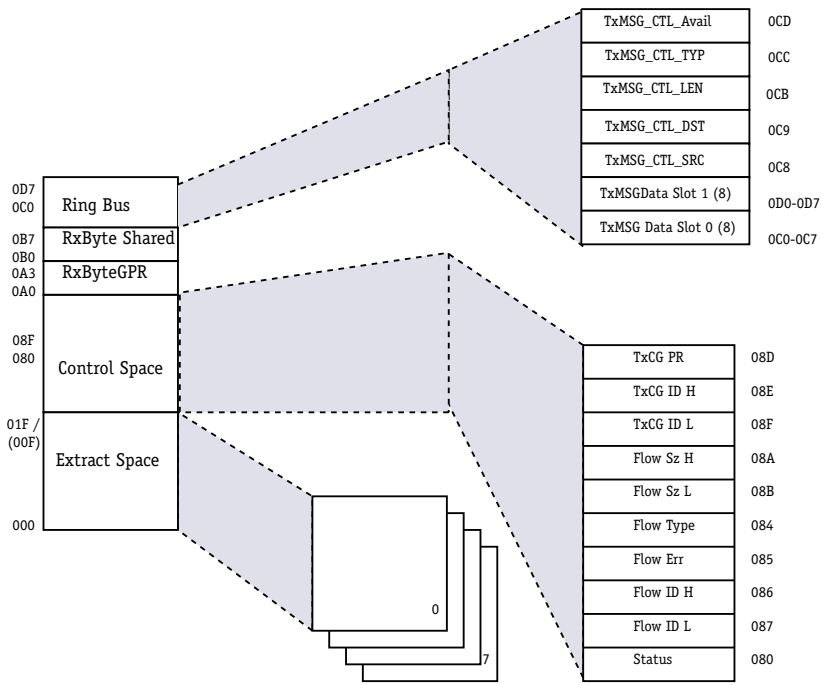


Table 28 RxByte Processor Memory Map Summary

Global Address Offset from 0xBDE04000	FDP Offset	Register Name	Register Description	Access		
				GB	FDP	HW
000h-07Fh, 200h-27Fh	000-01Fh [1]	Extract Space	Extract registers0/1 used for descriptor dats.	R/W	W	R
0090h / 0290h	080h	RxStatus	Status register 0 / 1[2]	R/W	R/W	R/W
094h-097h / 0294h-0297h	084h- 87h	Flow Seg	Flow Segment Information 0 / 1 [2, 3]	R/W	R/W	R
	086h–087h	Flow_Seg[15:0]	PDU ID			
	085h	Flow_Seg[[17:16]	Flow Error			
	084h	Flow_Seg[26:24]	Flow Type			
09Ah-09Bh / 029Ah-029Bh	08A-08Bh	Flow Size[15:0]	Flow Size 0 / 1 [2, 3]	R/W	R/W	R
09Ch-09Fh / 029Ch-029Fh	08Ch-08Fh	TCGS	Tx Congestion 0 / 1 [2, 3]	R/W	R/W	R
	08Dh	TCGS[21]	Pause / Resume (1 = pause)			
	08Eh-08Fh	TCGS[15:0]	Flow ID			
0628h-062Bh (SBP0) 062Ch-062Fh (SBP1)	0A0h-0A3h	RxByte GPR	RxByte General Purpose Config Reg. (4 bytes for each SBP0 and 1) [3]	R/W	R	N/A
0660h-0667h	0B0h-0B7h	RxByte SR	RxByte Shared Registers (8 bytes mapped to both SBP0 and SBP1) [3]	R	R/W	N/A
440h	C8h–CDh	TXMSG0 _CTL	Ring Bus0 (TxMsg) control [4]	R/W	W	N/A
	0CDh	TXMSG0_ CTL[31]	Ring Bus0 Avail[5]			
	0CCh	TXMSG0_CTL[19:18]	Ring Bus0 Type			
	0CBh	TXMSG0_CTL[17:15]	Ring Bus0 Length – Bit 17 set to 0			
	-	-	RB0 Seq set to datascope by hardware			
	0C9h	TXMSG0_ CTL[9:5]	Ring Bus0 Destination			
	0C8h	TXMSG0_CTL[4:0]	Ring Bus0 Source			
460h / 464h	C0 – C7h	TXMSG0_D0H/L	Data (*See WARNING) [6, 3]	R/W	W	N/A
480h / 484h	D0 – D7h	TXMSG0_D1H/L		R/W	W	
448h	C8h-CDh	TXMSG1_CTL		R/W	W	
468h / 46Ch	C0 – C7h	TXMSG1_ D0H/L		R/W	W	
488h / 48Ch	D0 – D7h	TXMSG1_ D1H/L	NOTE: maps to RB1 D1 H/L	R/W	W	
450h	C8h-CDh	TXMSG2_ CTL		R/W	W	
470h / 474h	C0 – C7h	TXMSG2_ D0H/L		R/W	W	
490h / 494h	D0 – D7h	TXMSG2_D1H/L	NOTE: maps to RB2 D1 H/L	R/W	W	

Table 28 RxByte Processor Memory Map Summary (continued)

Global Address Offset from 0xBDE04000	FDP Offset	Register Name	Register Description	Access		
458h	C8h-CDh	TXMSG3_ CTL		R/W	W	N/A
478h / 47Ch	C0 – C7h	TXMSG3_D0H/L		R/W	W	
498h / 49Ch	D0 – D7h	TXMSG3_D1H/L	NOTE: maps to RB3 D1 H/L	R/W	W	

1 16 or 32 bytes mapped to FDP by datascope index

2 0 / 1 refers to Processor register set 0 and 1 respectively.

3 The FDP uses little endian addressing for the bytes within this register (least significant byte is associated with the highest address).

4 The TXMSG_CTL register fields are spread out over a range of 6 bytes from the FDP perspective (as listed), but note that global accesses to the register can be done with one 32-bit access.

5 Although the TXMSG_CTL registers are in general not readable by the FDP, the exception is the Ring Bus Avail bit, which can be read as bit 0 from byte offset xCD.

6 Ring bus Registers 0/1 map to RxByte0, Ring bus Register 2/3 map to RxByte1 and in each case are indexed by datascope. Refer to your SDP Programmers guide for byte mapping. See WARNING, below.

Warning: *Because datascope selects the actual RB register set, the RxByte Processor cannot pre-initialize certain static fields. The RxByte should write all dynamic registers, for example index / key for lookup, on a per segment basis. Static fields, that is, command type, return length etc, should be initialized by the XP or host. Refer to SDP Programmers guide for byte mapping.*

The minimum requirements for FP Rx microcode are that it:

- Wait for datascope ownership
- Set up a PDU ID and segment type for the segment
- Remove all Bytes for that header from the header FIFO
- Flip ownership for that datascope

Other functions FP Rx may perform include:

- Setting up a PDU length for first or only segments, if a default PDU length isn't used
- Launching a TLU lookup for use in descriptor building and enqueuing
- Processing per-queue flow control information and sending flow control messages to the FP Tx
- Storing header data in extract space for descriptor building

Generally, FP Rx microcoding is done much like microcoding for an SDP Byte Processor. The FP Rx Byte Processor is capable of the same sequencing and ALU operations as the SDP Byte Processor. The unique aspects of the FP Rx Byte Processor are:

- The external test conditions
- Outputs
- There are 96 control store entries (as opposed to 64 in the SDP Byte Processor). The CAM has 24 entries just like the SDP.

External Test Conditions

The external test conditions available to the FP Tx Byte Processor are:

0 Input data valid

The **input data valid** test condition indicates that the Byte feeding the Processor is valid.

1 Token

The **token** test condition can be used to test for ownership of a token passed between the two Byte Processors.

2 Not used

3 Not used

4 Not used

5 First/last header Byte

The **first/last header Byte** test condition can be used to test if the Byte currently being unloaded from the header FIFO is the first or last Byte of the header. It is configurable whether this test condition indicates the first or last Byte of the header via the Rx Byte Processor End Of Header field of the *RxDS Configuration* register. For details about the specific register, see "RxDS_Configuration Register (FP Rx Configuration Function)" on page 555 in Appendix A.

6 Drop mode

The **drop mode** test condition indicates that a cell needs to be dropped because the payload FIFO is full (504Bytes). See "Rx Drop Mode" on page 177.

7 Control word

The **control word** test condition indicates that the Byte currently being unloaded is coming from the control FIFO, not the header FIFO. The control FIFO is only used in PowerX mode so details of its usage are provided in the PowerX section.

Datascope

The FP Rx provides each Byte Processor with a number of datascopes or contexts into which it can store data. If 12- or 16Byte descriptors are being used, each Processor has 8 datascopes available to it. If 24- or 32Byte descriptors are being used, each Processor has 4 datascopes available to it. There are times when datascopes are not available to a Byte Processor because FP Rx has not finished consuming its contents. Because of this, microcode must wait until it is granted ownership of a datascope before it begins processing a header. Microcode tests for datascope availability using the Ownership bit of the RxStatus register. For details about the specific register, see "RxStatus1 Register (FP RxByte Processor Function)" on page 548 in Appendix A.

When FP Rx microcode has finished processing a header, it passes ownership for the datascope to hardware by setting the Ownership bit of the RxStatus register. This indicates that all of the data has been set up and the hardware can begin processing the segment. The hardware uses the datascope number as an index into extract space and TLU response space. The datascope also serves to keep the PDUs in order. Once the hardware is done handling the payload and enqueuing the PDU, the datascope is freed up.

Performance Requirement

Because there is no back pressure mechanism related to header FIFOs filling, FP Rx microcode must be designed to process headers in the amount of time it takes two (because only every other header is directed to a given Byte Processor) segments to arrive at the FP Rx interface. The header FIFOs can hold 64Bytes each to withstand temporary congestion.

The number of microcode instructions that can be executed in the time it takes a segment to arrive on the interface is a function of:

- Segment size (or minimum number of fabric clock cycles between segments)
- Fabric width
- Core clock speed - the Byte Processor operates on this clock
- Fabric clock speed
- Number of Byte Processors (2)

```
minimum number of core clocks per segment =
minimum number of fabric clocks between first cycles of
segments x
(1/fabric clock frequency) x
core clock frequency x
2
```

This minimum number of core clocks per segment constrains the number of Byte Processor instructions that can be executed to process a header. The minimum number of fabric clocks between first cycles of segments might be:

- The number of fabric clock cycles per segment, for a fabric protocol with fixed-length segments
- The number of fabric clock cycles per minimum-sized segment, for a fabric protocol with variable-length segments

If a segment-spacing mechanism is part of a fabric's protocol (for example the minimum SOF-to-SOF gap requirement of the PowerX protocol), it would be the number of fabric clock cycles, guaranteed by this mechanism, to be between segments.

For the PowerX protocol, because the Byte Processors must also process per-queue flow control messages which can be interspersed between and within segments, the microcode performance requirements must take the presence and frequency of these flow control messages into account. The calculation of this performance requirement is beyond the datascope of this document.

Another performance requirement is avoiding datascope overrun. If the number of outstanding cells in the FP Rx is more than the number of datascopes, and the microcode needs to obtain a new datascope, that is, not an idle cell, then the microcode would have to stall, and the header FIFOs would eventually overflow, which is not allowable. This type of datascope overrun can be avoided by configuring the payload FIFO XOFF threshold, so link-level flow control is applied to the fabric before a dangerous number of cells is received.

Setting Up Control Information

FP Rx microcode must set up certain control information for each segment. At a minimum it must set up a PDU ID and a segment type.

The PDU ID is a 16bit value that associates the payload within the segment with the payload from other segments from the same PDU.

There is a configurable mask which the FP Rx will apply to the PDU ID before this association is done. The PDU ID mask is configured via the *PDU Mask* [15:0] field of the

RxFCE Configuration0 register. Typically the PDU ID is extracted from the header. For details about the specific register, see "RxFCE_Configuration0 Register (FP Rx Configuration Function)" on page 559 and "RxFlowSeg1 Register (FP RxByte Processor Function)" on page 549 in Appendix A.

The segment type indicates whether the segment is a first, middle, last, or only segment for the PDU. Typically the segment type is extracted from the header.

If the segment is a first or only segment, a PDU length must also be set up if the default PDU length is not being used. This should indicate the number of Bytes of payload in the PDU and is used to find an appropriately sized buffer. Typically, the PDU length is extracted from the header of a first or only segment. For details about the specific register, see "RxFlowSz1 Register (FP RxByte Processor Function)" on page 550in Appendix A.

Some applications may not be able to provide a PDU length in first and only segment headers. For those applications, the FP Rx can optionally use a default length value for all PDUs and thus, the PDU length need not be written by microcode. In this mode, the same PDU length is presumed for all PDUs and the default length should be set up to be greater than or equal to the maximum PDU size. This option is configured via the *RxFCE Configuration1* register. For details about the specific register, see "RxFCE_Configuration1 Register (FP Rx Configuration Function)" on page 561 in Appendix A.

When the default PDU length option is used, the FP Rx error check which detects the premature arrival of a last segment, must be disabled because last segments may appear to be premature because the PDU length is assumed to be a maximum length. Also, when this mode is used, all PDUs are delivered to the same BMU pool because the length being used to select a pool will always be the same.

Writing to Extract Space

The FP Rx Byte Processors can write Bytes to extract space to be made available for descriptor building. The number of Bytes available per datascope depends on the descriptor size. If the FP is configured to use 12-, or 16Byte descriptors, 16Bytes of extract space are available. Otherwise, 32Bytes of extract space are available.

Because the Descriptor Build Engine (DBE) may start building a descriptor as soon as it receives ownership, microcode must finish writing all Bytes to extract space before passing ownership of the first segment to the hardware. In other words, extract space cannot be written for middle or last segments of PDUs.

TLU Lookups

The FP Rx can launch requests on the ring bus to perform TLU (Table Lookup Unit) lookups. The requests are made via the *MSG_CTL and MSG_DATA0* registers (plus

MSG_DATA1 in the case of 16Byte requests). Refer to "Table Lookup Unit" on page 245, Chapter 6. Many of the register fields can be statically configured via global writes: Type, Length of request (8 or 16Bytes), Source Ring Bus Node, Destination Ring Bus Node. Fields which are unique to each segment can be written by the RxByte Processor. Examples of these are the key and index, which are a part of the ring bus data (that is, *MSG_DATA0*). For more information about the standard TLU lookups please refer to the TLU documentation. After filling out any necessary request fields and checking the Available bit in the ring bus control register (*MSG_CTL_AVAIL*), the microcode can clear the Available bit to launch the request on to the ring bus.

The responses to the TLU lookups will come back via a dedicated TLU->FP interface, as opposed to via the ring bus. The data will be placed in response memory (256Bytes) where it can be accessed by the Descriptor Build Engine (DBE).

TLU Lookup Programming Guidelines

Since the FP does not use the ring bus for responses (only requests), do not send any ring bus messages to the FP. A message targeting the FP will circulate the ring indefinitely, and enough of these would degrade ring bus performance to zero. For this reason, the only node to which the FP should send messages is the TLU, which has a dedicated response interface to the FP. The exception is proxy requests; the FP can safely send these to any node on the ring bus because there is no response.

- Only one TLU operation can be performed per-segment (that is, the sequence number of the request and the index into response memory are determined by the datascope, and there is only one datascope per segment).
- TLU operations can only be launched on first/only segments, because those are the only segment types for which a descriptor is built.
- TLU operations, if used, must always be launched on every first/only segment. If the microcode wants to discard a segment via the Discard or Error indicators, it should set the segment type to middle and not launch a TLU operation. If the segment type remains first/only, a TLU operation would have to be launched.
- The response size can be 8, 16, or 32Bytes. This response size is a function of fields within the lookup request data (that is, MSG_DATA0) - refer to TLU documentation. The size of the request and size of the response are independent, but the response size cannot be greater than the size of the response memory (256Bytes) divided by the number of datascopes. For example, with a 16-datascope configuration (that is, 16Byte descriptors), the response size can be 8 or 16Bytes. For 8-datascope configurations (that is, 24 or 32Bytes descriptors), the response size can be 8, 16, or 32.

Do not use 32Byte responses with 16 or 24Byte descriptors.

General Purpose Registers

There are 8Bytes of shared general purpose registers which are available to both Byte Processors, plus two 4Byte configuration registers (one for each Processor). All of these can be initialized via global writes.

Both Processors have read and write access to the 8Bytes of shared space. The Processors can read any Byte at any time. The microcode should prevent the Processors from writing the same Byte at the same time, or else the resulting value will be undefined. All other combination of writes are allowed, including simultaneous writes to different Bytes.

The 4Byte configuration registers act as separate, private storage for each Processor. Typically, they are used to customize the execution behavior of one Processor from the other. They are configured via global writes, and can be read (not written) by the associated Byte Processor.

For details about the specific registers, see "TxFDP_CTL0 Register (TxByte General Purpose Function)" on page 544 and "TxFDP_CTL1 Register (TxByte General Purpose Function)" on page 545 in Appendix A.

Discarding Segments

Sometimes the FP Rx Byte Processor may want to discard a segment. A common use for discarding segments would be to discard "idle" segments which only convey flow control information.

To discard a cell, FP Rx microcode must:

- Set the PDU ID to be some value that does not match any PDU ID that the C-5 NP might use for valid PDUs
- Set the segment type to be a middle segment
- Set the *FLOW_DISC* or *FLOW_ERR* bit in the RxFlowSeg register. For details about the specific register, see "RxFlowSeg0 Register (FP RxByte Processor Function)" on page 546 in Appendix A. These bits both cause the segment to be dropped, and increment separate counters in the FP Rx statistics registers.
- Pass ownership of the segment to hardware

The effect of discarding a segment is that any payload associated with the segment in the payload FIFO will be dropped (not written to any BMU buffer).

Token Passing

If needed for synchronization purposes, a token can be passed between the two Byte Processors. After reset, the token is owned by Byte Processor 0. To pass the token to the other Processor, a Byte Processor should set *Token Out*, which is bit 2 in the Processor's control register. The *Token Out* bit must have been previously cleared, so that there will be a 0->1 transition. Ownership of the token bit can be tested by using external test condition bit 1.

Rx Drop Mode

When the amount of data in the payload FIFO passes the configurable XOFF threshold, link-level flow control can be applied by the FP Rx back to the fabric (refer to "Fabric to C-5 NP Link-Level Flow Control" on page 192). Despite this, some applications such as PRIZMA will continue to transmit idle segments. Thus it is possible for the payload FIFO to continue filling up well past its threshold. When it fills up to the maximum of 504Bytes, Drop Mode is activated, causing subsequent payloads to be dropped.

In Drop Mode, the Byte Processors will continue to process incoming headers. There will be no payload associated with these headers, so the microcode must not advance the datascope (which would pass the ownership of the segment to the remainder of the hardware pipeline). Thus the microcode needs to test for Drop Mode via external test condition 6 (see"External Test Conditions" on page 171), before deciding whether to advance the datascope.

To keep the headers and payload in sync, be sure that a transition into Drop Mode does not occur while a header is being received. This can be guaranteed through a proper configuration of the header/payload delimiter registers, as follows:

1. The header and payload regions for idle segments cannot be configured to overlap.
2. The microcode can test for the Drop Mode external test condition at any point during the header processing, with the following restriction.

   When Drop Mode is first engaged by an incoming idle segment, the test condition is invalid until 2 words after the last word of that segment's payload region. In the meantime, the microcode can start processing the header of the next idle segment, and needs to know whether Drop Mode is engaged or not. The easiest way to handle this is to configure the payload region for idle segments to end before the true end of the segment. This typically occurs anyway, since only a portion of the idle segment needs to be treated as payload. With this configuration, the Drop Mode condition will be guaranteed to be valid in time to flag the next header Bytes appropriately. Alternatively, if the payload region extends to the end of the idle segment, then Drop Mode will not be valid for the first word of the next header, so the microcode would

have to avoid testing for Drop Mode on Bytes within that word. Instead, Drop Mode could be tested starting with the second word of the header, implying that the header region for idle segments would have to be greater than 1 word (that is, 8Bytes instead of 4).

Descriptor Build Engine Microcoding

The Descriptor Build Engine (DBE) is responsible for composing descriptors and specifying the queue onto which descriptors will be enqueued. The DBE begins constructing a descriptor each time ownership of a first or only segment is passed to hardware. The DBE has the option to use and manipulate the following types of data to compose the descriptor:

- Extract space contents for the current datascope. This is typically header data, which was copied to extract space by a Byte Processor.
- TLU response (if any) for the current datascope.
- BTag and pool information for the PDU
- Literal field

Descriptor Build Sequence Programming

The descriptor build sequence is programmed using DBE microcode. The microcode control store has room for 64 52bit microinstructions. The instruction set is limited to variations of Move Data (from Source to Destination) with bit manipulation capabilities (mask, shift). Operations may be performed on words, half words or Bytes. A literal field can be used for mask operations, or to write absolute data of up to 16bits. No jump, branch, or loop control exists, so all descriptors are built with the same straight-line path through the microcode. When the descriptor is completely built, the microcode must write the queue number using a Move Special instruction with the Destination Index set to 0. This write operation signals to the hardware that the descriptor is ready to be enqueued. Finally, the last microinstruction should set the opcode to NOP and assert the Restart bit.

When the descriptor has been built, and the entire PDU has been received without an error, the descriptor is enqueued to the QMU. The queue number is not part of the descriptor but is sent to the QMU along with the descriptor.

Extract and Response

Each of the extract and response areas is divided into multiple datascopes to allow pipelining. The Extract space is either 16 or 32Bytes per datascope depending on the configured descriptor size, and the Response area is either 8, 16 or 32Bytes depending

upon the response size configuration. A descriptor build operation will begin once the associated Byte Processor has passed extract ownership to hardware (that is, written a 1 to the msb of the status Byte). Additionally, if configured to do so, the DBE will wait until the TLU has provided a response for the given datascope. 32Bytes of descriptor memory are always available for writing by the DBE, however only the amount of the descriptor specified by the size, starting at relative offset 0 (in the datascope), is transferred to the QMU.

Handling TLU Errors

If the DBE algorithm uses TLU response data, it should handle the case where the TLU lookup fails. A failed lookup has no valid response data associated with it and has an error returned to it by the TLU. Two methods for handling this are described below:

1 Default Queue Feature

For applications which derive the queue number from TLU response data, the default queue feature allows a configurable default queue number to be used in the case of a TLU lookup error. To enable this feature, configure the default queue number and assert the enable bit in the *RxFCE Configuration2* register. The register description can be found at "RxFCE_Configuration0 Register (FP Rx Configuration Function)" on page 559 in Appendix A.

2 Drop On TLU Error

If desired, PDUs can be dropped when a TLU lookup error is encountered. To implement this feature, the DBE should perform a Move Special instruction with the Source Index set to 100. The effect of this instruction is to set the Drop bit if there was a TLU error. This instruction can occur anywhere within the DBE sequence, as long as it is before the queue number write and final Restart instruction. When the DBE finishes building the descriptor and writes the queue number, the state of the Drop bit determines whether the descriptor gets dropped or enqueued.

The format and definition of the DBE microcode fields are shown in Table 29 and Table 30 on page 180.

Table 29 DBE Command Format

Bit Position	51	50 49	48 47	46 44	43 39	38 34	33 30	29 16	15 0
Field Name	**Restart**	Op-Code	Desc Size	Source	Src Indx	Dest Indx	Src Shift	Reserved	Literal

Table 30 DBE WCS

Field Name	Bit Position	Description
Restart	51	A **microinstruction** with Restart=1 causes the microinstruction pointer to restart the algorithm for the next descriptor. Restart=0 has no effect on the descriptor build process. Since the Restart effectively terminates the instruction flow for a given descriptor, this bit should only be used on the last microinstruction (which should also set Op-Code = NOP) and after the queue number has been written to the flow table (see Move Special). Refer to *C-Ware Microcode Programming Guide*, Part number: 4-017, Most recent publication date: April 19, 2001
Op-Code	50:49	**Operation** to be performed on Descriptor: 00 Move/Write from Src -> Dest 01 Move Mask/Read Modify Write Uses the Mask field (which is also the Literal field) to determine which bits of the destination descriptor data should be updated. This operation can only be used with Byte and half-word operand sizes. For each bit in the mask that is a '1', the corresponding bit will be updated in the destination data. For each '0', the bit in the destination data will retain its previous value. For Byte operations, bits 7:0 of the mask field act as the mask. For half-word operations, all 16 bits of the mask field are used as the mask. 10 Move Special - see 2 cases below If Destination Indx = 00, the hardware takes this instruction to mean the descriptor has been built. The descriptor is saved in descriptor memory, and the 9bit queue number and Drop Packet bit are saved in a flow table memory, ready to be enqueued to the QMU. If Destination Indx = 04, Set Drop Packet equal to bit 24 of the source data. 11 No-op
Operand Size	48:47	**Operand Size** 00 Word 01 Byte 10 Half Word 11 Reserved

Table 30 DBE WCS (continued)

Source	46:44	**Data** can be sourced from the following: 000 Extract Space 001 TLU Response (note: TLU responds directly to TLU/RB requests) 010 Buffer Memory Info Source data = {11'b0, Pool ID[4:0], BTag[15:0]} 011 Literal The location of the literal data within the 32bit source data is dependent on the operand size. Literals can only be used for Byte and half-word operations. Byte operations: Source data = {Literal[7:0], 0x000000} Half-word operations: Source data = {Literal[15:0], 0x0000} 100 Special, IDX [0] = TLU_ERROR in bit 24 of the data word. NOTE: TLU error aligns with drop packet to allow ease of use 101, 110, 111 Reserved
Src Indx	43:39	**Source Index** – Byte offset into source (e.g. extract space offset, buffer offset)
Dest Indx	38:34	**Destination Index** – Byte offset into destination descriptor. The destination index is also used to select between the two types of Move Special operations (see above).
Src Shift	33:30	`SrcShift`[33] = 1 indicates shift left, 0 indicates shift right `SrcShift`[32:30] = Number of bits to shift. Range is 0 to 7. NOTE 1: This is a shift and not a rotate operation; bits do not wrap. For Shift Right, 0's are shifted into the MSB. Likewise for Shift Left, 0's are shifted into the LSB. NOTE 2: Shift operations occur prior to mask operations.
Reserved	29:16	Reserved
Literal/ Mask	15:0	**Literal/Mask Field** - can be used as source data (when Source=011), or as a mask for a Move Mask operation. In either case, this field can only be used with Byte or half-word operand sizes. When used as literal source data, the location of the literal data within the 32bit source data is dependent on the operand size. Byte operations: Source data = {Literal[7:0], 0x000000} Half-word operations: Source data = {Literal[15:0], 0x0000} When used as a mask, this field determines which of the bits in the destination data should be updated and which should retain their previous value. Only destination bits which have the corresponding mask bit set to 1 will be updated.

Alignment

The source and destination addresses must be aligned to a 32bit boundary for full word operations, and aligned to a 16bit boundary for half-word operations. The addresses can use any alignment for Byte operations. See Table 31 on page 182.

The table shows what the destination data would look like after the write operation. Based on the alignment and operand size, certain destination Bytes are updated, and others retain their previous value.

Assume a write operation is performed with source data = AABBCCDDh.

Table 31 DBE Operand Alignment Examples.

Operand Size	SRC IDX[1:0]	DST IDX[1:0]	DATA DST (hex)[1]
Byte	00	00	AA------
		01	--AA----
		10	----AA--
		11	------AA
	01	00	BB------
		01	--BB----
		10	----BB--
		11	------BB
	10	00	CC------
		01	--CC----
		10	----CC--
		11	------CC
	11	00	DD------
		01	--DD----
		10	----DD--
		11	------DD
Half Word	00	00	AABB----
		10	----AABB
	10	00	CCDD----
		10	----CCDD
Word	00	00	AABBCCDD

1 A dash in the DST field means no change to existing destination data.

Bit shift operation

The shift operation is a true shift as opposed to a rotate because zeroes are shifted in, (either left or right as selected by the MSB of the shift field), as bits of the operand field are shifted out.

The bits which get shifted outside of the operand source field get dropped. The shift operation has no effect on which destination bits get updated (the destination data is a function of the operand size, the destination index, and optionally a bit mask).

Enqueuing

After the last, or only segment of a PDU has been received and processed by a Byte Processor and a descriptor has been built by the DBE, the FP Rx will enqueue the descriptor. The FP Rx only does unicast enqueues; it never does multicast enqueues. The descriptor weight accompanying the descriptor always has a value of 1.

The FP Rx will not enqueue a descriptor if the DBE has indicated that the PDU should be dropped or, if some other error has been detected with the PDU in such a way that the PDU will be dropped.

The FP Rx can be configured to enqueue descriptors to a default queue if a TLU error occurs. This default queue is used instead of the queue indicated by the DBE. This is done by setting up the default queue number and default queue enable fields in the *FCE Configuration2* register. For details about the specific register, see "RxFCE_Configuration2 Register (FP Rx Configuration Function)" on page 562 in Appendix A. See "Default Queue Feature" on page 179.

Interrupts

Seven events in the FP Rx are logged in the *Rx Interrupt Event* Register. Each register bit will remain asserted until written with a '1' ("write 1 to clear"), for instance by an interrupt handler running on the XP. The *Interrupt Enable Mask* register determines which events cause an interrupt are sent to the XP.

These topics are covered in the following sections.

- Error Status FIFO Full
- Parity Error
- No BTags available on allocate
- Buffer Write Errors
- BTag Programming Error
- BTag ECC Error

- BTag Allocation Retry Timeout

Error Status FIFO Full

Indicates that all 32 entries of the error status FIFO are in use. The FIFO contains the error status for each outstanding cell, plus a record of any out-of-frame errors that occurred (either a parity error on the first word of a cell, or a parity error on a PowerX control word). Only one out-of-frame error can be logged between each cell. Thus the FIFO would only fill if there were 16 outstanding cells with 16 out-of-frame errors. In practice, the payload FIFO XOFF threshold should prevent the number of outstanding cells from reaching 16; thus the error status FIFO should never become full.

Parity Error

A parity error was detected on the receive interface.

No BTags available on allocate

A BTag allocation request failed because none were available from the BMU.

Buffer Write Errors

The only condition which can cause a buffer write error would be the BMU being unable to satisfy repeated buffer write attempts because it is too busy.

BTag Programming Error

Non-existent memory error - this would only occur if the BMU were misconfigured to be storing BTags in a non-existent memory location.

BTag ECC Error

ECC error - BMU detects an ECC error when it reads a block of BTags out of SDRAM

BTag Allocation Retry Timeout

Retry timeout - BMU is unable to satisfy the allocate request because it is too busy.

Error Handling and Statistics

The FP Rx collects a variety of statistics on traffic and events. These statistics are available through nineteen 32bit registers that can be read or written at any time. The available statistics are shown in "RxFP_Statistics Registers (FP Rx Statistics Function)" on page 572 in Appendix A.

Enqueue Failures

If the FP Rx performs an enqueue and the QMU indicates that the enqueue operation failed, the PDU is dropped because the FP Rx does not retry the enqueue operation and it deallocates the BTag back to the BMU.

Segment Sequencing Errors

Generally, the FP Rx expects to see a first segment for a PDU, some number of middle segments, and a last segment. If a PDU is small enough, it might be transmitted with a single only segment, or perhaps first segment followed directly by a last segment. There are a number of error scenarios related to these expected sequences that the FP Rx will detect and handle.

- If a middle or last segment arrives for a PDU, without any preceding first segment, that segment will be dropped.
- If at the end of a last segment the amount of payload received is less than the expected PDU length (as indicated by the PDU length in the header of the first segment), the entire PDU will be dropped. This PDU length check bit can be disabled.
- If a first segment arrives for a PDU, when the FP Rx is expecting a middle or last segment for the PDU, that segment and the PDU that was in progress will be dropped.

It is possible that some number of middle segments and the last segment of one PDU can be dropped and then the first and some number of middle segments are dropped from the subsequent PDU with the same PDU ID in such a way that, to the FP Rx, these segments seem to form a coherent PDU and no error would be detected. This event is highly improbable.

When PDUs are dropped, the enqueue operation is not performed and the BTag will be deallocated back to the BMU.

Parity and CRC Errors

If parity checking is enabled and a bus parity error is detected for any cycle of a segment, that segment will be dropped. See "Common Components of the Programmable Processors" on page 62.

If a CRC checking is enabled and a CRC error is detected for a segment, that segment will be dropped. See the CRC section for details of CRC usage and configuration.

If individual segments are dropped because of CRC or parity errors, effectively this looks like missing segments to the FP Rx. Ultimately, subsequent segments will appear to cause a segment sequencing error as described above and PDUs will be dropped.

Headers are always passed to Byte Processors, even if they have CRC or parity errors in them. The Byte Processor has no way of knowing if the header is in error or not so it must assume that it is not.

FP Functionality

Initialization

The following sequence should be performed to initialize the FP.

1. While keeping the FP Rx and FP Tx disabled, set up the configuration registers. Load all FP control stores (microcode). Configure and enable all of the resources used by the FP (BMU, QMU, TLU).
2. Enable the *FP Rx (RxFP Enable* register) and *FP Tx (TxFP Enable* register).
3. Once the FP Rx has had time to acquire BTags for all of its pools, set the CFI Enable bit in the *RxFI configuration* register.
4. Set the *CFI Enable* bit in the *TxFI configuration* register.

Once initialized, the FP cannot be dynamically reconfigured and reinitialized. There must be a full C-5 NP reset to reinitialize the FP.

Another initialization issue has to do with SDRAM. When the FP Rx receives payload and writes it into SDRAM, the write is done with 16Byte granularity. This means the PDU payload could end at address 0, 16, 32 or 48 within the last 64Byte block. If the PDU is transmitted by an SDP, the payload is read in 64Byte blocks, so there may be some unitialized data in the last block. This uninitialized data can cause an ECC error. There are several ways to handle this case:

- Initialize SDRAM at startup *OR:*
- Disable ECC checking *OR:*
- Configure the FP Rx to always write in 64Byte blocks, via a bit in the *RxFCE Configuration2* Register called *Force 64Byte WRCB Transfers*.

Accessing the Tx Flow Control CAM

The internal CAM through which the FP Tx maps a per-queue flow control request to a queue number can be read and written, using the "TxFlowCam Register (FP Tx DeBug Function)" on page 541 in Appendix A. The default mapping preloaded into the CAM by hardware is simply a one-to-one mapping, such that the 7bit queue number equals the 16bit match value, for the range of 0 to 127. The procedure to configure the CAM is as follows:

1. Set the flow mask to 0xffff in the *TxFCE* Register.
2. Then delete the preloaded CAM entries by performing 128 writes to the *TxFlowCAM* register; on each write set the *DEL* (delete) bit and the next 16bit Match value.

Remember that the preloaded match values are simply 0 through 127.

3 Next, write the desired entries into the CAM by writing to the *TxFlowCAM* register, with the *WT* bit asserted. The 16bit match value and 8bit write data will be placed into the CAM.

Bit7 of the write data must be zero; bits [6:0] represent the 7bit queue number.

Accessing the FP Rx Descriptor Build Engine Write Control Store (WCS), Byte Processor WCSs, and Byte Processor CAMs

The Byte Processor control stores and CAMs in the FP Rx and Tx can be written and read. Physically, in both the FP Rx and Tx, there is a separate control store for each of the two Byte Processors, but the same data is written to both Byte Processor control stores simultaneously; that is, both FP Tx control stores and CAMs contain the same data and both FP Rx control stores and CAMs contain the same data.

Accessing the FP TxByte Processor WCSs and CAMs

The FP Tx Byte Processors each contain a WCS and a CAM (not to be confused with the FP Tx flow control CAM). Each of these WCSs and CAMs must be loaded by internal scan chains accessible via the TxWCS_CAM register. As with the SDPs these memories must be loaded via an internal scan chain accessible in Global Address Space.

Accessing FP Rx WCSs and CAMs

The FP Rx Byte Processors each contain a WCS and a CAM. In addition, the Descriptor Build Engine (DBE) contains a WCS. Of these three WCSs and two CAMs, all may be read via scan, but only the CAMs may be written via scan. The WCSs, are written via a special Byte writing mechanism. Both the scan access and Byte write access is provided via the "RxWCS_CAM Register (RxWCS_CAM Function)" on page 557.

Table 32 FP Rx WCS and CAM Access

Store Type	Description	Scan Access Rd	Scan Access Wr	Byte Wr Access
RxByte WCS (0 & 1)	RxByte Processors 0 and 1, 96-word Writable Control Store	Y	N	Y
RxByte CAM (0 & 1)	RxByte Processors 0 and 1, 24-entry CAM	Y	Y	N
DBE WCS	Descriptor Build Engine WCS	Y	N	Y

The *RxWCS_CAM* Register allows the writing and reading of the FP RxByte WCS, CAMs and the Descriptor Build Engine WCS. The CAMs referred to are the Byte Processor CAMs (not be confused with the FP Rx PDU ID CAM, which does not need to be loaded by software).

For details about the specific register, see "RxWCS_CAM Register (RxWCS_CAM Function)" on page 557 in Appendix A.

Warning:

The mechanisms described below can only be used when the FP Rx is held in reset.

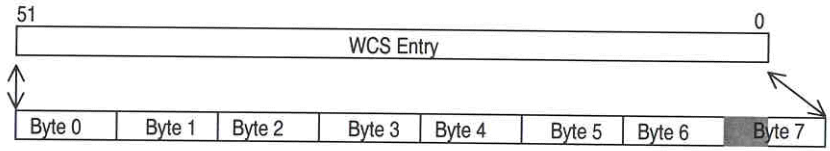
There are two RxByte Processors in the FP. When writing or reading from the associated stores, both are written and read at the same time. There is no mechanism to separately load RxByte0 WCS / CAM and RxByte1 WCS/CAM. This enforces RxByte0 to run the same Processor code / CAM data as RxByte1. The CAM must be written and read via scan as described below. The WCSs must be written via the Byte write mechanism described below. Data may only be read back from both the WCSs and CAMs via scan. This is intended for diagnostics use only (i.e. memory validation), as such the procedure is rather complex and not optimized for operational use. When the data is read back, via scan, the data from each of the two RxByte Processors is streamed back to bits *Scan_Out0* and *Scan_Out1* respectively. Data from the DBE is streamed to the DBE Scan Out bit.

The two FP Tx Byte Processors are loaded in parallel. That is, the two TxByte Processors run identical microcode.

For details about the specific register, see "TxWCS_CAM (WCS_CAM Function)" on page 539 in Appendix A.

Byte Write: To Byte write either the FP RxByte or the DBE WCS, the FP must be placed in reset. This ensures the Byte 'Address' counter is pointing to Byte 0. There is no direct ability to specify the addressing in RxByte or DBE WCS. The WCSs are loaded left to right, with the most significant Byte first. The most significant Byte of each word is only half used, with the most significant 4bits of this Byte as "don't care" values. Thus each WCS "word" is 7Bytes long as shown in the figure below.

Figure 42 Byte Load Sequence Map for WCS entry



Each of the FP-RxByte and DBE WCS has a Byte pointer that advances sequentially. The pointer starts with the most significant Byte (nibble) of the WCS, and increments through to the least significant Byte of the WCS moving from WCS location 0 through to WCS location 63, that is, left (msb) to right (lsb), bottom (addr0) to top (addr 63).

The Byte write hardware along with the scan chain organization has been designed and optimized for Byte write loading of the WCS (vs. scan loading). Specifically the CAM is first in the chain, followed by the WCS. This allows the CAM to be loaded first via scan with out the need to shift bits down the entire chain for the WCS (See Scan operations below). When the CAM is loaded in this fashion, the WCS will have undetermined data written to it. In this case it is expected that the WCS will be loaded after the CAMs have been initialized.

Scan Write: The CAM/WCS Scan Chain is 94bits long comprised of the following fields left to right where the right most bit of the right most field is shifted in first: CAM Addr (6), Cam Group(9), CAM Pattern(18), CAM Tag (9), WCS Addr (9), WCS Data(52). The CAMs are at the near end of the chain (i.e. all WCS fields are at the end of the chain).

The CAMs, if used, should be written prior to the WCS Byte loading. It is only required to scan the 42bits of CAM fields and the perform an Update operation via bit 2 of the *FP Rx WCS_CAM* Register. Scanning to the CAM may be done on a Random Access basis since the CAM Addr field selects the specific CAM location during the update operation. Both *RxByte0* and *RxByte1 CAMs* are updated simultaneously.

Figure 43 RxByte Scan Chain

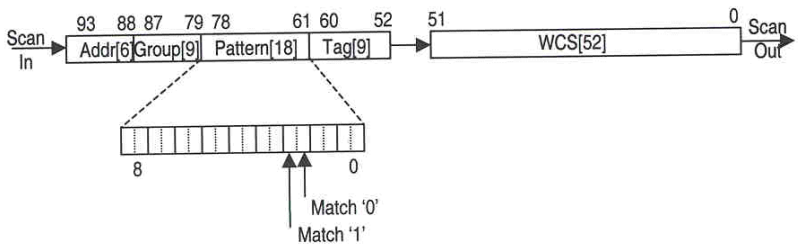


Table 33 RxByte Scan Chain Fields

Field Name	Description
Addr[6]	**CAM Address** - Six bits. Used during Writes only to select a CAM location for initialization. For the FP Rx the CAM is 24 entries long (i.e. Addr 0-23 valid)
Group[9]	**Group** – A nine bit index created by the RxByte Program Counter used during reads to qualify the pattern match.
Pattern[18]	**Match Pattern**- A nine – two bit pattern used to qualify a match on '1', '0', 'X'. The Pattern may be driven from either RxByte Processor IREG3 from the Payload bus. Each two sequential bits of this field select: 00 – Match 'X', 01 – Match '1', 10 – Match '0', 11 – Invalid entry (no match possible)
Tag[9]	**Tag** - A nine bit value associated with a match. When a Match on a pattern is made the Tag Value is available for use on the RxByte Processor B-Bus.
WCS[52]	Writable Control Store
Addr[6]	**CAM Address** - Six bits. Used during Writes only to select a CAM location for initialization. For the FP Rx the CAM is 24 entries long (i.e. Addr 0-23 valid)
Group[9]	**Group** – A nine bit index created by the RxByte Program Counter used during reads to qualify the pattern match.
Pattern[18]	**Match Pattern**- A nine – two bit pattern used to qualify a match on '1', '0', 'X'. The Pattern may be driven from either RxByte Processor IREG3 from the Payload bus. Each two sequential bits of this field select: 00 – Match 'X', 01 – Match '1', 10 – Match '0', 11 – Invalid entry (no match possible)
Tag[9]	**Tag** - A nine bit value associated with a match. When a Match on a pattern is made the Tag Value is available for use on the RxByte Processor B-Bus.

WARNING: *Writing the CAMs can invalidate WCS entries. As such, the CAMs, if used, should be written first followed by WCS Byte write operations to load the WCSs.*

WCS/CAM Scan Read: The CAM/WCS Scan Read Chain is the same 94bit chain used in the Scan Write operation described above and is not optimized for operational reading (i.e. it is intended for diagnostic manufacturing pass/fail screening. The address of the CAM during reads is selected by the CAM Group and Match values, that is, the CAM addr field is not used. The address of the WCS is selected by the Byte write counter, requiring a destructive write to the WCS prior to the SCAN CAPTURE selection via the FP WCSs register. To further complicate matters, and to avoid writing the word or WCS intended to be read, bit 2 of the Byte address selection is inverted during the read so that you first write to location 0xabcd XOR 0x0004, to read 0xabcd. After you have captured the scan and scanned out the results, you select the next WCS address by performing seven additional Byte writes, thereby incrementing the WCS Byte address to the next word, and again performing a SCAN CAPTURE. In reading the WCS in this destructive fashion, you

must be careful to either Byte write original data back to 0xabcd XOR 0x0004 or to rewrite the entire WCS between reads.

In all cases where the WCS/CAM has been written, since both RxByte 0 and 1 CAM and WCSs are written at the same time, Scan Out0 should always equal Scan Out1.

DBE Scan Read: The DBE Scan Chain is a separate 52bit Scan Chain but operates, using the appropriate FP Rx WCS_CAM Register bits, precisely as described for WCS/CAM Scan Read operation described above.

Fabric to C-5 NP Link-Level Flow Control

Fabric to C-5 NP link-level flow control occurs when the fabric asks the FP Tx to stop transmission for the entire link; that is, all queues. In Utopia modes, this is effected using the Utopia protocol flow control signaling. For non-Utopia modes, the fabric will make a flow control request to the FP Rx which gets passed to the FP Tx. PowerX fabrics send out-of-band link-level requests using the control pins. PRIZMA fabrics send in-band messages, embedded in the segment header.

C-5 NP to Fabric Link-Level Flow Control

The C-5 NP will apply link-level flow control to the fabric when the FP Rx runs out of payload FIFO space. The flow control request is either made using the Utopia protocols, or (in non-Utopia modes) the FP Rx asks the FP Tx to transmit a flow control request to the fabric on its behalf. Non-Utopia link-level flow control will be covered in more detail in the mode-specific sections.

The FP Rx payload FIFO can hold up to 504Bytes of payload from segments. When the number of Bytes available in the payload FIFO becomes lower than a configurable XOFF threshold, the FP Rx requests link-level flow control. As the payload FIFO then drains and the number of Bytes available reaches a separate, configurable XON threshold, the FP Rx allows transmission to continue. For details about the specific register, see "RxDS_Configuration Register (FP Rx Configuration Function)" on page 555 in Appendix A.

Latency

Flow control operations have an inherent latency due to pipelining of data feeding the payload FIFO, delays in issuing a pause request to the fabric, and latencies in a fabric's ability to pause. The XOFF threshold must be set high enough so that even with such latency, the incoming payload does not overrun the payload FIFO, which would cause unpredictable behavior. For details about the specific register, see "RxDS_Configuration Register (FP Rx Configuration Function)" on page 555 in Appendix A.

There is some latency between the time that the FIFO threshold is hit and the time a pause is requested of the fabric. For example, with the various Utopia protocols, the flow

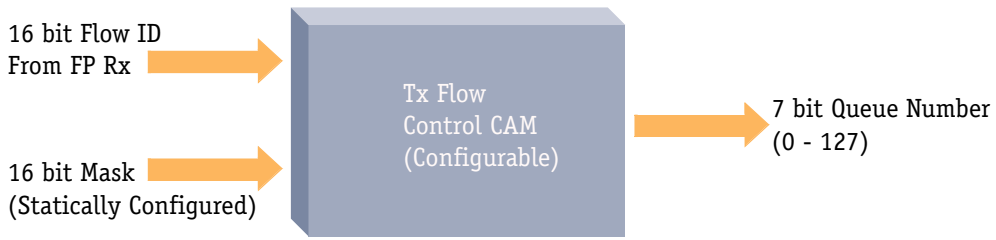
enable signal can only be deasserted at certain times during cell transmission. If the XOFF condition is reached after that time has passed, the FP Rx must be capable of receiving another full cell after the one currently arriving. The XOFF threshold must be set with this in mind. Other protocols which request link-level flow control via an FP Tx transmission have analogous latencies that must be considered when setting the threshold.

The XON threshold should always be set to a value greater than the XOFF threshold.Typically it is set to a value high enough that the XOFF condition won't immediately recur.

Fabric to C-5 NP Per-Queue Flow Control

A fabric can request that the FP Tx pause or resume transmission from a particular queue. The flow control request is handled by the FP Rx Byte Processor microcode, which then passes a 16bit flow ID and a pause/resume bit to the FP Tx (this could be different from a PDU ID). This flow ID is qualified with a 16bit mask (see the "TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535) and then used for a lookup in the Tx flow control CAM (content addressable memory). The output of the CAM is a 7bit queue number which corresponds to one of the 128 FP Tx queues; this is how the Tx knows which queue to pause or resume. Please see Figure 44 following.

Figure 44 Mapping Per-Queue Flow Control Requests to FP Tx Queues



If the lookup matches a CAM entry, the corresponding queue is paused or resumed. If the lookup does not match any CAM entry, no queue will be paused or resumed. The default mapping preloaded into the CAM by hardware is simply a one-to-one mapping, such that the queue number equals the match value, for the range of 0 to 127. The mask and CAM are completely configurable to any mapping scheme.

To configure the CAM, follow this procedure:

1. Set the flow mask to 0xffff in the TxFCE Register.
2. Then delete the preloaded CAM entries by performing 128 writes to the *TxFlowCAM* register; on each write set the DEL (delete) bit and the next 16bit Match value.

Remember that the preloaded match values are 0 through 127.

3 Next, write the desired entries into the CAM by writing to the *TxFlowCAM* register, with the WT bit asserted. The 16bit match value and 8bit write data will thus be placed into the CAM.

Bit7 of the write data must be zero; bits [6:0] represent the 7bit queue number.

To enable flow control, the Flow Control Enable configuration bit must be asserted.

If a queue is paused while not actively transmitting, the FP Tx will simply not begin transmission from that queue even if it is non-empty. Similarly, if a queue is resumed while it is not active, the FP Tx will transmit from that queue when it next has an opportunity.

Per-queue flow control requests are sent to the FP Tx using the FP Rx Byte Processors. To process these, the FP Rx Byte Processor must write 2Bytes to form a 16bit flow ID and then write to the pause/resume register. If one of the flow ID Bytes is not being used, that is, it is being masked off), then the Byte Processor need not write to that Byte. It is the act of writing to the pause/resume register which sends the request to the FP Tx, so that must be done after the flow ID has been set up.

Per-queue flow control requests are sent to a 4-deep FIFO for each Byte Processor. A single, 16-entry FIFO in the FP Tx is fed alternately from the two Byte Processors' FIFOs. The FP Tx drains a flow control request from its FIFO once every 6 core clock cycles. There is no back pressure mechanism on the Byte Processors' FIFOs, so microcode must be constructed such that, between the two Byte Processors, neither overflows their FIFO.

Because the FP Tx FIFO is fed alternately from the two Byte Processors' FIFOs, there is no inherent ordering between flow control requests from the two Byte Processors. This could be accomplished through microcode synchronization if required.

There is no harm in sending a pause request for a queue that is already paused or a resume request for a queue which has already been resumed. This has no effect on the FP Tx's per-queue flow control.

The C-5 NP does not apply per-queue flow control to the fabric. It relies only on link-level flow control in that direction.

To configure the CAM, use the procedure described in "Initialization" on page 187.

To enable flow control, the *Flow Control Enable configuration* bit must be asserted.

C-5 NP to Fabric Per-Queue Flow Control

The C-5 does not apply per-queue flow control to the fabric. It relies only on link-level flow control in that direction.

Descriptor Sizes

The FP Tx supports descriptor sizes of 12, 16, 24, and 32Bytes. The descriptor size is configured via the *Desc Size* field of the *TxFCE Configuration* register on the FP Tx and the *Desc Size* field of the *RxFCE Configuration0* register on the FP Rx. The encodings for these two fields are different; the important point is that the descriptors sizes must be the same for the Rx and Tx. For details about the specific registers, see "TxFCE_Configuration Register (FP Tx Configuration Function)" on page 535 and "RxFCE_Configuration0 Register (FP Rx Configuration Function)" on page 559 in Appendix A.

CRC

A 32bit CRC can optionally be generated and included in each segment. The region of the segment for which CRC is calculated can be specified in the TxFI CRC and RxFI CRC registers. The start of the region (First Index) is configurable but must be a multiple of 4. The end of the region must extend to end of the segment; the CRC value resides in the last 4Bytes of the segment. Refer to the TxFI CRC and RxFI CRC descriptions for details regarding the Last Index field (the values in these 2 fields will not be exactly equal). With CRC checking enabled, the CRC of each segment is checked for all segments that are received, and failing segments are dropped.

The inclusion of CRC in each segment decreases the amount of payload included in each segment. CRC is configurable via the TxFI CRC register in the FP Tx and the RxFI CRC register in the FP Rx.

The CRC cannot be used in conjunction with variable length cells in the PowerX mode.

Endianness (Byte and Bit Ordering)

The FP can handle either Big Endian or Little Endian Byte ordering. This is configurable via the *TxFI Configuration* and *RxFI Configuration* registers. See "Big Endian Byte Ordering on Data Pins 31:0" on page 196 and "Little Endian Byte Ordering on Data Pins 31:0" on page 196, below. The Byte Processors in both the FP Tx and FP Rx will process the most significant Byte first.

Bit ordering is always fixed at [7:0] within a Byte.

- Byte ordering must be configured as Little Endian for PowerX mode
- Byte ordering must be configured as Big Endian for PRIZMA mode.
- Utopia mode applications can be designed to be either Big or Little Endian.

Big Endian Byte Ordering on Data Pins 31:0

Most Significant Byte			Least Significant Byte
31:24	23:16	15:8	7:0

Little Endian Byte Ordering on Data Pins 31:0

Least Significant Byte			Most Significant Byte
31:24	23:16	15:8	7:0

Debugging and Test Features

1. Debug MUX

Through use of the configurable Debug MUX, certain FP events can be logged in event counters located in the XP. For instance, the number of PDUs which are transmitted can be logged. For details about the specific FP events which can be monitored, see "RxFP_Debug_Event_Mux_Control (FP Rx DeBug Function)" on page 567 in Appendix A. For information on how to use the XP event counters, refer to the XP Chapter.

2. FP Rx Statistics Registers

In addition to the Debug MUX mechanism mentioned above, the FP Rx contains its own statistics registers for logging the number of received segments, PDUs, errors, etc. These registers are automatically updated by hardware, and can be initialized to a value of 0 at any time by simply performing a global write to a particular statistics register. The statistics can be read with global reads. They are described in further detail in the FP Rx section earlier in this chapter.

3. Internal Debug State Registers

Some internal FP state points are made visible through two 32bit debug state registers, one in the FP Tx and one in the FP Rx. These registers can be accessed with global reads, and contain the current status of internal state machines and other key state points. For descriptions of the register fields, refer to "TxDebug_Internal_State Register (FP Tx DeBug Function)" on page 545 and "RxDebug_Internal_State Register (FP Rx Statistics Function)" on page 573 in Appendix A.

4. Accessing FP Memories

For debugging and testing purposes, it is possible to write to and read from several of the internal memories in the FP. While accessing these memories, the FP should be disabled .

The WCSs and most CAMs are loaded by software during initialization, and can also be accessed for debug purposes. Refer to the Initialization section for more information on accessing those.

The other memories, listed below, are not programmable, but can be read and written for debug purposes while the FP is disabled.

Writing and Reading the Rx PDU ID CAM

The internal CAM that the FP Rx uses to keep track of active PDU IDs can be read and written using the "RxPDU_ID_CAM Register (FP Rx DeBug Function)" on page 570 at address 0xbde04698. To write to the CAM, set up the 16-match value and the corresponding 8bit data value, and assert the Write bit. To delete an entry, write the match value to the register and assert the Delete bit. To read an entry, set up the match value, set the Search bit, and then read out the 8bit result by reading the register.

Writing and Reading the Rx Flow Table and Descriptor Memory

The internal memory that the FP Rx uses to save the current state of active PDUs as well as the memory in which the descriptor build engine stores descriptors can be read and written. These two spaces are accessed via the "RxMemory_Data Register (FP Rx DeBug Function)" on page 570 and "RxMemory_Address Register (FP Rx DeBug Function)" on page 570in Appendix A, using the addresses listed below. To write a location in one of these memories, set up the memory address register and then write the data to the memory data register. The act of writing to the data register triggers the hardware to perform the memory write. To read a location, simply set up the address register, and then collect the result by reading the data register.

Bit 13 of the address selects between the Descriptor Memory (0) and RxFlowTable (1).

- The Descriptor Memory is organized as 1280 entries of 32bits. bits 12:2 in the address register select the entry. Address bits 1:0 are irrelevant because accesses are performed in 32bit quantities.
- The Rx Flow Table is organized as 160 entries of 72 bits. Bits 11:4 select the entry. bits 3:2 select between fields within the 72bit word.

Descriptor Memory/RxFlowTable Debug Address Map	
0 to (8K-1)	Descriptor Memory
8K to (12K-1)	Rx Flow Table

Table 34 Rx Flow Table Fields

Address[3:2]	Data[31:0]
00	btag[15:0], offset[15:0]
01	3'b0, pool[4:0], buffer[7:0], length[15:0]
10	21'b0, Drop, Queue Valid, Queue[8:0]
11	Reserved

Writing and Reading the Tx Flow Table

The internal memory that the FP Tx uses to save the current state of active PDUs can be read and written. Refer to the descriptions of "TxFlowTbl Register (FP Tx DeBug Function)" on page 540, (*TxFlowTbl_Data_Low* and *TxFlowTbl_Data_High)* in Appendix A. To read an entry, simply write the 7bit index (Tx queue number) to TxFlowTbl, and then read the resulting 60bit data from the high (upper 28 bits) and low (lower 32 bits) data registers. To write an entry, set up the data registers first, then write the 7bit address to TxFlowTbl while asserting the WT bit.

Writing and Reading Merge Space

Merge Space is an internal memory into which the FP Tx copies descriptors, to make their content available to the Byte Processors for header construction. This space can be read and written via the "TxMergeAddr (FPTx Debug Function)" on page 542 and "TxMergeData (FPTx Debug Function)" on page 543 registers in Appendix A.

Writing and Reading DMEMs

The FP Tx and Rx each have a 12K data memory which can be accessed with global reads and writes, using the addresses from the "TxByte Processor Memory Map" on page 158 and "RxByte Processor Memory Map" on page 168.

- The FP Tx uses 2KB for storing payload for its 8 active flows, 8KB for up to 128 descriptors, and the remainder for BTags to be deallocated.
- The FP Rx uses 10KB for storing payload in 64B buffers (one for each of 159 concurrent flows), and 2KB for storing BTags.

Reading TLU Responses

The TLU response area in the FP Rx (256Bytes) can be globally read for debug purposes only, at the locations specified in the "RxByte Processor Memory Map" on page 168.

Fabric Interface Configuration and Operation

FP Payload Bus Bandwidth

Depending on the bandwidth needed by an FP application, the payload bus should be configured to allocate a higher proportion of its bandwidth to the FP. This is accomplished via a bit called Payload Bus Arbiter FP More Slots bit in the XP Miscellaneous Control Register in the XP. For details about the specific registers, begin with Table 135 on page 447 and "Serial Bus Configuration Register (XP Miscellaneous Control Function)" on page 467.

Network Processor-to-Network Processor Operation (Back to back)

Two C-5s can be directly connected via the fabric port. In this configuration, the Network Processors should be configured in Utopia 3 mode, with the FP Tx's configured as an ATM device and the FP Rx's configured to be PHY devices.

FP Interface Modes

The FP interface can be configured to operate in a number of different modes:

- Utopia 3
- Utopia 2
- PRIZMA
- PowerX

Utopia Modes

The FP Tx can be configured to operate using either the Utopia 1/2 or Utopia 3 protocol. For either protocol, it can be programmed to operate as either the ATM (master) device or the PHY (slave) device. As an ATM device, the interface can be operated with an 8-, 16-, or 32bit bus width. As a PHY device, it can only be operated with a 16- or 32bit bus width.

For Utopia 2 or 3, set the Interface field appropriately in the *RxFI* register. The FP Tx naturally defaults to Utopia 3 mode unless one of the Utopia 2, PRIZMA or PowerX mode bits is asserted in the TxFI register. For Utopia 2, set the U2 Mode and U2 Tristate Enable bits in the *TxFI* register. For details about the specific registers, see "TxFI_Configuration Register (FP Tx Configuration Function)" on page 530 in Appendix A.

See Table 35 and Table 36 on page 200, Utopia1, 2, and 3 ATM Mode and PHY mode, C-5 Network Processor to Fabric Interface Pin Mapping.

Table 35 Utopia1, 2, 3 ATM Mode, C-5 NP to Fabric Interface Pin Mapping

Rx Signals				Tx Signals			
C-5 NP	**I/O**	**Utopia**	**Note**	**C-5 NP**	**I/O**	**Utopia**	**Note**
FRXCTL0	Output	RxEnb[1]	Pullup or nc [2]	FTXCTL0	Output	TxEnb*	Pullup or nc
FRXCTL1	Input	RxClav	Pulldown	FTXCTL1	Input	TxClav	Pulldown
FRXCTL2	Input	RxSOC	Pulldown	FTXCTL2	Output	TxSOC	Pulldown
FRXCTL3	Input	n/a	Pullup or Pulldown	FTXCTL3	Input	n/a	Pullup or Pulldown
FRXCTL4	Input	n/a	Pullup or Pulldown	FTXCTL4	Input	n/a	Pullup or Pulldown
FRXCTL5	Input	n/a	Pullup or Pulldown	FTXCTL5	Input	n/a	Pullup or Pulldown
FRXCTL6	Input	RxPrty		FTXCTL6	Output	TxPrty	

1 Active Low
2 No Connection

Table 36 Utopia1, 2, 3 PHY Mode, C-5 NP to Fabric Interface Pin Mapping

Rx Signals				Tx Signals			
C-5 NP	**I/O**	**Utopia**	**Note**	**C-5 NP**	**I/O**	**Utopia**	**Note**
FRXCTL0	Input	TxEnb[1]	Pullup	FTXCTL0	Input	RxEnb*	Pullup
FRXCTL1	Output	TxClav	Pulldown or nc [2]	FTXCTL1	Output	RxClav	Pulldown or nc
FRXCTL2	Input	TxSOC	Pulldown	FTXCTL2	Output	RxSOC	Pulldown
FRXCTL3	Input	n/a	Pullup or Pulldown	FTXCTL3	Input	n/a	Pullup or Pulldown
FRXCTL4	Input	n/a	Pullup or Pulldown	FTXCTL4	Input	n/a	Pullup or Pulldown
FRXCTL5	Input	n/a	Pullup or Pulldown	FTXCTL5	Input	n/a	Pullup or Pulldown
FRXCTL6	Input	TxPrty		FTXCTL6	Output	RxPrty	

1 Active Low
2 No Connection

C-5 NP Utopia Operation

Because the Utopia specifications are ambiguous and subject to interpretation, this document provides C-Port's interpretation of the specifications as well as the notes about

the C-5 NP's implementation. Where aspects of the Utopia specifications are optional, whether C-5 NP supports them or not is described.

This document attempts to clarify the Utopia standards by reducing them to a few assertions, with references to the Utopia specifications from the ATM Forum Technical committee. The following versions of the document were used:

- Utopia Specification Level 1, Version 2.01, af-phy-0017.000, March 21, 1994
- Utopia Level 2, Version 1.0, af-phy-0039.000, June 1995
- Utopia 3 Physical Layer Interface, af-phy-0136.000, November, 1999

Table 37 C-Port Supported Utopia Modes

Bus Width (bits)	U1 PHY	U1 ATM	U2 PHY	U2 ATM	U3 PHY	U3 ATM
32	n/a	n/a	n/a	n/a	Yes	Yes
16	n/a	n/a	Yes	Yes	Yes	Yes
8	No	Yes	No	Yes	No	Yes

Utopia 3

General Compliance

- Must support single PHY operation; multi-PHY support is optional - C-5 NP supports single PHY only
- Must support at least one of 8-, 16-, or 32bit interface - C-5 NP supports 8, 16, and 32
- Must support 52Byte cells
- Parity pin/support is optional - C-5 NP supports parity
- The C-5 NP only supports full transfer (cell-level handshaking). Once cell transmission starts, the cell is transferred, uninterrupted. (Section 2.2.5)
- `RxClk and TxClk` are never driven by the C-5 NP. They are inputs

Control Signals

The "cell transfer" period referred to in the descriptions below refers to the consecutive bus cycles, starting with the cycle in which Start of Cell (SOC) is asserted and lasting the number of bus cycles required to transfer the fixed cell size. The first cycle of a cell transfer is the cycle in which SOC is asserted. All cycles during the cell transfer are valid data cycles.

"Asserted" and "deasserted" are used as logical terms, and correspond to different logic values depending on the active state of the signal. For example, RxEnb and TxEnb are active low, therefore asserted is a logic value of 0 and deasserted is a logic value of 1.

TxClav Specification

- Asserted by PHY to indicate readiness to accept a cell
- Can change from deasserted to asserted at any time
- Can only change from asserted to deasserted two cycles after the cycle in which TxSOC is asserted (Section 3.2.1)
- Once asserted (indicating readiness to accept a cell), it must stay asserted until associated cell transfer begins (Section 3.2.1)

TxEnb Specification

- Asserted during cell transfer
- Can only change from deasserted to asserted if TxClav was asserted two cycles previously (Section 3.1.1)
- Can only change from deasserted to asserted with the assertion of TxSOC
- Can only change from asserted to deasserted at end of cell transfer
- Must be deasserted at end of cell if another cell is not starting immediately after the current one.

In Utopia 3 mode, C-5 NP ignores the TxEnb signal and considers there to be valid cell data on the data lines for consecutive cycles starting with an SOC cycle, until the fixed cell size number of Bytes have been received.

TxSOC Specification

- Asserted for one cycle to indicate first cycle of cell
- Can only be asserted for a single cycle
- Can only be asserted when TxEnb is asserted (Section 3.1.1)
- Can only be asserted when TxClav was asserted for the two previous cycles
- Cannot be asserted in the middle of cell transfer

RxClav Specification

- Asserted by PHY whenever it has a cell available to transfer

- Can only change from asserted to deasserted at the same time RxSOC changes from deasserted to asserted (Section 3.2.2); that is, If the PHY does not have a subsequent cell to transmit, it must indicate so at the beginning of the current cell
- Once asserted, it must stay asserted until the cycle after the next RxSOC assertion (Section 3.2.2)
- Can change from deasserted to asserted at any time

RxEnb Specification

- Asserted "in response" (Section 3.2.2) to RxClav assertion to "initiate" (Section 3.1.2) a cell transfer
- Can only change from deasserted to asserted when RxClav was asserted 2 cycles before
- Must remain asserted during the cell transfer, at least until two cycles before end of cell (Section 3.2)
- Must be deasserted two cycles before end of cell if:
  - Another cell cannot be received or
  - RxClav has been deasserted during the cell transfer
- Once asserted (indicating readiness to accept a cell), it must stay asserted until an associated cell transfer begins

RxSOC Specification

- Asserted for one cycle to indicate first cycle of cell
- Can only be asserted for a single cycle
- Can only be asserted when RxEnb was asserted for the two previous cycles
- Cannot be asserted in the middle of cell transfer

Utopia 2

General

Handshaking response time is expected to be one cycle, not two like Utopia 3. For example, if RxEnb is deasserted during the middle of a cell transfer for one cycle, the Utopia PHY is expected to insert one invalid data cycle on the very next cycle.

Clocks are expected to be provided by (outputs from) the ATM device. The C-5 NP does not drive any clocks. It requires them to be inputs.

The C-5 NP only supports cell-level handshaking.

C-Port recommends that a Utopia 1 or 2 PHY device tristate the RxSOC, RxData (and we presume RxPrty) lines during cycles following cycles where RxEnb is not asserted. C-5 NP implements this recommendation.

Control signals

The "cell transfer" period referred to in the descriptions below refers to the consecutive bus cycles, starting with the cycle in which SOC is asserted and ending with the valid bus cycle which transfers the last Byte of the fixed cell size. During the cell transfer there can be any number of invalid data cycles as indicated by deassertion of the Enb signal. The first cycle of a cell transfer is the cycle in which SOC is asserted.

"Asserted" and "deasserted" are used as logical terms, and correspond to different logic values depending on the active state of the signal. For example, RxEnb and TxEnb are active low, therefore asserted is a logic value of 0 and deasserted is a logic value of 1.

TxClav Specification

- Asserted by PHY to indicate readiness to accept a cell
- Can change from deasserted to asserted at any time.
- Can change from asserted to deasserted any time.
- Once asserted (indicating readiness to accept a cell), it must stay asserted until an associated cell transfer begins.
- Must deassert 4 cycles before the end of a cell transfer to avoid transfer of a subsequent cell. If asserted 4 cycles before the end of a cell transfer, it must stay asserted at least until a subsequent cell transfer begins.

As recommended by the Utopia specification, C-5 NP will keep TxClav asserted through the cell transfer until at least the fourth cycle before the end of the cell.

TxEnb Specification

- Asserted during valid cycles of a cell transfer
- Must be asserted with TxSOC
- Can be deasserted during a cell transfer to indicate invalid data cycles. When TxEnb is deasserted, data on TxData is invalid.
- Cannot be asserted when a cell transfer is not in progress

The C-5 NP will not deassert TxEnb during a cell transfer.

TxSOC Specification

- Asserted for one cycle to indicate first cycle of cell
- Can only be asserted for a single cycle
- Can only be asserted when TxEnb is asserted
- Can only be asserted when TxClav was asserted in the previous cycles

TxSOC cannot be asserted in the middle of cell transfer

RxClav Specification

- Asserted by PHY whenever it has a cell available to transfer
- Can change from deasserted to asserted at any time
- Once asserted, it must stay asserted until the cycle after the next cell transfer begins
- Must remain asserted throughout a cell transfer
- Must be asserted to coincide with the cycle following the last cycle of a cell transfer to allow back-to-back cell transfer.

RxEnb Specification

- Asserted in response to RxClav assertion to a cell transfer
- Must be asserted with RxSOC
- Can only change from deasserted to asserted when RxClav was asserted in the previous cycle
- Must be deasserted one cycle before end of cell if a subsequent cell cannot be received.
- Can be deasserted during a cell transfer to indicate invalid data cycles. When RxEnb is deasserted, data on RxData in the following cycle is invalid.

C-5 NP will not deassert RxEnb during a cell transfer.

RxSOC Specification

- Asserted for one cycle to indicate first cycle of cell
- Can only be asserted for a single cycle
- Can only be asserted when RxEnb was asserted in the previous cycle
- Cannot be asserted in the middle of cell transfer

C-5 NP will tristate RxSOC in cycles following cycles where RxEnb is deasserted.

PRIZMA Mode

The FP can be configured to operate with the IBM PRIZMA switching fabric. The interface to the PRIZMA fabric is through the UDASL chip. In PRIZMA fabric terminology, the segment is called a "packet". That term will be used in this section.

See Table 38 on page 208, PRIZMA Mode, C-5 NP to Fabric Interface Pin Mapping.

While the PRIZMA fabric supports operation up to 125MHz, the C-5 NP will only support operation up to 110MHz.

This sectionexplains how the C-5 NP handles the following PRIZMA mode parameters.

- Packet Sizes
- In-Band Flow Control
- Link-Level Flow Control
- Idle Packets
- Queue Grants

- Packet Sizes

  The PRIZMA fabric supports packet sizes between 64 and 80Bytes. The C-5 NP supports this range of packet sizes but packet sizes must be a multiple of 4Bytes. The PRIZMA fabric must be configured to place the packet qualifier Byte as the first Byte of the header. Typically, the destination bitmap (on ingress to the fabric) or queue grants (on egress from the fabric) would be the next Bytes of the header but there is no requirement for this. Because microcode generates the PRIZMA address bitmap and processes, the queue grant bits in the PRIZMA header, the PRIZMA fabric can be configured to have 16 or 32 queues per priority.

- In-Band Flow Control

  When operating with a PRIZMA fabric, the fabric is configured to use in-band flow control and the Utopia protocol Enb and Clav signals are not used. The Enb inputs to the C-5 NP should be pulled down and the Clav inputs to the UDASL chip should be pulled up. The SOC pins of the C-5 NP and the UDASL should be connected.

  When using in-band flow control, the FP Tx generates PRIZMA-format idle packets whenever it has no segments to transmit or whenever it has been paused by the PRIZMA fabric. Packets (data or idle) are transmitted from the C-5 NP in an absolutely back-to-back fashion on the interface; that is, there will be no unused cycles on the

bus. The C-5 NP must be configured to generate idle packets using the *TxFI Configuration* register to enable idle packet generation and the *TxDM Header/Payload Delimiter* register to specify the length of idle packets so they are the same length as data packets. The *TxIdleHdr* register must be programmed to a value of 0xCCCCCCCC so that the content of idle packets is correct.

For details about the specific registers, see TxFI_Configuration Register (FP Tx Configuration Function) and TxDM_Header/Payload Delimiter Register (FP Tx Configuration Function) in Appendix A.

When idle packets are generated, no microcode generated header is used, so PRIZMA microcode needs no support for idle packet header generation.

- Link-Level Flow Control

  If the C-5 NP requires link-level flow control, the FP Tx will assert the TxPause bits in the PRIZMA packet qualifier Byte of the packets that it transmits. The FP Tx will always assert or deassert all four of the TxPause bits together; that is, if one TxPause bit is asserted (deasserted) all TxPause bits are asserted (deasserted).

  The C-5 NP presumes that the UDASL chip is configured to use full inband flow control in such a way that, when the UDASL's input FIFO fills and it requires link-level flow control, it is expected to deassert the shared memory grant bit, for at least one priority, in the packet qualifier of packets that it transmits to the C-5 NP. This occurs while the FP Rx extracts PRIZMA shared memory grant information from the packet qualifier Byte of each packet; microcode does not have to do this.

  If a shared memory grant is lost for any priority, the C-5 NP will pause all data packet transmission and transmit only idle packets to the fabric. After the C-5 NP powers up, it does not allow the FP Tx to transmit until it has received a PRIZMA packet with a shared memory grant for each of four priorities. Because of this, the PRIZMA fabric must be operated using 4 priorities or, if fewer priorities are to be used, the fabric must first be configured for 4 priorities so that it generates idle packets with shared memory grants for each of four priorities and then, reconfigured to the desired number of priorities.

- Idle Packets

  The FP Rx expects that idle packets will be received whenever the fabric has no data packets to transmit or whenever the C-5 NP has paused the fabric. FP Rx data splitting registers must be configured to appropriately identify and split idle packets. Idle packets should be configured so at least 32Bytes are treated as payload. Microcode must recognize and discard idle packets. See “Discarding Segments” on page 176

When the PRIZMA fabric is paused, it will stop transmitting data packets but will continue to transmit idle packets into the C-5 NP. Because some portion of these packets must be directed into the payload FIFO, the payload FIFO may eventually overflow and lose some of this idle packet data. This is not a problem because this payload would be discarded when it is popped from the payload FIFO anyway. Refer to the section regarding "Rx Drop Mode" on page 177.

- Queue Grants

  Queue grants in PRIZMA headers sent into the C-5 NP can be handled by FP Rx microcode so they generate per-queue flow control messages to the FP Tx.

Configuring for PRIZMA Mode

To configure the FP to operate with a PRIZMA interface, the interface is configured essentially like it is for 32bit Utopia 3 PHY operation with a few small changes and additions. These differences are:

- The PRIZMA bit of the *FP TxFI Configuration* register must be set
- The Interface field of the *FP RxFI Configuration* register must have a value of four.
- The idle cell enable bit of the *FP TxFI Configuration* register must be set
- The *TxIdleHdr* register must contain a value of 0xCCCCCCCC
- The idle packet length field of the TxDM Header/Payload Delimiter register must be programmed with a value which is 1 less than the number of fabric interface cycles required to transmit a packet

For details about the specific registers, see "TxDM_Header/Payload Delimiter Register (FP Tx Configuration Function)" on page 532 in Appendix A.

While the C-5 NP supports the basic protocol of the PRIZMA fabric, a particular application may not be possible due to limitations of microcode space and cycle time. Consult the PRIZMA application note to understand some of the trade-offs involved in developing a PRIZMA application.

Table 38 PRIZMA Mode, C-5 Network Processor to Fabric Interface Pin Mapping

Rx Signals				Tx Signals			
C-5 NP	**I/O**	**Utopia**	**Note**	**C-5 NP**	**I/O**	**Utopia**	**Note**
FRXCTL0	Input	TxEnb[1]	pulldown (not connected to fabric) [2]	FTXCTL0	Input	RxEnb*	pulldown (not connected to fabric)

Table 38 PRIZMA Mode, C-5 Network Processor to Fabric Interface Pin Mapping (continued)

Rx Signals				Tx Signals			
C-5 NP	**I/O**	**Utopia**	**Note**	**C-5 NP**	**I/O**	**Utopia**	**Note**
FRXCTL1	Output	TxClav	nc	FTXCTL1	Output	RxClav	nc
FRXCTL2	Input	TxSOP	Pulldown	FTXCTL2	Output	RxSOP	Pulldown
FRXCTL3	Input	n/a	Pullup or Pulldown	FTXCTL3	Input	n/a	Pullup or Pulldown
FRXCTL4	Input	n/a	Pullup or Pulldown	FTXCTL4	Input	n/a	Pullup or Pulldown
FRXCTL5	Input	n/a	Pullup or Pulldown	FTXCTL5	Input	n/a	Pullup or Pulldown
FRXCTL6	Input	TxPrty	Pullup or Pulldown, unless connected to TxPrty	FTXCTL6	Output	RxPrty	nc

1 Active Low
2 No connect

PowerX Mode

The FP can be configured to operate with the PowerX TeraChannel® Switch Fabric.

See Table 39 on page 212, PowerX Mode, Fabric Interface to Pin Mapping.

The C-5 NP supports the 32bit PowerX interface, not the 16bit interface.

The C-5 NP transmits: invalid data, start of frame data, valid frame data, pause, and resume PowerX bus cycles. It never transmits abort or flow control cycles.

C-5 NP supports: the receipt of invalid data, start of frame data, valid frame data, pause, and resume and flow control bus cycles. It ignores abort or reserved bus cycles that it receives.

Constraints

The C-5 NP can be programmed to support any of the PowerX frame types, however it may not be possible to support all types in a single application given microcode and configuration constraints. The use of the service channel and urgency fields of the PowerX header is completely a function of the application microcode.

Requirements

The payload length field of the PowerX header must be generated by FP Tx microcode. FP Tx hardware makes the current segment length available to microcode for this purpose. FP Rx hardware extracts the payload length field of the PowerX header on frames that it

receives to determine when the frame has been completely delivered. There is no need for FP Rx microcode to process the payload length field.

The FP Tx can optionally generate variable-length frames which the PowerX fabric supports. When configured to do so, the FP Tx generates fixed, maximum-sized frames for all frames of a PDU except the last one. For the last frame of the PDU, the FP Tx transmits the shortest possible frame (which is a multiple of 4Bytes). For short frames, dead cycles will be inserted as necessary to meet the minimum SOF-to-SOF spacing as configured in the "TxDM_Header/Payload Delimiter Register (FP Tx Configuration Function)" on page 532. If variable-length frames are not enabled, the FP Tx will always transmit maximum-sized frames for all frames of a PDU. The use of variable-length frames can be enabled via the *TxFI Configuration* register.

The use of variable-length frames is incompatible with the use of CRC in those frames. If CRC is to be used, fixed frames must be used.

The PowerX fabric must be guaranteed a minimum number of clock cycles between consecutive SOF cycles. This is done by setting the Min. Cell Size field mentioned above, which guarantees a minimum SOF-to-SOF gap by having a minimum size variable frame length.

The PowerX fabric must be configured to provide a minimum SOF-to-SOF gap to guarantee that FP Rx microcode can keep up with an extended stream of minimally-spaced frames (with minimally-spaced flow control cycles interspersed). The size of this gap is dependent on the performance of the FP Rx microcode. Refer to the formula in "Performance Requirement" on page 172.

As PowerX flow control bus cycles are received by the C-5 NP, the FP Rx directs the flow control messages to a control FIFO in one of the Byte Processors. The control FIFOs parallel the header FIFOs in the two Byte Processors and control messages are delivered alternately to each of the Byte Processors. The C-5 NP should be configured to direct only the 2 meaningful Bytes of PowerX flow control bus cycles to a control FIFO. This is done via the *Control Word Size* field in the *RxDS Configuration* Register. For details about the specific registers, see "RxDS_Configuration Register (FP Rx Configuration Function)" on page 555 in Appendix A.

Byte Processor Unloading

In PowerX mode, when a Byte Processor unloads the header FIFO, it is really unloading the control or header FIFO. Control FIFO contents take precedence over header FIFO contents so that if a Byte Processor does a FIFO unload it gets a control FIFO Byte if there are any present. The Byte Processor can test whether the Byte it has unloaded came from the

control FIFO instead of the header FIFO using the "control word" test condition. If it is true, the Byte came from the control FIFO and is part of a flow control message.

For PowerX flow control messages (16bits), just as with headers, the Byte Processors process the most significant Byte first. Since PowerX uses Little Endian Byte ordering, the Byte Processor will first see the Byte which was received on pins 7:0, and next see the Byte received on pins 15:8.

Upon receiving a flow control message, an FP Rx Byte Processor can pause or resume the corresponding FP Tx queue by writing a flow control message to the FP Tx.

To configure the FP to operate with a PowerX interface the following setting must be done:

- Set the PowerX bit of the TxFI register ("TxFI_Configuration Register (FP Tx Configuration Function)" on page 530)
- The *Interface* field of the RxFI register must have a value of 3 ("RxFI_Configuration Register (FP Rx Configuration Function)" on page 551)
- *Minimum SOF*
- Set the *Control Word Disable* bit of the *RxDS Configuration Register* to 0 ("RxDS_Configuration Register (FP Rx Configuration Function)" on page 555)
- Set the Control Word Size field of the RxDS Configuration Register to 2 ("RxDS_Configuration Register (FP Rx Configuration Function)" on page 555)
- If variable size frames are desired, the *variable size frame* enable bit should be set in the *TxFI Configuration* register, and CRC turned off ("TxFI_Configuration Register (FP Tx Configuration Function)" on page 530)
- Set the *Byte parity* bit of the *FP Rx interface* configuration register to 1 ("RxFI_Configuration Register (FP Rx Configuration Function)" on page 551
- Set the *Parity Mask* bit of the RxFI register to 0000111 so that control pins 2:0 are included in parity calculations
- Enable little endianness in the *RxFI* and *TxFI* registers (set Big Endian bit = 0 in both registers)
- Set the *Registered Input* bit to 1 in the *RxFI* register. (The same bit in the *TxFI* register is irrelevant for PowerX mode since there are no inputs to the FP Tx.)

While the C-5 NP supports the basic protocol of the PowerX fabric, a particular application may not be possible due to limitations of microcode space and time. You

should consult the PowerX Application Note to understand some of the trade-offs involved in developing a PowerX application.

Table 39 Power X Mode, Fabric Interface to Pin Mapping

Rx Signals				Tx Signals			
C-5 NP	**I/O**	**Power X**	**Note**	**C-5 NP**	**I/O**	**Power X**	**Note**
FRXCTL0	Input	RxCtrl[0]	Pulldown	FTXCTL0	Output	TxCtrl[0]	Pulldown
FRXCTL1	Input	RxCtrl[1]	Pulldown	FTXCTL1	Output	TxCtrl[1]	Pulldown
FRXCTL2	Input	RxCtrl[2]	Pulldown	FTXCTL2	Output	TxCtrl[2]	Pulldown
FRXCTL3	Input	RxPrty[3]		FTXCTL3	Output	TxPrty[3]	
FRXCTL4	Input	RxPrty[2]		FTXCTL4	Output	TxPrty[2]	
FRXCTL5	Input	RxPrty[1]		FTXCTL5	Output	TxPrty[1]	
FRXCTL6	Input	RxPrty[0]		FTXCTL6	Output	TxPrty[0]	

Chapter 5

Buffer Management Unit

Chapter Overview

This chapter covers the following topics:

Buffer Management Unit (BMU) Overview

The Buffer Management Unit (BMU) provides the interface to external SDRAM for the C-5 NP. The BMU partitions the SDRAM into buffers accessible to the Channel Processors (CPs), the Executive Processor (XP), and the Fabric Processor (FP) for payload storage. The BMU also provides services for managing the buffer handles called Buffer Tags (BTags) and services for maintaining BTag reference count tables called the Multi-Use Counters (MUC), which are used for forwarding payload to multiple targets.

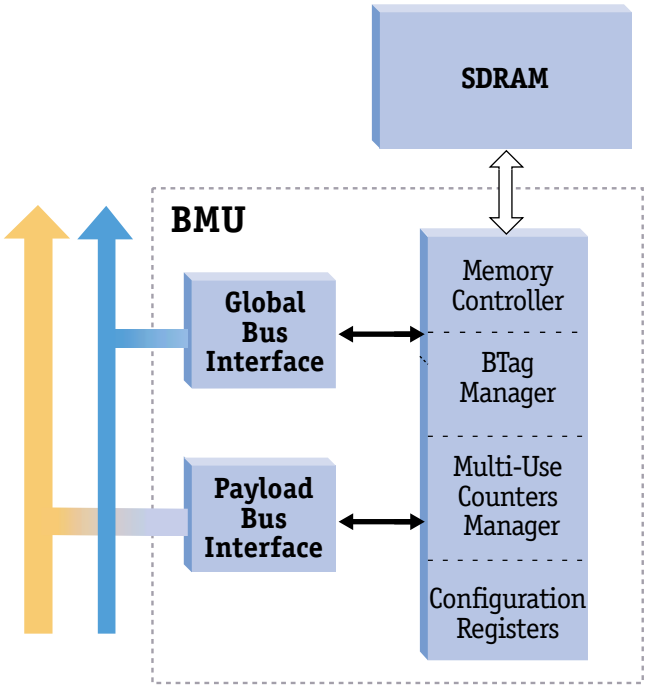
BMU Major Components

The major components of the BMU are listed inTable 40 on page 214. In addition, Figure 45 on page 215 shows the BMU Block Diagram.

Table 40 Major Components of the BMU and Their Functions

Item	Function
Memory Controller	Processes all requests for SDRAM transactions, primarily buffer memory requests for Payload storage. Buffer access is made from CP or XP application software, or the FP hardware using a Payload transaction Control Block (WrCB0, RdCB0,RxCB0,TxCB0).
BTag Manager	Handles BTag allocation and deallocation. BTag operations are made from CP or XP application software, or the FP hardware using a Payload transaction Control Block (WrCB0, RdCB0,RxCB0,TxCB0).
Multi-Use Counter Manager	Handles Multi-Use Counter (MUC) allocation, decrement and automatic BTag deallocation. MUC operations are made from CP or XP application software using a Payload transaction Control Block (WrCB0, RdCB0, RxCB0, TxCB0).
Configuration Registers	Used for setting up physical and buffer memory configuration, and for debug and test. Configuration operations are made from CP or XP application software using loads/stores from/to Global memory space.

Figure 45 BMU Block Diagram



BMU Physical Memory Organization

The SDRAM memory array is organized as 128bit words operating in accordance with Joint Electronic Device Engineering Council (JEDEC) specifications at 100MHz and 125MHz (depending on the SDRAM used). This provides a maximum bandwidth of 12.8Gbps or 16Gbps respectively. The BMU supports four-beat bursts of 16 Bytes each, optimized for 64-Mbyte parts, and for similar parts with four (4) internal banks. The C-5 NP supports one (1) physical bank of SDRAM, but a number of parts and arrays are supported. Registered DIMMs can be supported by adjusting timing parameters.

In addition to the 128bit words of user data, the BMU be configured to handle an additional eleven (11) bits of data, two (2) Out-of-Band (OOB) bits and nine (9) ECC (Error Correction Code) bits when ECC is enabled.

When the Out-of-Band (OOB) (2bits) and ECC (9bits) are used the total bits stored is increased from 128bit words to 139bit words. Therefore, the number of parts increase to accommodate the additional 11bits. Refer to Table 41 on page 216.

Table 41 Supported SDRAM Configurations

Parts	Number of Parts	Capacity of SDRAM Card	Address Bits	Open Page Size*	Bandwidth (Min.)	Bandwidth (Max.)
64Mbx8	18	128MB	27	8KB	1GB@125MHz	2GB@125MHz
64Mbx16	9	64MB	26	4KB		
64Mbx32	5	32MB	25	2KB		
128Mbx8	18	256MB	28	8KB		
128Mbx16	9	128MB	27	4KB		
128Mbx32	5	64MB	26	2KB		
256Mbx16	9	256MB	28	4KB		
256Mbx32	5	128MB	27	2KB		

* Open page size is determined by the formula (512KB * Number of Parts). Each part has 512KB of open page.

All transactions with the SDRAMs are 4 beat bursts=64Bytes of data. Writes of quantities < 16Bytes are not supported due to the addition of SECDED (Single Error Correcting, Double Error Detecting) ECC (Error Correction Code) support. Such writes would require read-modify-write transactions using up twice the write bandwidth.

Out-of-Band Bits

The Out-of-Band (OOB) bits hold control information that travels with payload data. The bits are organized as 8bits per 64Bytes of data and stored as 2bits for each 16Bytes. Therefore, to move all Out-of-Band (OOB) bits [7:0] with 64Bytes of user data the sequence is: 2bits are stored with the first 16Bytes of data, then the next 2bits are stored with the second 16Bytes of data, then the next 2bits are stored with the third 16Bytes of data, then the next 2bits are stored with the fourth and final 16Bytes of user data. Refer toTable 19 on page 101.

SECDED ECC Support

Data stored in SDRAM can be protected by a *Single Error Correcting, Double Error Detecting* (SECDED*) Error Correction Code* (ECC) if ECC is enabled and extra memory is included in the system. Nine (9) ECC bits can correct all single bit errors and detect all double bit errors across 130bits (128bits of data and 2bits for OOB) of data read/written per SDRAM clock cycle. For each 130bit write, nine (9) ECC *check bits* are generated and stored along with the user data (typically 128bits). When the data is read back from SDRAM, the nine (9) check bits are re-generated and checked against the check bits that were stored. If they are the same, then there is no error. If there is an odd number of bits that differ, then there is a single bit error. If there is an even number of bits that differ then there is a double bit error. This is implemented using the *ECC Enable* single bit register and the *Single ECC Error* register, that counts the number of (ECC) errors that have occurred. Refer to Table 52 on page 240.

BMU Buffer Memory Organization

The Buffer Memory is organized as described in the following sections.

Buffer Pools

The BMU divides SDRAM into sections called *buffer pools*. Pools are intended to provide protection among the many users of buffer memory, and to allow the applications (via chip configuration) to carve memory into different size buffers. Up to 30 buffer pools can be configured. Each Pool Area= (Buffer Size * Number of Buffers). The *Pool0 Base* to *Pool29 Base* registers are used to define the base address in SDRAM for a pool (Buffer Memory). Refer to Figure 46 on page 220.

Configuration software must ensure that pools do not overlap and that there is enough physical memory to hold all the pools.

Buffers

Each buffer pool contains up to 65,536 fixed size buffers. The number of buffers and size of the buffers can be different for each buffer pool. The number of buffers in a pool must be a multiple of eight (8) and the size of each buffer must be a power of two (2) between 64kBytes and 64Bytes, excluding 128Byte buffers. The Buffer Size is user selectable using the *Pool0 BTag Shift* to *Pool29 BTag Shift* registers. Refer to Table 42 on page 219 and Figure 46 on page 220.

Pools are generally configured during system initialization. Unpredictable behavior results when a pool is accessed prior to initialization. Refer to "BMU Setup" on page 243.

Buffer Tags (BTags)

Each buffer in a pool has a handle defined that identifies its location in the pool. These handles are called *Buffer Tags* (BTags). There is a one to one relationship between Buffers and BTags (1Buffer to 1BTag). Each BTag is 2Bytes. The BTags themselves are stored in SDRAM and inside the BMU. To allocate (assign) a buffer to a CP or XP, software must issue a BTag read (RdCB) request. Buffers are allocated in multiples of eight (8). The *Num BTag0* to *Num BTag29* registers are used to set the number of BTags in a Pool. The *BTag FIFO Base0* to *BTag Base29* registers are used to define the base address in SDRAM for the Pool BTag FIFO. Each Pool BTag FIFO Area= (2Bytes * Number of Buffers).

Storage Space (SDRAM Partitions)

The SDRAM space is partitioned using the variables shown in Table 42 on page 219. In addition, refer to Figure 46 on page 220.

Table 42 Legal Ranges for SDRAM Partition Variables

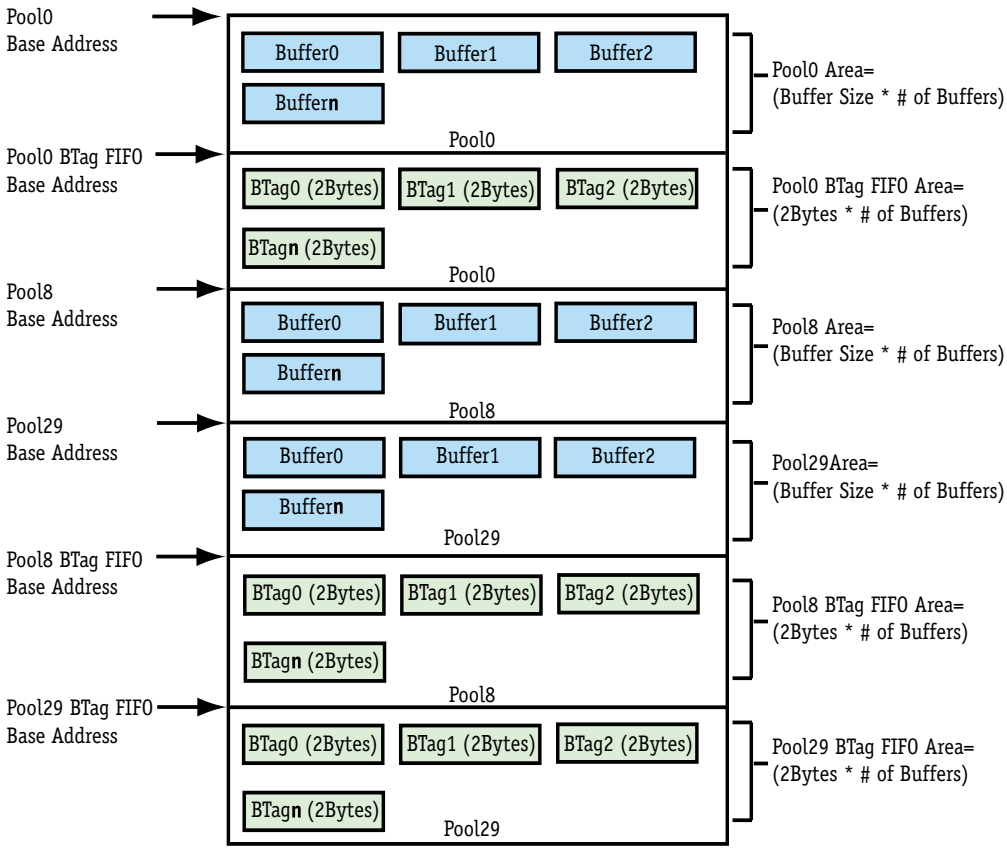
Item	Range
Number of Pools	0 to 29
Number of Buffers per Pool	0 to 65,535 (must be in multiples of 8)
Individual Buffer Size	<table><tr><th>Size</th><th>Encoded Value</th></tr><tr><td>64kB</td><td>0</td></tr><tr><td>32kB</td><td>1</td></tr><tr><td>16kB</td><td>2</td></tr><tr><td>8kb</td><td>3</td></tr><tr><td>4kB</td><td>4</td></tr><tr><td>2kB</td><td>5</td></tr><tr><td>1kB</td><td>6</td></tr><tr><td>512B</td><td>7</td></tr><tr><td>256B</td><td>8</td></tr><tr><td>128B</td><td>(Not Supported)</td></tr><tr><td>64B</td><td>10</td></tr></table>
Number of BTags per Pool	0 to 65,535 (must be in multiples of 8)

Buffer Access

All transactions with the SDRAM are 64Bytes in length. Access to buffers<64Bytes in length still requires 64Byte transactions. Operations of <16Bytes of data are not supported in the BMU. All buffer accesses must be aligned to 16Byte boundaries. The minimum size of an internal data transfer is 64Bytes, taking four (4) 16Byte slots on the Payload Bus. Buffer transfers of <64Bytes result in empty slots on the Payload Bus. Buffer writes of<64Bytes use data masking to suppress the undesired writes to SDRAM.

The BMU is optimized for 64Byte aligned access to buffers. Unaligned transfers are possible, but require special handling. Refer to “Unaligned Buffers” on page 224.

Figure 46 SDRAM Storage Space for User Data Example



Types of Transactions

The BMU supports seven (7) functions divided into three (3) categories. The different functions are initiated by CPs, XP or the FP using the Multi-Use Control Blocks by just changing the fields. Multi-Use Control Blocks use the following registers: WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB0_SDP_Addr; RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr; RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB0_SDP_Addr; and TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr. Refer to Table 43 on page 221, Table 44 on page 222, and Table 45 on page 223.

Table 43 Multi-Use Control Blocks (for Wr, Rx, Rd and Tx)

Mode	Category	Function	Fields Used	Details
• CP to/from BMU • XP to/from BMU • FP to/from BMU	Memory Transactions	Buffer Memory Transfer Operation	PoolID, BTag, Offset	See "Using Wr/Rd Control Blocks for Payload Transactions" on page 224 and "Using Rx/Tx Control Blocks for Payload Transactions" on page 224.
	BTag Management Transactions	Initializing BTags	PoolID, BTag, Command, Pool	See"BTag Initialization Operation" on page 226.
		Allocating BTags		See "BTag Allocation Operation" on page 229.
		Deallocating BTags		See "BTag Deallocation Operation" on page 231.
	Multi-Use Counter Management Transaction	Allocating Multi-Use Counters		See "MUC Allocation Operation" on page 234.
		Decrementing Multi-Use Counters		See "MUC Decrement Operation" on page 236.
		Reading Multi-Use Counters		See "MUC Read Operation" on page 238.

Table 44 WrCB0_ Variables per Field for BMU

<table>
<tr><th>Register</th><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>WrCB0_DMA_ Addr</td><td>PoolID</td><td>20:16</td><td>PoolID —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>0 to 29</td></tr>
<tr><td>BTag</td><td>30</td></tr>
<tr><td>Multi-Use Counter</td><td>30</td></tr>
</table></td></tr>
<tr><td>WrCB0_Sys_ Addr</td><td>BTag</td><td>31:16</td><td>Buffer Tag —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>Enter the BTag associated with the Buffer. Legal Range= 0 to 65535.</td></tr>
<tr><td>BTag</td><td>0</td></tr>
<tr><td>Multi-Use Counter</td><td>Enter the BTag associated with the counter. Legal Range= 0 to 65535.</td></tr>
</table></td></tr>
<tr><td></td><td>Offset</td><td>15:4</td><td>Offset —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>Enter the Offset within a Buffer.</td></tr>
</table></td></tr>
<tr><td></td><td>CMD</td><td>15:9</td><td>Command —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>BTag Initialization</td><td>0</td></tr>
<tr><td>BTag Deallocate</td><td>1</td></tr>
<tr><td>Multi-Use Counter Allocation</td><td>2</td></tr>
<tr><td>Multi-Use Counter Decrement</td><td>3</td></tr>
</table></td></tr>
<tr><td></td><td>Pool</td><td>8:4</td><td>Buffer Pool —
<table>
<tr><th>Operation Type</th><th>Function</th><th>Value</th></tr>
<tr><td>BTag Initialization</td><td>Enter the Pool to write to.</td><td rowspan="4">0 to 29</td></tr>
<tr><td>BTag Deallocation</td><td>Enter the Pool of the Buffer being deallocated.</td></tr>
<tr><td>Multi-Use Counter Allocation</td><td>Enter the Pool associated with the counter.</td></tr>
<tr><td>Multi-Use Counter Decrement</td><td>Enter the Pool associated with the counter.</td></tr>
</table></td></tr>
</table>

Table 45 RdCB0_ Variables per Field for BMU

<table>
<tr><th>Register</th><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>RdCB0_DMA_Addr</td><td>PoolID</td><td>20:16</td><td>PoolID —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>0 to 29</td></tr>
<tr><td>BTag</td><td>30</td></tr>
<tr><td>Multi-Use Counter</td><td>30</td></tr>
</table></td></tr>
<tr><td rowspan="4">RdCB0_Sys_Addr</td><td>BTag</td><td>31:16</td><td>Buffer Tag —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>Enter the BTag associated with the Buffer. Legal Range= 0 to 65535.</td></tr>
<tr><td>BTag</td><td>0</td></tr>
<tr><td>Multi-Use Counter</td><td>Enter the BTag associated with the counter. Legal Range= 0 to 65535.</td></tr>
</table></td></tr>
<tr><td>Offset</td><td>15:4</td><td>Offset —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>Buffer Memory Transfer</td><td>Enter the Offset within a Buffer.</td></tr>
</table></td></tr>
<tr><td>CMD</td><td>15:9</td><td>Command —
<table>
<tr><th>Operation Type</th><th>Value</th></tr>
<tr><td>BTag Allocation</td><td>0</td></tr>
<tr><td>Multi-Use Counter Read</td><td>1</td></tr>
</table></td></tr>
<tr><td>Pool</td><td>8:4</td><td>Buffer Pool —
<table>
<tr><th>Operation Type</th><th>Function</th><th>Value</th></tr>
<tr><td>BTag Allocation</td><td>Enter the Pool from which to allocate the Buffers.</td><td rowspan="2">0 to 29</td></tr>
<tr><td>Multi-Use Counter Read</td><td>Enter the Pool associated with the counter.</td></tr>
</table></td></tr>
</table>

Buffer Memory Transactions

Buffer Memory Transactions are Payload Data Block Moves using Control Blocks (WrCB0, RdCB0, RxCB0 and TxCB0). Each is described here.

Using Wr/Rd Control Blocks for Payload Transactions

Writes to SDRAM and reads from SDRAM use: WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr registers and RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr registers. Refer to "Write Control Blocks (WrCB0_ , WrCB1_)" on page 91 and "Read Control Blocks (RdCB0_ , RdCB1_)" on page 95.

These registers (WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr and RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr) are physically located in the respective CPs, XP, and FP and not in the BMU Configuration Space.

Using Rx/Tx Control Blocks for Payload Transactions

Receiving payload to SDRAM and transmitting payload from SDRAM use: RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr registers and TxCB0_Sys_Addr,TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr registers. Refer to "SDP RxByte Processor Receive Control Blocks (RxCB0_, RxCB1_)" on page 98 and "SDP TxByte Processor Transmit Control Block (TxCB0_, TxCB1_)" on page 102.

These registers (RxCB0_Sys_Addr,RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr and TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr) are physically located in the respective CPs, and XP and not in the BMU Configuration Space.

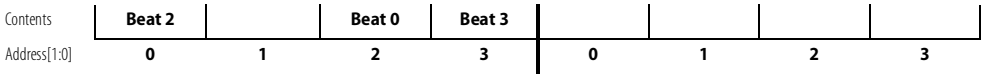
Read/Write Ordering

Since SDRAM is four-way bank interleaved, the BMU uses a round-robin algorithm to choose requests for each bank. This can result in a read response returning in an order other than the order they were issued or acknowledged on the buses.

Unaligned Buffers

The memory controller reads and writes to the SDRAM in naturally aligned 64Byte quantities. Any portion of a naturally aligned 64Bytes block can be read or written; however, special attention must be given to algorithms that require the crossing of a 64Byte boundary. Any transaction that attempts to read or write a data length from an address that causes the least significant two (2) bits of offset to increment from 0x3 to 0x0 will wrap. That is, the other bits of the address are not affected. For example, a write of length 48Bytes to an address with offset bit [1:0]== 0x2 will write memory as shown in Figure 47 on page 225.

Figure 47 Buffer Wrapping



If the intent is to write across the 64Byte boundary then two (2) writes are required. For the same alignment as above, the first write is length 32Bytes at offset bits [1:0] == 0x2 and the second write is length 16Byte at the address of the next contiguous block. Refer to Figure 48 on page 225.

Figure 48 Unaligned Buffer Access

BTag Management Transactions

The BMU maintains a BTag FIFO for BTag allocation and deallocation.

Space for the entire BTag FIFO for each pool is located in the SDRAM, the location defined in the *BTag FIFO Base0* to *BTag FIFO Base29* registers. BTags are allocated from the FIFO and deallocated to the FIFO. The BMU reads BTags in groups of eight (8) and collects eight (8) BTags before writing them back to the FIFO. The BMU maintains an on-chip, hardware-managed cache that can temporarily store BTags from the various pools. The BTag cache typically provides quicker access for allocation and deallocation of BTags and reduces the use of SDRAM bandwidth for BTag management. When the BTag FIFO Cache is empty, operations bypass directly to SDRAM FIFO space. The BTag FIFO Cache space and BTag FIFO SDRAM space are extensions of each other rather than a subset/superset relationship.

BTag Transaction Functions (Operation and Examples)

BTag transactions consist of three (3) different functions: Initialization, Allocation, and Deallocation. BTag transactions are invoked using Control Blocks (WrCB0, and RdCB0). Each is described here along with examples.

BTag Initialization Operation

Warning: *All BTags must be initialized by software before allocating them to access buffer memory.*

BTag Initialization uses a control block (WrCB0) to write starting values from the DMEM of either the requesting CP or XP to initialize BTag FIFO Space.

The base address of the BTag FIFO SDRAM Space is specified in the BTag FIFO Base address register for each pool (*BTag FIFO Base0* to *BTag FIFO Base29*). The size of the Pool*n* BTag FIFO Area= (2Bytes * Number of Buffers). A BTag must be written for each buffer in the pool. The number of BTags per pool must be a multiple of eight (8) because all BTags are written, stored, and read in groups of eight (8).

BTags are 16bit values written to the BTag FIFO Space in groups of 8, 16, 24, or 32. The write control blocks (WrCB) are used for this purpose. The software must generate a buffer in local DMEM containing the 16bit BTags. The BTag numbers themselves can be in any order but they must use all integral values for a given pool from 0 to the number of BTags minus 1. Internally, BMU hardware takes care of allocating the BTags between BTag FIFO Cache Space and BTag FIFO SDRAM Space.

BTag Initialization Example

Buffer Initialization uses a control block (WrCB0) to write the BTags to the BMU from the DMEM of either the requesting CP or XP.

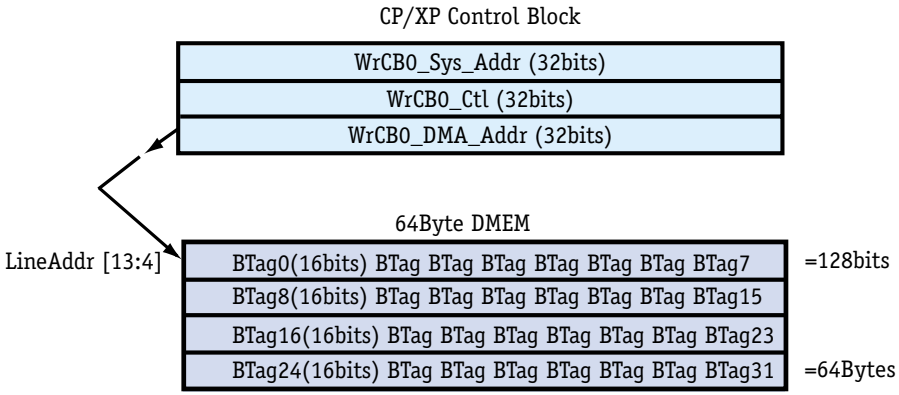
The bits for WrCB0_Sys_Addr, WrCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 46 on page 227.

Table 46 WrCB0_ Settings for BTag Initialization

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. Legal range =16 to 64Bytes. **Length (Bytes) / Number of BTags** 16 / 8 32 / 16 48 / 24 64 / 32
WrCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — Enter 0 for BTag Operation.
	CMD	15:9	**Command** — Enter 0 for BTag Initialization.
	Pool	8:4	**Buffer Pool** — Pool to write to. Legal range= 0 to 29
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 30 for BTag Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA.

The WrCB0_DMA_Addr bits [13:0] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. All 32BTags, (16bit BTags, 128bits) are located inside the 64Byte DMEM as shown in Figure 49 on page 228.

Figure 49 BTag Initialization Implementation



BTag Allocation Operation

Buffer Allocation uses a control block (RdCB0) to read BTags from the BMU into the DMEM of the requesting CP, XP, or FP. Allocation assigns a particular BTag (from the BMU) to be used by a particular processor.

BTag Allocation Example

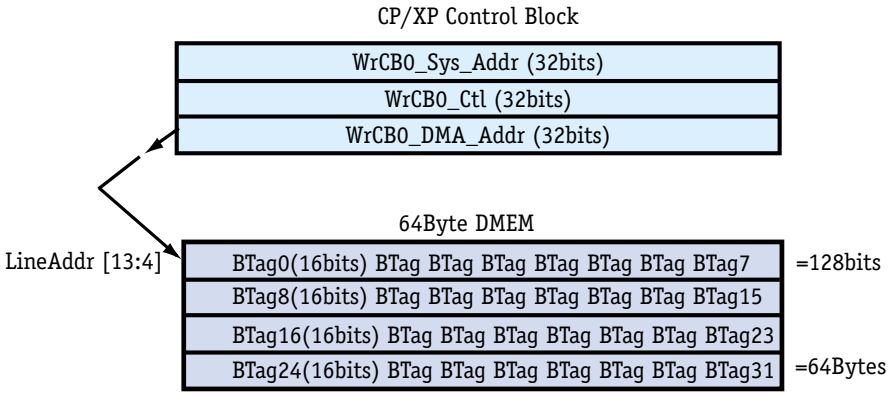
The bits for RdCB0_Sys_Addr, RdCB0_Ctl and RdCB0_DMA_Addr are set as shown in Table 47 on page 229.

Table 47 RdCB0_ Settings for BTag Allocation

Register	Field Name	Bit Position	Description
RdCB0_Ctl	Length	13:4	**Length** — Length of DMA transfer in Bytes. Legal range=16 to 64Bytes. **Length (Bytes) / Number of BTags** 16 / 8 32 / 16 48 / 24 64 / 32
RdCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — Enter 0 for BTag Operation.
	CMD	15:9	**Command** — Enter 0 to BTag Allocation.
	Pool	8:4	**Buffer Pool** — Pool from which to allocate the Buffers. Legal range= 0 to 29
RdCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 30 for BTag Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA.

The RdCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. All 32BTags, (16bit BTags, 128bits) are located inside the 64Byte DMEM as shown in Figure 50 on page 230.

Figure 50 BTag Allocation Implementation



BTag Deallocation Operation

Buffer Deallocation uses a control block (WrCB0) to write BTags back to the BMU from DMEM by either the requesting CP, XP, or FP. Deallocation returns a particular BTag (from a particular CP or XP) back to a pool in the BMU.

BTag Deallocation Example

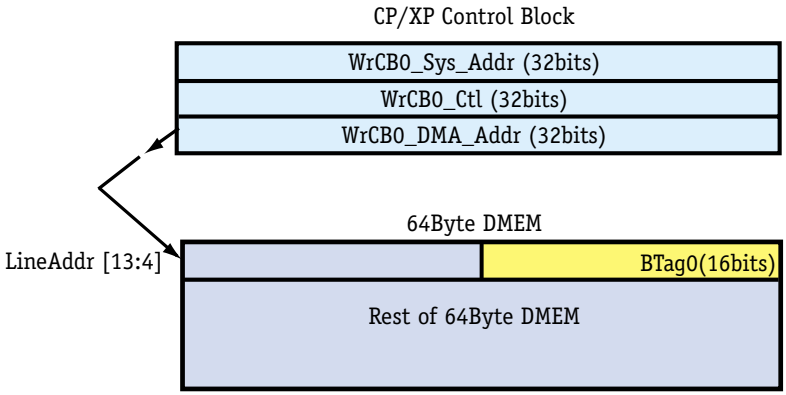
The bits for WrCB0_Sys_Addr, WrCB0_Ctl and WrCB0_DMA_Addr are set as shown in Table 48 on page 231.

Table 48 WrCB0_ Settings for BTag Deallocation

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. Legal value=16Bytes, required to access 1 line of DMEM. **Length (Bytes)** / **Number of BTags**: 16 / 1
WrCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — Enter 0 for BTag Operation
	CMD	15:9	**Command** — Enter 1 for BTag Deallocation.
	Pool	8:4	**Buffer Pool** — Pool of Buffers being Deallocated. Legal range= 0 to 29
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 30 for BTag Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA.

The WrCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is 16Byte aligned. The first two Bytes inside the first 32bit word of the 64Byte DMEM holds the BTag (the first two Bytes) as shown in Figure 51 on page 232.

Figure 51 BTag Deallocation Implementation



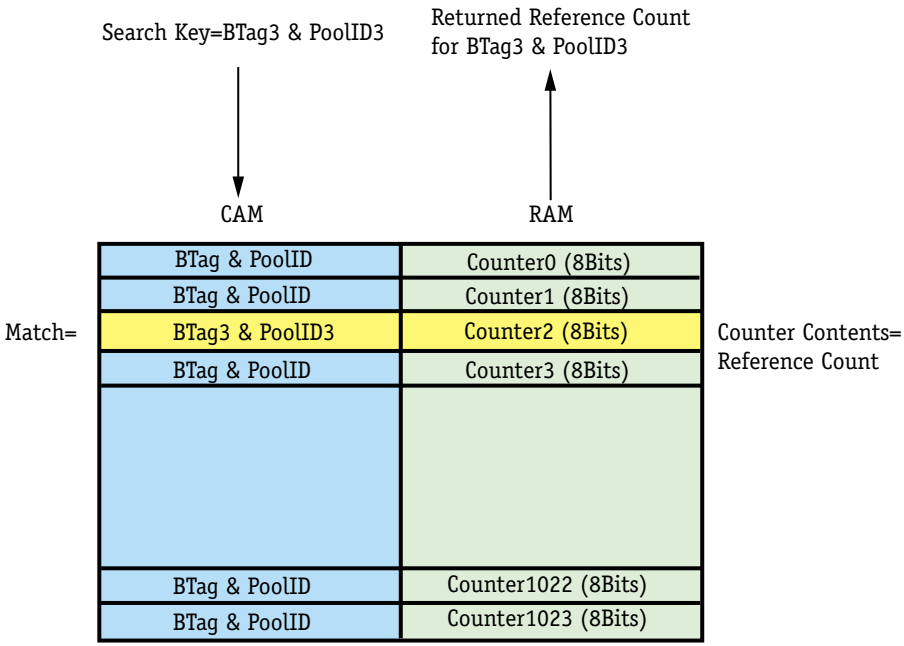
Multi-Use Counter (MUC) Management Transactions

Multi-Use Counters (MUC) are used to track buffer accesses when a buffer has multiple targets (CPs, XP, or FP), such as, a multicast packet. Each time an application running on a particular CP accesses the Multi-Use buffer, its Multi-Use Counter is decremented. When a counter reaches zero, all users have accessed the buffer and the BTag is deallocated.

Typically, the software prefetches a number of BTags without knowing whether or not they are going to be used as single BTags or Multi-Use BTags. That fact only becomes apparent later during processing, *after* the buffer has been written to memory. At that point the software tries to allocate a counter for the Multi-Use BTag from a particular Pool.

There are 1024 8bit counters available. One counter is associated with one (1) BTag at any one time using a *Content Addressable Memory* (CAM) array. BTag & PoolID are stored in the CAM to form the association. An initial Reference Count is stored in the counter. When the software must associate a counter with a buffer, it sends a command to the BMU to allocate a MUC. Refer to Figure 52 on page 233.

Figure 52 Multi-Use Counter Table



MUC Transaction Functions (Operation and Examples)

MUC transactions consist of three (3) different functions: Allocation, Decrement, and Read. MUC transactions are invoked using Control Blocks (WrCB0, and RdCB0). Each is described here along with examples.

MUC Allocation Operation

MUC Allocation uses a control block (WrCB0) to write an initial reference count from DMEM of the requesting CP or XP. MUC Allocation assigns the (BTag & Pool) to a MUC (from the BMU) with the initial reference count.

MUC Allocation Example

The bits for WrCB0_Sys_Addr, WrCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 49 on page 234.

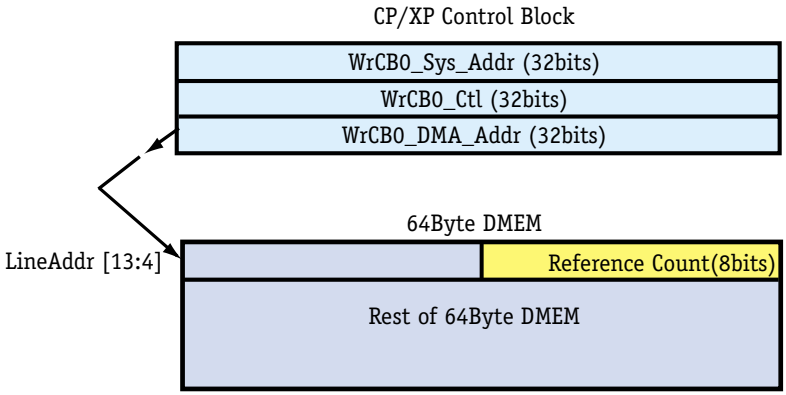
Table 49 WrCB0_ Settings for Multi-Use Counter Allocation

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. Legal value=16Bytes, required to access 1 line of DMEM.
WrCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — The BTag associated with the counter. Legal range= 0 to 65535.
	CMD	15:9	**Command** — Enter 2 for Multi-use Counter Allocation.
	Pool	8:4	**Buffer Pool** — The Pool associated with the counter. Legal range= 0 to 29.
WrCB0_DMA_Addr	PoolID	20:16	**Pool ID** — Enter 30 for Multi-Use Counter Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA.

Length (Bytes)	Number of BTags
16	1

The WrCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is 16Byte aligned. The first Byte inside the first 32bit word of the 64Byte DMEM holds the reference count as shown in Figure 53 on page 235.

Figure 53 Multi-Use Counter Allocation Implementation



MUC Decrement Operation

MUC Decrement uses a control block (WrCB0) to identify a MUC in the BMU and decrement the associated reference count. Only one (1) counter can be decremented per operation. When the MUC decrements to zero, the BMU hardware automatically deallocates the counter and the associated BTag.

MUC Decrement Example

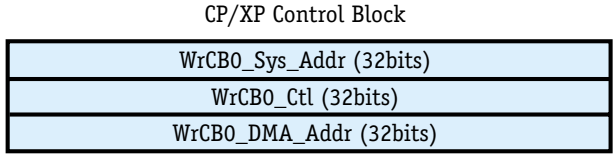
The write control block (WrCB) is used to send the BMU MUC decrement command. The bits for WrCB0_Sys_Addr, WRCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 50 on page 236.

Table 50 WrCB0_ Settings for Multi-Use Counter Decrement

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. Legal value=16Bytes, required to generate a single line operation. **Length (Bytes)** / **Number of BTags**: 16 / 1
WrCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — The BTag associated with the counter. Legal Range= 0 to 65535.
	CMD	15:9	**Command** — Enter 3 for Multi-Use Counter Decrement.
	Pool	8:4	**Buffer Pool** — The Pool associated with the counter. Legal Range= 0 to 29.
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 30 for Multi-Use Counter Operation.
	LineAddr	13:4	**DMEM Line Address** — Not used.

The WrCB0_DMA_Addr bits [13:4] *LineAddr* field is not used as shown in Figure 54 on page 237.

Figure 54 Multi-Use Counter Decrement Implementation



MUC Read Operation

A control block (RdCB0) is used to read specific MUC contents from the BMU into DMEM by either the requesting CP or XP. This function is intended for debug and test purposes. A MUC read request can read one (1) counter per operation.

MUC Read Example

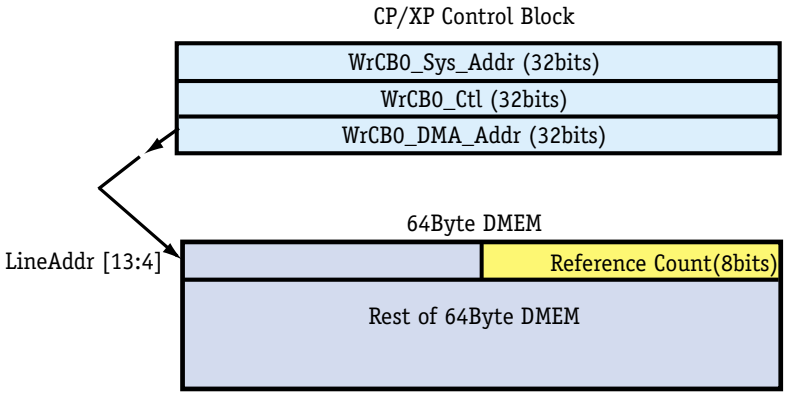
The bits for RdCB0_Sys_Addr, RdCB0_Ctl and RdCB0_DMA_Addr are set as shown in Table 51 on page 238.

Table 51 RdCB0_ Settings for Multi-Use Counter Read

Register	Field Name	Bit Position	Description
RdCB0_Ctl	Length	13:4	**Length** — Length of DMA transfer in Bytes. Legal value=16Bytes, required to access 1 line of DMEM. **Length (Bytes)** / **Number of BTags**: 16 / 1
RdCB0_Sys_Addr	BTag	31:16	**Buffer Tag** — The BTag associated with the counter. Legal Range= 0 to 65535.
	CMD	15:9	**Command** — Enter 1 for Multi-Use Counter Read.
	Pool	8:4	**Buffer Pool** — The Pool associated with the counter. Legal Range= 0 to 29.
RdCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 30 for Multi-Use Counter Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA.

The RdCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is 16Byte aligned. The first Byte inside the first 32bit word of the 64Byte DMEM holds the reference count as shown in Figure 55 on page 239.

Figure 55 Multi-Use Counter Read Implementation



BMU Configuration Space

The BMU has memory-mapped Configuration Space that contains a number of registers. The registers are used for three (3) purposes: physical memory configuration, buffer memory configuration, and test and debug. Refer to Table 52 on page 240 for a list of BMU registers by function.

Table 52 BMU Registers

BMU Register Types	Register Function	Specific Register Details
Physical Memory Configuration	Physical memory size in Bytes. This configuration register is written with a value representing the amount of physical memory that software had determined was present in the system.	See "Memory Size Register (Miscellaneous Function)" on page 521.
	SDRAM controller configuration register. A write to this register tells the SDRAM controller the timing properties of the SDRAM and also initiates the SDRAM configuration process.	See "SDRAM Config Register (Miscellaneous Function)" on page 522.
	A single bit in a register that enables the Single Error Correction/Double Error Detecting (SECDED) error code if set to 1. ECC is disabled if the register bit is set to 0.	See "ECC Enable and Test Enable Register (Miscellaneous Function)" on page 523.
Buffer Memory Configuration	Starting physical address in SDRAM for each Pool.	See "Pool0 Base to Pool29 Base Registers (Buffer Pool Base Address Function)" on page 518.
	BTag shift amount is an encoded version of buffer size for each Pool telling hardware how much to shift the BTag during address calculations.	See "Pool0 BTag Shift to Pool29 BTag Shift Registers (Buffer Size for a Pool Function)" on page 519.
	Starting physical address in SDRAM for the BTag FIFO for each Pool.	See "BTag FIFO Base0 to BTag FIFO Base29 Registers (BTag FIFO Base Address Function)" on page 520.
	The number of BTags in each Pool.	See "Num BTags0 to Num BTags29 Registers (Number of BTags in a Pool Function)" on page 520.

Table 52 BMU Registers (continued)

BMU Register Types	Register Function	Specific Register Details
Test and Debug	This read-only register counts the number of single Error Correction Code (ECC) errors that have occurred.	See "Single ECC Errors Register (Miscellaneous Function)" on page 523.
	Control for ECC read and write test modes.	See "ECC Enable and Test Enable Register (Miscellaneous Function)" on page 523.
	BMU C-5 NP debug register in canonical format.	See "Debug Config Register (Miscellaneous Function)" on page 524.
	These two (2) registers capture the write address of a transaction that led to a write memory violation.	See "Wr_Mem_Violation_Hi Register (Miscellaneous Function)" on page 525 and "Wr_Mem_Violation_Lo Register (Miscellaneous Function)" on page 525.

Test and Debug Registers

The BMU contains various registers to provide test and debug access to internal state.

Memory Error Reporting

The *Single ECC Error* register is reset to 0 by hardware. After reset, the register counts the number of corrected single-bit ECC errors encountered during SDRAM access. Single-bit, corrected errors are not reported anywhere else.

Error conditions detected by the BMU are generally reported back to the requester except for single-bit ECC corrected errors and some violations on write transactions. The *Wr_Mem_Violation_Hi* and *Wr_Mem_Violation_Lo* registers capture the global or payload address of transactions that cause write violations.

ECC Test Modes

The *ECC_Enable and Test _Enable* register controls error checking during normal operation. In addition, the ECC Enable and Test Enable register provides ECC write and read test modes for testing ECC RAMs and portions of the chip data path. When bit [1] *ECC Write Test* field is enabled, the *ECC WriteTest Bits* field bits [10:2] provide the test ECC write data directly rather than the normal ECC generation logic. When this mode is enabled, all four (4) 16Byte beats of a payload write transaction write the same test ECC write data. When *ECC Read Test Enable* field bit [11] is enabled, rather than checking the ECC, the ECC bits are returned directly from SDRAM in the least significant 9bits of the data on a payload read transaction. All four (4) 16Byte beats of the payload read return the associated ECC data for the beat.

Debug Register

The BMU has a tap for the global debug logic. The *Debug Config* register controls multiplexors that allow selection of various BMU events or transactions for routing to the global debug counters located in the XP. Refer to "XP Debug Mode Register (XP Mode Configuration Function)" on page 490, and "Debug Counter0 Control Register (XP Configuration Function)" on page 483.

For complete details about specific registers go to their reference. Refer to "Buffer Management Unit (BMU) Configuration Registers" on page 512.

BMU Setup

Prior to the CP, XP, or FP accessing the SDRAM, the BMU must be set up properly, configured, and initialized as described in the following steps and using the applicable registers listed in Table 52 on page 240.

Warning: *Attempting to access a buffer pool before it is set up results in unpredictable behavior.*

1 Configure physical memory:

   a Write encoded physical memory size (either 64, 128, or 256MBytes) to the *Memory Size* register.

   b Write memory timing parameters to the *SDRAM Config* register.

   c Hardware disables ECC error correction and detection on reset. Write a 1 to bit [0] of the *ECC Enable* register to enable checking.

2 Configure buffer memory:

   a Write the physical address for the starting location in SDRAM for each Pool used into the *Pool0 Base* to *Pool29 Base* registers. Pool Base registers for unused Pools need not be configured.

   b Write the BTag shift amount to set buffer size for each Pool used into the *Pool0 BTag Shift* to *Pool29 BTag Shift* registers. Pool BTag Shift registers for unused Pool need not be configured.

   c Write the physical address for the starting location in SDRAM for the BTag FIFO for each Pool used into the *BTag FIFO Base0* to *BTag FIFO Base29* registers. BTag FIFO Base registers for unused Pools need not be configured.

   d Write number of BTags for each Pool used into the *Num BTag0* to *Num BTag29* registers. The number of BTags per pool must be a multiple of 8. Num BTag registers for unused Pools need not be configured.

3 Initialize BTags:

   a All BTags must be initialized before they can be allocated. Initialization software must write the BTag values to the BTag FIFO for each Pool used using the BTag initialization payload transaction. Refer to "BTag Initialization Example" on page 227. BTags can be written in groups of 8, 16, 24, or 32 at a time. Specific values can be written in any order, but when completely initialized each FIFO must use all the BTags numbered from 0 to Pool size minus 1.

If memory size is not known at configuration time, software must auto size physical memory. Hardware sets Memory Size to the maximum value at reset. Software configures a temporary pool or fabricates a BTag directly (in this case 0) without allocating and writes to location 0 of physical memory. Then software writes to location 64M and reads back location 0. If location 0 is overwritten, the physical memory limit has been reached and the address wrapped. If location is not overwritten, software tests the next physical memory boundary, that is, 128M, and so on until the physical memory limit is discovered. Then the Memory Size register can be written and buffer memory configured for normal operation.

Chapter 6

Table Lookup Unit

Chapter Overview

This chapter covers the following topics:

Table Lookup Unit (TLU) Overview

The Table Lookup Unit (TLU) provides access to application-defined topology, control, and statistics tables in external SRAM. It accesses an external SRAM array operating at up to 142MHz. Communication between the processors (CPs, XP, and FP) and the TLU is carried out via messages passed on the Ring Bus. Each processor (16CPs, XP, and FP), as well as the TLU is a *node* on the Ring Bus. The Ring Bus uses a 64bit wide data path. Refer to "Ring Bus Overview" on page 370.

The internal architecture of the TLU is extensively pipelined. Thus, the TLU can service a number of outstanding requests simultaneously to ensure the most efficient use of the available external SRAM cycles. Near 100% cycle utilization of the SRAM array is achieved.

The TLU supports several types of table lookup algorithms and provides resources for efficient generation of table entry addresses in SRAM, "hash" generation of addresses, and binary table searching algorithms for both exact-match and longest-prefix-match strategies.

The TLU also provides resources for efficiently managing and manipulating table keys and associated data. Table entry insertion is performed by the XPRC, CPRCs, or by memory mapped access to the Ring Bus registers on the XP from an external host processor.

To optimize application performance, the TLU allows: up to eight (8) lookup algorithms, the mapping of sixteen (16) lookup tables, supports seven (7) different table types, and configurable table sizes.

The associated data maintenance facilities of the TLU also serve as a high-performance statistics accumulation resource and as an intermediate storage medium for segmentation and reassembly (SAR) operations. The C-5 NP uses external 64bit wide Pipelined Bursting Static RAM (SRAM) modules for storage of its tables. These modules allow implementation of tables of 2^{20} x 64bit entries at a cycle time of up to seven (7 nanoseconds) using 4Mbit SRAM technology.

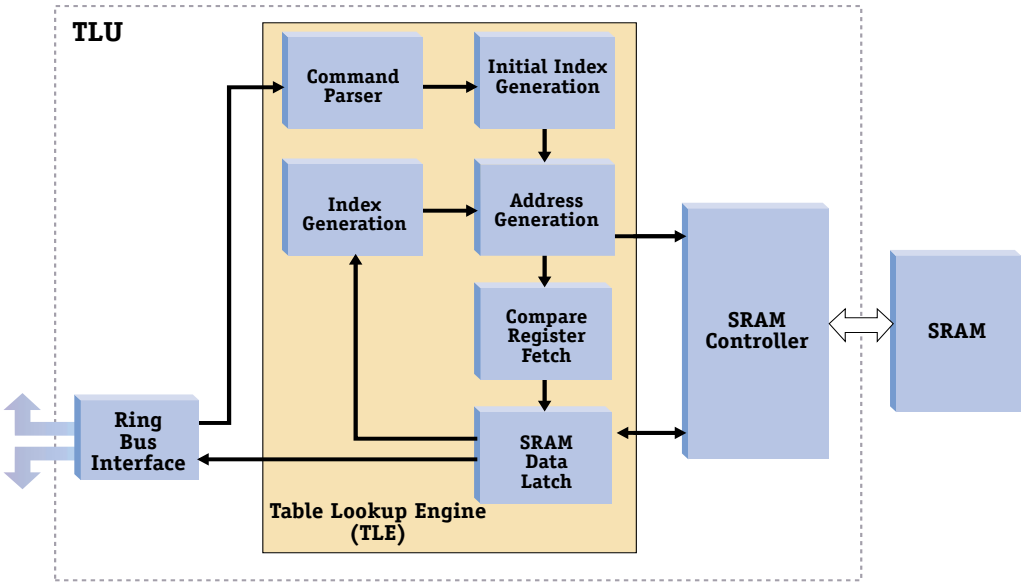
TLU Major Components

The major components of the TLU are listed in Table 53 on page 247. In addition, Figure 56 on page 247 shows the TLU Block Diagram.

Table 53 Major Components of the TLU and Their Functions

Item	Function
Table Lookup Engine (TLE)	Performs table lookups. The TLE comprises six (6) blocks: command parser, initial index generation hash, address generation, compare register fetch, SRAM data latch, and index generation. The TLE supports seven (7) different table types: Indexed Pointers, Hash, Trie, VP Trie, Key, Data and External.
SRAM Controller	Manages the external storage arrays. The bandwidth to external SRAM is 64bits at 133MHz, achieving an aggregate capacity of 1.04GBytes per second.

Figure 56 TLU Block Diagram



TLU Flow Process

The TLU consists of several blocks that allow you to implement a variety of table lookup algorithms to meet your application needs. In general, its functional blocks are organized in a basic loop that performs the following functions:

1. Parses a Ring Bus command.
2. Calculates the initial index based on a Key (for example, the head of a Trie or an initial hash value).
3. Fetches the Key.
4. Compares the index value with the Key (and if they match go to step 6, if no match go to step 5).
5. Calculates a new index (and then go back to step 3).
6. Fetches the data at the current index.
7. Returns the data to the CPs or XP via the Ring Bus, or to the FP via a dedicated path between the TLU and FP.

Each block has several programmable, pipelined stages. Each stage, passes data to the next downstream stage. At any given time, every stage can have valid data, allowing many TLU operations to occur simultaneously.

TLU Flow Process Details

This section describes the transactional flow through the TLU in more detail. All of the blocks involved in the flow are shown in Figure 56 on page 247.

Ring Bus Interface and Command Parser

The Ring Bus Interface block of the TLU is the only interface between the TLU and the rest of the C-5 NP. The Ring Bus comprises a receive (Rx) and a transmit (Tx) side.

- The Receive (Rx) section of the Ring Bus Interface monitors the Ring Bus for commands destined for the TLU. When a command is received (Rx), the command is removed from the Ring Bus and sent to the TLU's Command Parser block. TLU commands can be either an *indication* or a *request* message type. Indications are messages from a source node to a destination node that automatically generates a confirmation from hardware. Whereas, requests are messages that generate a response through software. Refer to Table 56 on page 255 for list of the TLU commands, descriptions and parameters.

- The Transmit (Tx) side of the Ring Bus Interface is responsible for returning data to the requesting CPs or XP. Each Ring Bus slot message returns at most 32Bytes. If more than 32Bytes are requested, then multiple messages are transmitted. Since all Ring Bus responses are initiated by a specific request, the only command information passed back to the requesting node (CPs, or XP) is the sequence number. Refer to"Ring Bus Registers" on page 106, and "Ring Bus Overview" on page 370.

Responses sent to the TLU are discarded.

Initial Index Generation

The Initial Index Generator block calculates the initial index into a table.

Address Generation

The Address Generation block calculates the SRAM address for the next SRAM access. An address is calculated as follows:

*SRAM address = (base_address * 256) + index + offset*

The TLU does not support the burst access feature of the SRAMs. Instead, it has an internal burst counter that automatically increments the address for consecutive reads and writes.

Compare Register Fetch

The Compare Register Fetch block is responsible for comparing the last SRAM read data with the current key. Compares occur whenever a find type command (*Find*, *Findw*, or *Findr*) accesses a Key table. Successful compares allow *Findw* commands to execute and F*indr* commands to return TLU response data.

Each node compare of a Trie or Hash table takes a SRAM cycle.

SRAM Data Latch

The SRAM Data Latch block latches the data from an SRAM read so that it can be processed by the next stage.

Index Generation

The Index Generator block incrementally calculates the next index into a table. Branch decisions are calculated by examining a number of different operands. These include: the current Key, fields from the last Key Fetch operation, the current index, and a variety of internal registers. These internal registers are used to keep track of the current bit position for tries, pointers for most significant prefix matching, and other general record keeping functions. After the first calculation is performed, the Index Generation block calculates the next index for iterative functions.

The Index Generation block can be programmed on a per table basis using one of two (2) methods for generating a new index. The two (2) methods include:

- Incrementing of the previous index.
- Using a pointer from the previous Key Fetch operation. This method is used for all tries and hashes. For lists with multiple pointers (that is, tries), the previous stage passes control information to this block describing the correct pointer to choose.

SRAM Controller

The TLU's SRAM Memory Controller is designed to maximize the bandwidth utilization of the SRAMs. The SRAM Memory Controller supports SRAM frequencies to 133MHz using 3.3V LVTTL. The SRAM physical interface supports SRAM technologies up to 64Mbits. Refer to Table 54 on page 250.

Table 54 TLU SRAM Configurations

SRAM Technology	Minimum Table Size, 1 Bank	Maximum Table Size, 4 Banks
1Mb (32k x 32pins)	256kB	1MB
2Mb (64k x 32pins)	512KB	2MB
4Mb (256k x 18pins)	2MB	8MB
8Mb (512k x 18pins)	4MB	16MB
64Mb (4M x 18pins)	32MB	(128MB is not supported)

The TLU can perform a read or write operation every cycle using ZBT SRAMs. The physical interface provides four (4) copies of CEx (chip enable) and WEx (write enable), as well as, inverted copies of the four (4) MSBs of the address. The multiple CExs and WExs are used to decrease loading on these signals. The inverted address bits are used for bank expansion. They are tied to the CE2 and CE2x inputs of the ZBT RAMs for bank expansion.

The SRAM controller does not support bursting. Sequential accesses are generated using an internal address incrementer.

TLU Supported Table Types

Networking systems use synchronized tables containing topology and control information to make forwarding and characterization decisions. Such tables are accessed in the forwarding path for lookup resolution and forwarding decisions, and are typically managed by an agent that runs on an application processor that adds, removes, and modifies entries in these tables. Examples of such databases would be an IP Routing Table or an ATM Virtual Connection (VC) table. The TLU supports seven (7) different types of tables listed in Table 55 on page 251 along with their functions. Each table type has a particular data structure. Search algorithms (lookup algorithms) are constructed by linking various tables together.

The TLU supports four (4) different Key sizes: 32, 48, 96 and 112bit. In addition, intermediate key sizes are supported by masking unused bits to zero.

Warning: *Ensure all table memory areas are initialized before performing table inserts or lookups. Possible unreported errors providing erroneous data could occur.*

Table 55 Supported Table Types

Item	Function
Indexed Pointer	Contain entries that point to an entry (via an index) in another table. This type of table can be used to evaluate a portion of a lookup key.
Hash	Used for *exact match* algorithms. That is, algorithms that require a lookup key to be matched exactly. Hash tables evaluate the lookup key via a *hash function*.
Trie	These binary trees provide an efficient means for resolving Hash table collisions.
Variable Prefix (VP) Trie	Used for *most-specific match* algorithms.
Key	Used in exact match algorithms, and contain both the key of an entry and the associated data.
Data	Contain the data associated with an entry. Data tables can be used as a stand-alone table or as the last table (that contains the associated data for a previously evaluated key) in a set of tables.
External	Used to interface with external lookup engines when it is swapped with a SRAM bank.

Table memory in SRAM is divided into sixteen (16) separate tables that are numbered from (0 to 15).

- Tables (0 to 7) can consist of the following types:
  - Data
  - Variable Prefix (VP) Trie
  - Trie
  - Hash
  - Index Pointers
  - Key
- Tables (8 to 15) can *only* consist the following types:
  - Key
  - Data

Each table can be divided into table entries (from 256 to 1024k) that are accessed through an index. Table entries are a fixed length and can be (from 8Bytes to 1024Bytes) in length. The table number, number of entries, and entry size are configurable using the TLU configuration registers.

After creation of the tables, an application has the ability to specify the order in which lookups are performed through a series of tables. Up to three (3) tables can be *linked* together to more efficiently obtain a lookup result. Generally, the table linking is configured using the *Lookup_Algorithm_Configuration1* register. In a linked-table scenario, the last table must always point to either a Data or Key table, where the associated data is stored.

TLU Table Mapping

The TLU uses *virtual* tables (*VTB#*) to access physical tables (TBL#). The TLU has sixteen (16) virtual tables, these virtual tables are mapped to physical tables using the *Virtual_Table_Configuration* register. Each virtual table can mapped to anyone of the sixteen (16) physical tables. This provides the ability to change the virtual table to physical table mapping on the fly.

In addition, virtual tables allow you to build and update one (1) copy of a topology table in the TLU while another table is being used by the forwarding path for lookups. The TLU can switch between these two (2) tables simply by adjusting the value of the *Virtual_Table_Configuration* register to point to the new table. This technique is called hot swapping of tables.

Mapping Virtual Tables to Physical Tables

The *Virtual_Table_Configuration* register controls the mapping of virtual tables to the physical tables. By changing the value of the *TBL#* field in the *Virtual_Table_Configuration* register, the application can start performing its lookups in the new table as soon as the new value is written. The *Virtual_Table_Configuration* register is written to using the TLU *WriteReg* (0x0/0x10) command.

Ensure the WriteReg command's FLUSH field bit [48] is set. This ensures that all table lookups in progress are completed before changing the virtual to physical table mapping.

Figure 57 on page 254 shows an example where there are two (2) copies of the Key table:

- One that is actively being used in the forwarding process
- The other is being populated by an application running on the XP or an external host

Figure 57 Virtual Table Linking

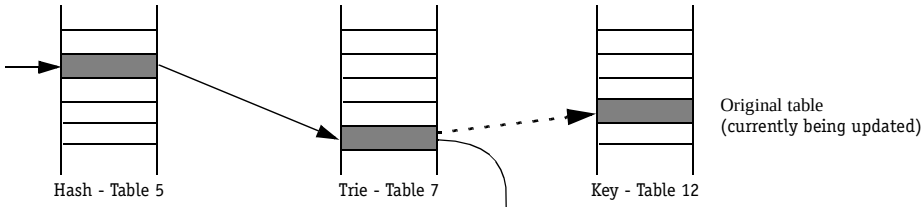


Table Number	
Virtual	**Actual**
5	5
7	6
~~12~~	~~12~~
12	13

TLU Commands Overview

The following section describes the eleven (11) commands used to control the TLU. These commands are sent to the TLU via the Ring Bus. Refer to Table 56 on page 255 for a list of the TLU commands with their parameters, command ID/extended command ID, returned data, and function.

Table 56 TLU Commands

Command (Parameters)	Command ID/Extended Command ID	Returned Data	Function
Write (VTB#, IDX, MSK, DATA, OFF, LEN)	0x2	None	Write data into a virtual table at index.
Read (VTB#, IDX, OFF, LEN	0x3	Data	Reads data from a virtual table.
Find (ALG#, KEY)	0x6	Virtual Table Value, Index, or Error	Finds a *key* using *ALG#*. Sets Ring Bus Error Flag if *key* is not found.
Findw (ALG#, KEY, DATA, OFF, LEN)	0x4	Error	Writes *data* into a table using a *key*. Sets Ring Bus Error Flag if the key is not found.
Findr (ALG#, KEY,DATA, OFF, LEN)	0x5	Data, or Error	Reads *length* double words of *data* from a v*table#* using a *key* at *offset* double words. Sets Ring Bus Error Flag if the key is not found.
XOR (VTB#, IDX, DATA/PCRC, OFF, MSK, LAST)	0x1	None, or CRC in CRC mode.	XORs up to a 32bit value to *offset*. Only masks of up to four (4) consecutive bytes are valid. Note: A special CRC mode exists for CRC calculations.
Add (VTB#, IDX, DATA, OFF, MSK)	0x7	None	Adds up to a 32bit value to *offset*. Only masks of up to four (4) consecutive bytes are valid.
WriteReg (ADDR, DATA)	0x0/0x10	None	Write *data* to TLU register at *ADDR*.
ReadReg (ADDR, DATA)	0x0/0x11	Data	Read *data* from TLU register at *ADDR*.
Echo (DATA)	0x0/0x04	Data	Return *data* from TLU. For test purposes.
NOP ( )	0x0/0x05	None	Inserts a NOP into the TLU pipe. Used to skip an SRAM access during that cycle.

TLU Command Parameters

The TLU command parameters along with their functions are listed in Table 57 on page 256.

Table 57 TLU Command Parameters

TLU Command Parameter Field	TLU Command Parameter Name	Function
VTB#	Virtual table number	Virtual tables are mapped to a physical tables using the *Virtural_Table_Configuration* register. The actual physical table (*TBL#*) accessed is translated using the *Virtual_Table_Configuration* register.
IDX	Table index number	The index number points to a specific entire in a table. The index is used by the *Read*, *Write*, *Add*, and *XOR* commands. It is a 24bit value.
OFF	Offset	The *offset* in 8Byte increments into the table entry. The legal range= 0 to 127. The actual SRAM address is given by: (base_address [table# [vtable#]] * 256)+ (index << size [table# [vtable#]]) + offset
LEN	Length	The number of 8Byte words to read. Valid values are: • 1 for 8Bytes • 2 for 16Bytes • 4 for 32Bytes
MSK	Mask	Byte *mask* for *Writes* and Arithmetic Logic Unit (ALU) operations. Each bit corresponds to one (1) Byte.
ALG#	Algorithm number	The *lookup algorithm* is a method used for linking up to three (3) tables together in order to perform a better search with the last table storing the associated data. Up to eight (8) algorithms are supported. The legal range= 0 to 7, (0x400 to 0x407). Using the *Lookup_Algorithm_Configuration1* register you can define each algorithm. The specific algorithm number selects which of the eight (8) lookup algorithm to use. Refer to Table 64 on page 286 for an example.
KEY	Key	The *key* is used for all find commands (*Find*, *Findw*, *Findr*), to generate an index using the *ALG#*. The TLU supports four (4) different Key sizes: 32, 48, 96 and 112bit. In addition, intermediate key sizes are supported by masking unused bits to zero.
ADDR	Address	*Address* of a register to write or read.

Table 57 TLU Command Parameters (continued)

TLU Command Parameter Field	TLU Command Parameter Name	Function
DATA	Data	*Data* refers to the *DATA* field in each individual TLU command format. The purpose of the data field varies based on the TLU command. Therefore, for the specific definition of the *DATA* field refer to the particular TLU command format.
CRC	CRC enable	This enables the CRC mode.

CRC parameter is only used in the CRC Mode associated with the XOR command. Refer to "XOR Command" on page 270.

Detail TLU Commands

Each of the eleven (11) TLU commands are described in the following section along with its purpose, command ID/extended ID, fields, bit positions. Also provided, where applicable are the command's data alignment rules, returned data and error types.

Write Command

The *Write* command is used to write data to the TLU's SRAM. Two (2) types of writes are available:

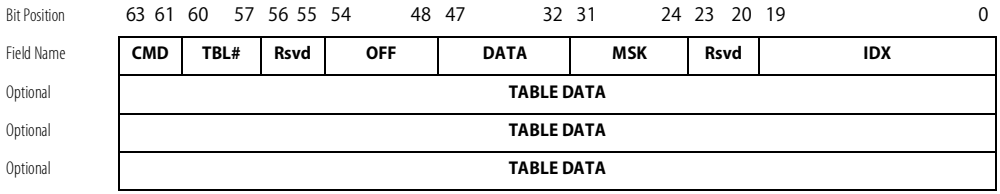
- The first type, is a masked write from one (1) to two (2) bytes in length. The bytes are selected using the *MSK* field bits [31:24]. Bytes must be contiguous and aligned on word boundaries (that is, a mask of 0x06 is illegal, while 0x03 and 0xC0 are both legal mask values). The write data is contained in the *DATA* field bits [47:32] in the first control word.
- The second type, uses consecutive Ring Bus slots to write to consecutive SRAM locations. Since the write command occupies the first Ring Bus slot, up to three (3) SRAM locations may be written consecutively. The TLU uses the value in the Ring Bus length field to determine the actual number of SRAM location to write. For two-slot writes, the mask can be set at either 0x0F or 0xF0 to write 32bits to the SRAM; or set to 0xFF to write 64bits. For four-slot writes the mask field should be set to 0xFF.

Data written with masks set to anything other than 0xFF generate a read-modify write cycle. This means that a read occurs and then four clocks later the value is written back to the SRAM. Note that since this operation is not locked, another process could be executing a read-modify write on the same address resulting in corrupted data.

Write Command Format

Purpose Writes data to the TLU's SRAM.

Command ID 0x2



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x2 for Write.
TBL#	60:57	**Table Number** — Identifies table number. Legal range= 0 to 15.
Reserved	56:55	Read as zero.
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table for read/write. Legal range= 0 to 127.
DATA	47:32	**Data** — Data field for 8 or 16bit (single bus slot) writes. *MSK* field determines data alignment in SRAM.
MSK	31:24	**Write Mask** — Byte mask for single slot writes (8Bytes). The mask is also used for two-slot and four-slot writes.
Reserved	23:20	Read as zero.
IDX	19:0	**Index** — Designates a table entry in a given TBL#.
TABLE DATA	63:0	**Table Data** — Data to write to SRAM. If these fields are present, then the *Data* field bits [47: 32] in the first slot is ignored. Set *MSK* field bits [31:24] to 0xFF.

Write Command Data Alignment Rules

- 8bit writes are placed in the *DATA* field of the first slot and aligned to a byte boundary.
  - For masks of 0x01, 0x04, 0x10, and 0x40 data is stuffed into *DATA* field bits [39:32] of slot1.
  - For masks of 0x02, 0x08, 0x20, 0x80 data is stuffed into *DATA* field bits [47:40].
- 16bit writes are also placed in the *DATA* field bits [47:32] of the first slot.
- 32bit write data is stuffed into *TABLE DATA* field bits [31:0] of the second slot.
- 64bit write data is stuffed into *TABLE DATA* field bits [63:0] of the second slot.
- 192bit write data is stuffed into *TABLE DATA* field bits [63:0] of the second, third, and fourth slot.

Write Command Returned Data

The *Write* command does *not* return any data.

Write Command Error Types

The *Write* command does *not* return any errors.

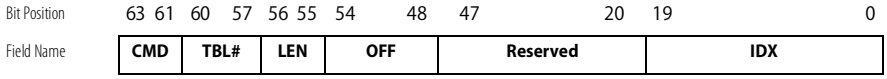
Read Command

The R*ead* command is used to read data from the TLU SRAM.

Read Command Format

Purpose Read data from the TLU's SRAM.

Command ID 0x3



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x3 for Read.
TBL#	60:57	**Table Number** — Identifies a table number. Legal range= 0 to 15.
LEN	56:55	**Length** — Tells the TLU how many SRAM locations to access. All SRAM reads are in multiples of 8Bytes. Number of 8Byte double words to read - 1. Legal ranges are detailed here:
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table for write. Legal range= 0 to 127.
Reserved	47:20	Read as zero.
IDX	19:0	**Index** — Designates a table entry in a given TBL#.

Encoded Value	Actual Length
0	1
1	2
2	Illegal (Not supported)
3	4

Read Command Data Alignment Rules
The Read command does *not* have these rules.

Read Command Returned Data
The *Read* command returns the requested data to the calling function (CP or XP). If *index* or *offset* is out of range, the returned data is undefined.

Read Command Error Types
The *Read* command does *not* return any errors.

Find Command

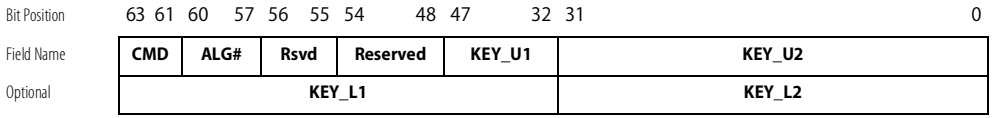
The *Find* command attempts to locate a key in a table using a preprogrammed, linked-table algorithm as specified by *ALG#*.

Prior to executing this command, ensure the Lookup_Alogorithm_Configuration1 register references the ALG#, using the LNK1, LNK2, or DATA fields. Also, ensure that reference coincide with the ALG# field bits [60:57] of the Findr command format. If not, the Find command returns a indeterminate value. Refer to "TLU Table Mapping" on page 253.

Find Command Format

Purpose Find an index and a table, given a key.

Command ID 0x6



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x6 for Find.
ALG#	60:57	**Algorithm Number** — Identifies an algorithm to use for the lookup. Legal range= 0 to 7.
Reserved	56:55	Read as zero.
Reserved	54:48	Read as zero.
KEY_U1	47:32	**Key Upper 1** — Upper 16bits of a 48bit or 112bit key.
KEY_U2	31:0	**Key Upper 2** — All 32bits of a 32bit key; lower 32bits of 48bit key; upper 32bits of 96bit key; upper-middle 32bits of 112bit key.
KEY_L1	63:32	**Key Lower 1** — Middle 32bits of 96bit key; lower-middle 32bits of 112bit key.
KEY_L2	31:0	**Key Lower 2** — Lower 32bits of 96bit key or 112bit key.

Find Command Data Alignment Rules
The *Find* command does *not* have these rules.

Find Command Returned Data
- The *Find* command returns a *virtual table* value and an *index*. The data is formatted with the *index* in the *KEY_U2* field bits [31:0] and *table* in *KEY_U1* field bits [35:32].
- If the *key* is *not* found, then undetermined data is returned to the calling function (CP or XP) and the Ring Bus error bit is set to one (1).

Find Command Error Types
If a *Find* command takes more than 255 SRAM accesses, the *Find* command times out, the Ring Bus error bit is set to one (1), and the data field bits [31:0] is set to 0x2.

Findw Command

The *Findw* command performs a *Find* followed by *Write*. As with a *Find* command, it locates a *key* in a table using a preprogrammed, linked-table algorithm as specified by *ALG#*.

Prior to executing this command, ensure the Lookup_Alogorithm_Configuration1 register references the ALG#, using the LNK1, LNK2, or DATA fields. Also, ensure that reference coincides with the ALG# field bits [60:57] of the Findw command format. If not, the Findw command returns a indeterminate value. Refer to "TLU Table Mapping" on page 253.

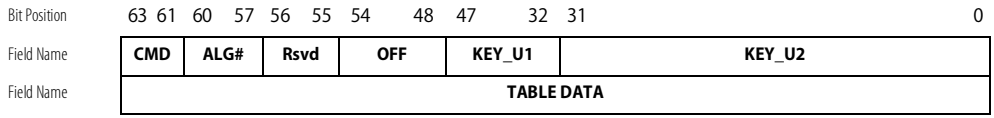
Findw Command Format

The Findw has a similar format to Write, except it can only write 8Bytes of data, and does not support write masks.

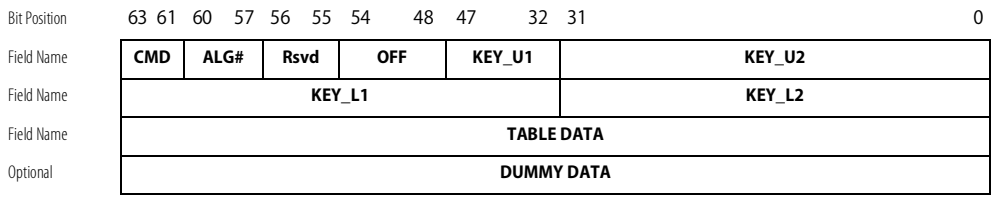
Purpose — Find an index and a table, given a key and write data to the TLU's SRAM. Note: The Findw accommodates both a 2-slot and 4-slot format, as shown here.

Command ID — 0x4

Findw 2-slot format:



Findw 4-slot format:



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x4 for Findw.
ALG#	60:57	**Algorithm Number** — Identifies an algorithm. Legal range= 0 to 7.
Reserved	56:55	Read as zero.

Field Name	Bit Position	Description
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table for write. Legal range= 0 to 127.
KEY_U1	47:32	**Key Upper 1** — Upper 16bits of a 48bit or 112bit key.
KEY_U2	31:0	**Key Upper 2** — All 32bits of a 32bit key; lower 32bits of 48bit key; upper 32bits of 96bit key; upper-middle 32bits of 112bit key.
KEY_L1	63:32	**Key Lower 1** — Middle 32bits of 96bit key; lower-middle 32bits of 112bit key.
KEY_L2	31:0	**Key Lower 2** — Lower 32bits of 96bit key or 112bits key.
TABLE DATA	63:0	**Table Data** — Data to write to SRAM.
DUMMY DATA	63:0	**Dummy Data** — Dummy field sent when long keys (>48bits) are used.

Findw Command Data Alignment Rules

The *Findw* command does *not* have these rules.

Findw Command Returned Data

The *Findw* command does not return any data.

Findw Command Error Types

- If the *Findw* command takes more than 255 SRAM accesses, the *Findw* command times out, the Ring Bus error bit is set to one (1), and the data field bits [31:0] is set to 0x2.
- If the *key* is not found, then the Ring Bus error bit is set to one (1), and the data field bits [31:0] is set to 0x1.

Findr Command

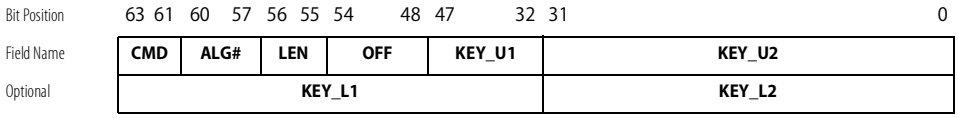
The *Findr* command performs a *Find* on a *key* and then a *Read*. As with a the *Find* command, it locates a *key* in a table using a preprogrammed, linked-tables algorithm as specified by *ALG#*.

Prior to executing this command, ensure the Lookup_Alogorithm_Configuration1 register references the ALG#, using the LNK1, LNK2, or DATA fields. Also, ensure that reference coincides with the ALG# field bits [60:57] of the FIndr command format. If not, an the Find commands returns an indeterminate value. Refer to "TLU Table Mapping" on page 253.

Findr Command Format

Purpose Find an index and a table, given a key and read data from the TLU's SRAM.

Command ID 0x5



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x5 for Findr.
ALG#	60:57	**Algorithm Number** — Identifies an algorithm. Legal values are 0 to 7.
LEN	56:55	**Length** — Tells the TLU how many SRAM locations to access. All SRAM reads are in multiples of 8Bytes. Number of 8Byte double words to read - 1. Legal ranges are detailed here:
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table entry for read. Legal range= o to 127.
KEY_U1	47:32	**Key Upper 1** — Upper 16bits of a 48bit or 112bit key.

Encoded Value	Actual Length
0	1
1	2
2	Illegal (Not supported)
3	4

Field Name	Bit Position	Description
KEY_U2	31:0	**Key Upper 2** — All 32bits of a 32bit key; lower 32bits of 48bit key; upper 32bits of 96bit key; upper-middle 32bits of 112bit key.
KEY_L1	63:32	**Key Lower 1** — Middle 32bits of 96bit key; lower-middle 32bits of 112bit key.
KEY_L2	31:0	**Key Lower 2** — Lower 32bits of 96bit key or 112bit key.

Findr Command Data Alignment Rules

The *Findr* command does *not* have these rules.

Findr Command Returned Data

The *Findr* command returns the requested data to the calling function (CP or XP). If offset is out-of-range, the returned data is undefined.

Findr Command Error Types

- If a *Findr* command takes more than 255 SRAM accesses, the command times out, the Ring Bus error bit is set to one (1), and the data field bits [31:0] is set to 0x2.
- If the *key* is not found, then the Ring Bus error bit is set to one (1), and the data field bits [31:0] is set to 0x1.

Add Command

The *Add* command behaves similarly to the *Write* command, except that data is added to the existing data in the table. *Add* supports 8, 16, and 32bit add-ends.

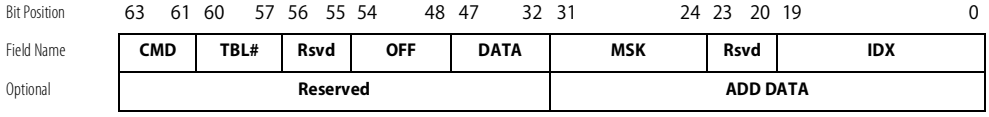
- For 8 and 16bit add-ends, the data is packed in the *DATA* field bits [47:32] of the first register. The *MSK* field bits [31:24] is used to identify the correct byte lane of the target add.
- For 32bit add-ends, the data is located in the lower 32bits of the *ADD DATA* field bits [31:0] in the optional register. The *MSK* field [31:24] is used to indicate if the target is aligned in the upper half of the SRAM (0xF0) or the lower half (0x0F) of the SRAM. To read the result of the *Add*, issue a *Read* at least four (4) clocks after the *Add* has been issued.

The Add command generates a read-modify-write cycle. This means that a read occurs and then four (4) clocks later the value is written back to the SRAM. This operation currently is not locked. Another process could be executing a read-modify-write on the same address resulting in corrupted data.

Add Command Format

Purpose Adds data to a Table Entry.

Command ID 0x7



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x7 for Add.
TBL#	60:57	**Table Number** — Identifies a table number. Legal range= 0 to 15.
Reserved	56:55	Read as zero.
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table entry for write. Legal range= 0 to 127.
DATA	47:32	**Data** — Data field for 8 or 16bit (single bus slot) writes. *MSK* field determines data alignment in SRAM.
MSK	31:24	**Byte Mask** — Byte mask for single slot writes (8Bytes).
Reserved	23:20	Read as zero.
IDX	19:0	**Index** — Designates a table entry in a given TBL#.

Field Name	Bit Position	Description
Reserved	63:32	Read as zero.
ADD DATA	31:0	**Additional Data** — Optional Data field for 32bit adds. *MSK* field bits [31:24] determines data alignment in SRAM.

Add Command Data Alignment Rules

- 8bit writes are placed in the *DATA* field bits [47:32] of the first slot and aligned to a byte boundary.
  - For masks of 0x01, 0x04, 0x10, and 0x40 data is stuffed into *DATA* field bits [39:32] of slot1.
  - For masks of 0x02, 0x08, 0x20, and 0x80 data is stuffed into *DATA* field bits [47:40].
- 16bit writes are placed in the *DATA* field bits [47:32] of the first slot.
- 32bit write data is stuffed into the *ADD DATA* field bits [31:0] of the optional register.

Add Command Returned Data

The *Add* command does *not* return any data.

Add Command Error Types

The *Add* command does *not* return any errors.

XOR Command

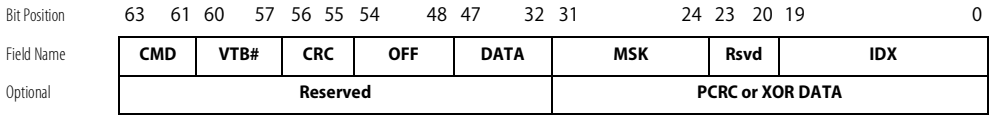
The *XOR* command behaves similarly to the *Add* command when *CRC* field bits [56:55] is set to 0x0. *XOR* supports 8, 16, and 32bit operands.

If the CRC field bits [56:55] are set to non-zero, 0x1, then the XOR command functions differently. Refer to "CRC Mode (Using the Non-zero XOR Command Options)" on page 272.

XOR Command Format

Purpose — Performs partial XOR operation on table data.
Note: The XOR command provides an alternative CRC Mode function using the available CRC field non-zero options.

Command ID — 0x1



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x1 for XOR operation.
VTB#	60:57	**Virtual Table Number** — Identifies a Virtual table number. Legal range= 0 to 15.
CRC	56:55	**CRC** — This entry is the CRC has detailed here: **Encoded Value** / **Function** 00 — XOR 01 — CRC (non-last) 10 — CRC Tx Last 11 — CRC Rx Last
OFF	54:48	**Offset** — Offset (in 8Byte increments) into table for write. Legal range= 0 to 127.
DATA	47:32	**Data** — The data to be XORed when using 1-Slot. Mask determines destination. The TLU always initializes this field to all zeros at the end of the CRC calculation.
MSK	31:24	**Byte Mask** — Byte mask for single slot writes (8Bytes)

Field Name	Bit Position	Description
Reserved	23:20	Read as zero.
IDX	19:0	**Index** — Designates a table entry in a given TBL#.
Reserved	63:32	Read as zero.
PCRC	31:0	**Partial CRC** — Generated by the SDP. This value is XORed with the value at *VTB#*, *IDX*, and *OFF*.
XOR DATA		**XOR Data** — Optional data for 32bit XORs.

XOR Command Data Alignment Rules

- 32bit write data is stuffed into bits [31:0] in the optional register, *PCRC* or *XOR DATA* field bits [31:0].
- 16bit writes are placed in the *DATA* field bits [47:32] of the first slot.
- 8bit writes are placed in the *DATA* field bits [47:32] of the first slot and aligned to a byte boundary.
  - So for masks of 0x01, 0x04, 0x10, and 0x40 data is stuffed into *DATA* field bits [39:32] of slot1.
  - For masks of 0x02, 0x08, 0x20, and 0x80 data is stuffed into *DATA* field bits [47:40].

XOR Command Returned Data

The XOR command does *not* return any data.

XOR Command Error Types

XOR command does *not* return any errors.

CRC Mode (Using the Non-zero XOR Command Options)

The CRC Mode also uses the same *XOR* command format. However, if the *CRC* field bits [56:55] are set to non-zero, 0x1, then the *XOR* command functions differently. When a non-zero value is selected three (3) CRC modes are available. Refer to Table 58 on page 272.

Table 58 Non-zero CRC Modes and Their Names

Encoded Value	Function Name
00	XOR
01	CRC (non-last)
10	CRC TX Last
11	CRC Rx Last

The TLU expects the target data to be in the format as described in "Partial CRC-32 Support" on page 310. Refer to "Partial CRC-32 Support" on page 310 and this section for a better understanding of the *XOR* command CRC Mode functions.

Table 59 Non-zero CRC Modes and Their Functions

Encoded Value/ Name	Function Details
01/CRC (Non-last)	If set to 01,then the *PCRC* optional field bits [31:0] is XORed with the data at *index* and *offset*. *CRC_Len* field bits [47:32] of the Partial CRC-32 Data Entry Format is incremented by one (1). Refer to "Partial CRC-32 Data Entry Format" on page 311.
10/CRC Tx Last	If set to 10, then *PCRC* field bits [31:0] is XORed as above except, the data is *not* written back to SRAM. Instead, a 12bit index is generated from the upper 12bits of *CRC_Len* field bits [47:32].
11/CRC Rx Last	If set to 11, then *PCRC* field bits [31:0] is XORed as above except, the data is *not* written back to SRAM. Instead, a 12bit index is generated from the upper 12bits of *CRC_Len* field bits [47:32].

After a CRC Mode has been executed and completed, either a CRC Tx Last (10) or CRC Rx Last (11). Their data is located in the DATA field bits [47:32] in the XOR command format, and are transferred to the CRC_Len field bits [47:32] in the Partial CRC-32 Data Entry Format. The DATA field [47:32] (16bits) should hold a zero (0) for CRC Tx Last (10), whereas, it should hold a one (1) for CRC Rx Last (11). The DATA field bits [47:32] value is transferred to the CRC_Len field bits [47:32] which is used to reset the cell counter, that is, to either a zero (0) or a one (1). Refer to "Partial CRC-32 Support" on page 310.

CRC Mode Flow

The generated 12bit index, resulting from either a CRC Tx Last (10) or CRC Rx Last (11), is used to access the *Partial CRC table*. The Partial CRC table is a 4K table that is used to convert from CRC-32 to FCS or from FCS to CRC-32, starting at the *CRC-32_FCS_Correction_Table_Base_Address* register. The data from this table is rotated and XORed up to 16 times depending on the bottom four (4) bits of the *CRC_Len* field bits [47:32] of the Partial CRC-32 Data Entry Format and finally XORed with the *PCRC* field bits [31:0].

For CRC Rx Last (11) only, t he final value is compared with *CRC 32_Checkvalue* register bits [31:0]. Next, the TLU returns the data in the CRC data structure with the Ring Bus error flag set to one (1) to indicate the status of the compare.

For both the CRC Tx Last (10) and CRC Rx Last (11), the TLU resets the *PCRC* field bits [31:0] to zero (0).

The *XOR DATA* field bits [47:32] are copied to the *CRC_Len* field bits [47:32] in the SRAM entry.

If CRC field bits [56:55] are set to non-zero, then MSK field bits [31:24] must be set to 0x0F.

CRC Mode Data Alignment Rules

The alternative XOR function does *not* have these rules.

CRC Mode Returned Data

- For CRC (non-last) (01), does *not* return any data.
- For CRC Tx Last (10), the returned value is the actual (Full) CRC calculation.
- For CRC Rx Last (11), the final CRC is calculated then compared with the *CRC-32 Checkvalue register*.

CRC Mode Error Types

- For CRC Rx Last (11) only:
  - If a match, then a response is returned. Note: The returned response data should be ignored.
  - If no match, then a response is returned, and the Ring Bus error bit is set to one (1).

Write Register Command

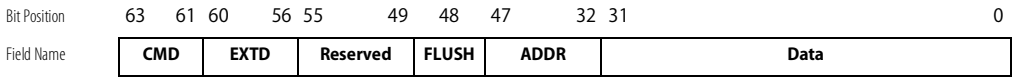
The *WriteReg* command writes *data* to the register at *index*.

WriteReg Command Format

Purpose Write data to a Register.

Command ID 0x0

Extended Command ID 0x10



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x0 for Write Register command.
EXTD	60:56	**Extended Command** — 0x10
Reserved	56:49	Read as zero.
FLUSH	48	**Flush** — If set during a register write, then the TLU stalls until the pipe is empty before updating the register. This bit MAY need to be set if switching virtual tables on the fly. Note: Using this bit, significantly slows down the TLU.
ADDR	47:32	**Address** — Address of register to write.
Data	31:0	Write data.

WriteReg Command Data Alignment Rules

All registers are 32 bits.

WriteReg Command Returned Data

The WriteReg command does *not* return any data.

WriteReg Command Error Types

The WriteReg command does *not* return any errors.

Read Register Command

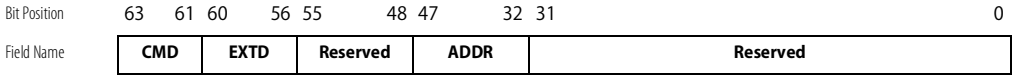
The *ReadReg* command reads a register at an *address*.

ReadReg Command Format

Purpose Reads data from a Register.

Command ID 0x0

Extended Command ID 0x11



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x0 for a Read Register command.
EXTD	60:56	**Extended Command** — 0x11
Reserved	55:48	Read as zero.
ADDR	47:32	**Address** — Address of register to read.
Reserved	23:20	Read as zero.
Reserved	31:0	Read as zero.

ReadReg Command Data Alignment Rules

All registers are 32 bits.

ReadReg Command Returned Data

The ReadReg command returns *data* and returns the *value* of the register in the lower 32bits of the Ring Bus returned data, while the upper 32bits are undefined.

ReadReg Command Error Types

The ReadReg command does *not* return any errors.

Echo Command

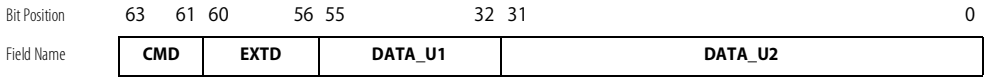
The *Echo* command "echoes" the input command and returns the input data to the output. The length of the returned data is always 8Bytes.

Echo Command Format

Purpose Copy the input command to the output.

Command ID 0x0

Extended Command ID 0x04



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x0 for the Echo command.
EXTD	60:56	**Extended Command** — 0x4
DATA_U1	55:32	**Data Upper 1** — Data to echo.
DATA_U2	31:0	**Data Upper 2** — Data to echo.

Echo Command Data Alignment Rules

The Echo command does *not* have these rules.

Echo Command Returned Data

The Echo command does *not* return any data.

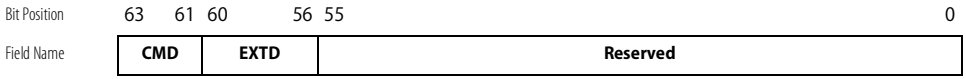
Echo Command Error Types

The Echo command does *not* return any errors.

No-Operation (NOP) Command

The *NOP* command inserts an empty slot into the TLU control pipe, causing the TLU to skip an SRAM access during that cycle.

Purpose	Insert an empty slot into the TLU pipeline.
Command ID	0x0
Extended Command ID	0x05



Field Name	Bit Position	Description
CMD	63:61	**Command** — Set to 0x0 for NOP.
EXTD	60:56	**Extended Command** — 0x05
Reserved	55:0	Read as zero.

Data Alignment Rules for NOP Commands

The NOP command does *not* have these rules.

Returned Data for NOP Commands

The NOP command does *not* return any data.

Error Types for NOP Commands

The NOP command does *not* return any errors.

TLU Configuration and Status Registers

TLU registers are 32bits wide, are accessed through the Ring Bus, and are addressed with a 16bit address. The TLU supports sixteen (16) table types and eight (8) lookup algorithms.

A *table* is a collection of entries that are managed by a user application. A table is defined by: type, number of entries, entry size and other parameters that are specified at table creation time. Tables are numbered from (0 to 15). Tables (0 to 7) can be: Data, VP Trie, Trie, Hash, Index Pointers or Key type tables, and tables (8 to 15) can *only* be Key or Data type tables.

An *algorithm* defines a *search order* for a lookup to the TLU. For example, an algorithm would instruct the TLU to start a particular search at table3 (Hash), then go to table5 (Trie, for collision resolution) and do an exact match key comparison and return the associated data at table12 (Key + Data). Lookup algorithms are numbered from (0 to 7). The lookup algorithm is a method for linking tables together and is defined using the *Lookup_Algorithm_Configuration1* and *Lookup_Algorithm_Configuration2* registers.

For TLU registers ≥ 0x100, the least significant byte defines the table number (*TBL#*) for the associated register. For example, to write the *Table _Configuration2* register for TBL#6, the register address would be 0x206.

TLU Registers

Nine (9) registers are used to set up the TLU's virtual tables. The registers are used for four (4) purposes: CRC-32 mode operation, collecting TLU statistics, configuration of the tables, and configuration of algorithms.

Table 60 TLU Registers

TLU Register Type	Register Function	Specific Register Details
CRC-32 Mode	Compares the final CRC-32 checksums.	See "CRC-32_Checkvalue Register" on page 279
	Contains base address of 2k Entry C0rrection Table.	See "CRC-32_FCS_Correction_Table _Base_Address Register" on page 280
Statistics	Records the minimum number of TLU FIFO slots after register reset.	See "TLU_Statistics Register" on page 280

Table 60 TLU Registers (continued)

TLU Register Type	Register Function	Specific Register Details
Table Configuration	Defines table type's, size and base address.	See "Table_Configuration1 Register" on page 281
	Defines the key length for physical tables (0 to 7).	See "Table_Configuration2_Lower Register" on page 283
	Defines the key length for physical tables (8 to 15).	See "Table_Configuration2_Upper Register" on page 284
	Maps a virtual table to a physical table.	See "Virtual_Table_Configuration Register" on page 284
Algorithm Configuration	Assigns virtual tables to an algorithm.	See "Lookup_Algorithm_Configuration1 Register" on page 285
	Assigns virtual tables to an algorithm for either a Trie or VP Trie table.	See "Lookup_Algorithm_Configuration2 Register" on page 287

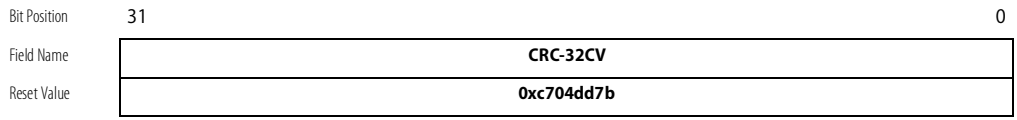
Each register is listed here along with its purpose, applicable fields, and parameters:

CRC-32_Checkvalue Register

This register compares the final CRC-32 checksums. It is used with the *XOR* command, CRC Mode function.

Purpose Used to compare the final Cyclic Redundancy Check (CRC) 32 checksums.

Address 0x0



Field Name	Bit Position	Description
CRC-32CV	31:0	**CRC-32 Check Value** — Used to compare the final CRC-32 checksums.

CRC-32_FCS_Correction_Table_Base_Address Register

This register contains the base address of the 2K Entry Correction Table used to convert from CRC-32 to FCS or from FCS to CRC-32. The 2K Entry Correction Table is used by the *XOR* command, CRC Mode function, to calculate a final CRC given a sequence of partial CRCs.

Purpose Base address for the 2K Correction Factor table used to convert between FCS and CRC-32.

Address 0x1

Bit Position	31 ... 16	15 ... 0
Field Name	**Reserved**	**CRC/FCS Base**
Reset Value	**raz**	**0x0**

Field Name	Bit Position	Description
Reserved	31:16	Read as zero.
CRC/FCS Base	15:0	**CRC/FCS Base Address** — This is the base address of the 2k Entry Correction Table used to convert from FCS to CRC or from CRC to FCS. The base address format is the same as for *Table_Configuration1* register.

TLU_Statistics Register

This register records the minimum number of TLU input FIFO slots after the register was reset. The input TLU FIFO is 68 slots deep (0x44). The register is reset to 0x43 at power up, or can be reset to an arbitrary value by writing to the register. If values >0x43 are written, the values are rounded down to 0x43.

Purpose Records the minimum number of TLU input FIFO slots after the register was reset.

Address 0x2

Bit Position	31 ... 8	7 ... 0
Field Name	**Rsvd**	**MINFIFO**
Reset Value	**0x0**	**0x43**

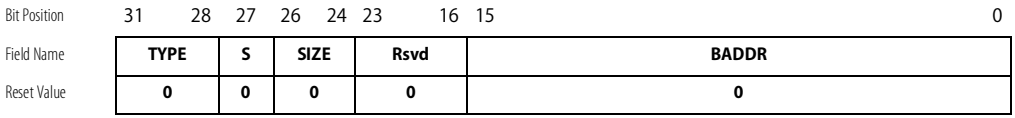
Field Name	Bit Position	Description
Reserved	31:8	Read as zero.
MINFIFO	7:0	**Minimum FIFO Slots** — Records the minimum number of TLU input FIFO slots after the register was reset.

Table_Configuration1 Register

This register defines the table's type, size and its base address. The TLU SRAM is 64bits wide and all addressing is on an 8Byte boundary. Therefore, a TLU SRAM address of 1 refers to byte 8, and an address of 8 refers to byte 64, and so on. The base address in *Table_Configuration1* register is the TLU SRAM address divided by 256. Thus, the next base address is: current base address + (table entry size/8) x number of entries) with the result rounded up to the next 2KByte boundary.

Purpose Defines the base address, size, and type of the table.

Address 0x100 - 0x10F



Bit Position	31 28	27	26 24	23 16	15 0
Field Name	TYPE	S	SIZE	Rsvd	BADDR
Reset Value	0	0	0	0	0

Field Name	Bit Position	Description
TYPE	31:28	**Table Type** — Defines table type. Legal range= 0 to 6. Refer to Table 61 on page 282.
S	27	**Increment Table Size** — Increases the Table Entry Size in VP Trie tables by one. Must=0 for all other table types.
SIZE	26:24	**Table Entry Size** — This field defines the size of an individual table entry. The entry size is $2^{(Table\_Entry\_Size\ +3)}$ + Size bytes. The minimum entry is 8Bytes and the maximum is 1024Bytes. Refer to Table 62 on page 282. Note: VP Tries are a special case of the *Size* field. The encoding of S=1 sets the number of entries per index to 3 (3 x 8 = 24Bytes).
Reserved	23:16	Read as zero.
BADDR	15:0	**Base Address** — Defines the base offset (index) of a table. The base offset is defined as (BADDR x 8) x 256Bytes. The upper bounds of the table are not defined.

Table 61 Available TLU Table Types

Table Type	Name	Description
0	Data	Generic Data Table.
1	VP Trie	Variable Prefix Trie. The key is compared with every node down a branch. The final leaf node then points directly to a data table.
2	Trie	Trie with "Patricia" Tree style skip fields. The leaf nodes of this table typically point to a key table, so that the key can be matched.
3	Hash	The search key is hashed to generate an index into this table. A node in this table either points to another data table or it points to another table if there was a collision.
4	Index Pointers	A table of indices that are used to link to a new table.
5	Key	A Key table is a Data table where the first and second entries are a key for the associated data. A search key is compared against the key in the table. If they match then the TLU returns the index (or data) associated with this key. If they don't match, then a 0 is returned.
6	External	Used to interface with external lookup engines when it is swapped with a SRAM bank.

Table 62 Legal Values for the Table Entry Size

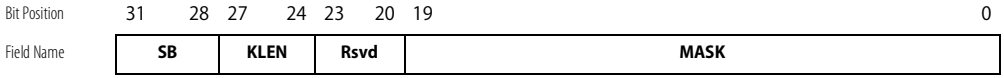
Table Type	Table Entry Size (Bits)			
	112	96	48	32
Data	Unrestricted	Unrestricted	Unrestricted	Unrestricted
VP Trie	1 (S=1)	1 (S=1)	1 (S=1)	1 (S=1)
Trie	0	0	0	0
Hash	0	0	0	0
Index Pointers	0	0	0	0
Key	≥0	≥0	≥0	≥0

Table_Configuration2_Lower Register

This register defines the key length of physical tables (0 to 7), (*TBL#*), which comprises: Data, VP Trie, Trie, Hash, Index Pointers or Key type tables.

Purpose Defines the key length for physical tables 0 to 7.

Address 0x200 - 0x207



Field Name	Bit Position	Description
SB	31:28	**Start Byte** — Byte to start the mask. If the Key Length = 32bits, then start must be 2 to 5. Note: The implementation of the Start Byte field is based on the table type. Refer to "Start Byte Field Usage Based on Table Type" on page 283.
KLEN	27:24	**Key Length** — Defines the length of the key as detailed here: Note: These bits [27:24], are *only* valid for registers 0x200 to 0x207. Intermediate key sizes are supported by masking unused bits to zero.
Reserved	23:20	Read as zero.
MASK	19:0	**Mask** — Masks off the initial index from a hash or a lookup key.

Encoded Value	Key Size (bits)
0x4	32
0xC	48
0x7	96
0xF	112

Start Byte Field Usage Based on Table Type

For an Index table lookup, the *Start Byte* field bits [31:28] tells the TLU where to apply the mask. This is shown in "Index Pointer Data Entry Format" on page 289, and in Table 65 on page 289.

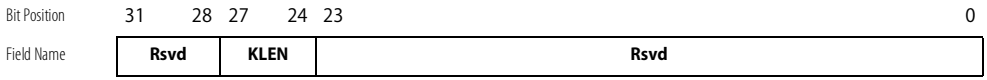
- For Hash and Trie tables, the *Start Byte* field bits [31:28] is *not* referenced. Instead the start byte/bit is calculated using the *count* field in each table's configuration register.
- For Data, Key, and VP Trie tables, the *Start Byte* field bits [31:28] is *not* used and the value of this field should always be set to zero (0).

Table_Configuration2_Upper Register

This register defines the key length of physical tables (8 to 15), (*TBL#*), which can *only* be Key or Data type tables.

Purpose Defines the key length for physical tables 8 to 15.

Address 0x208 - 0x20F



Field Name	Bit Position	Description
Reserved	31:28	Read as zero.
KLEN	27:24	**Key Length** — Defines the length of the key as detailed here: Note: Intermediate key sizes are supported by masking unused bits to zero.
Reserved	23:0	Read as zero.

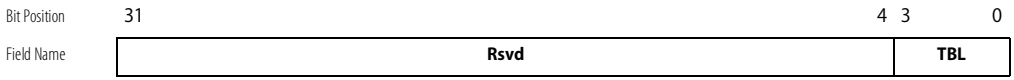
Encoded Value	Key Size (bits)
0x4	32
0xC	48
0x7	96
0xF	112

Virtual_Table_Configuration Register

This register maps a virtual table (*VTB#*) to a physical table (*TBL#)*. All TLU commands use a virtual table number (*VTB#*) as an argument. The default reset value is the least significant three (3) bits of the table address, so the default value of the register at address 0x306 is 0x6. All table references are through a virtual table (*VTB#*).

Purpose Maps a virtual table to a physical table.

Address 0x300 - 0x30F



Field Name	Bit Position	Description
Reserved	31:4	Read as zero.
TBL	3:0	**Table Number** — Physical table number.

Lookup_Algorithm_Configuration1 Register

This register assigns virtual tables (*VTB#*) to an algorithm. It defines the algorithm to use for the three (3) types of find commands (*Find*, *Findw*, and *Findr*). The data structures have fields in them that control branching between various tables. The TLU has the capability to link together up to three (3) different lookup (or "search") algorithms. The first table is specified using *LNK1* field bits [31:28], the second using *LNK2* field bits [26:24], and the third using *DATA* field [15:12]. The virtual table's data is specified with the *DATA* field bits [15:12].

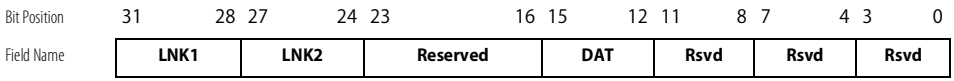
Up to eight (8) lookup algorithms may be set up to perform linked-table operations. The legal range= 0 to 7, (0x400 to 0x407).

For example, to program a hash function, specify *VTB#2* as a Hash table, and set *LNK1* to 2. Assuming collision branching to a Trie table, specify *VTB#3* as a Trie table, and set *LNK2* to 3. The associated data for the table would actually be defined as a Key table type (because you want to do a full compare on the Key after the Hash). The Key table could then be set as *VTB#6*, so the *DATA* field bits [15:12] would be set to 6.

Indexed accesses do not need to set this register.

Purpose Assign virtual tables to an algorithm.

Address 0x400 - 0x407



Field Name	Bit Position	Description
LNK1	31:28	**Link Table 1** — Start lookup at this virtual table.
LNK2	27:24	**Link Table 2** — Identifies the second lookup virtual table.
Reserved	23:16	Read as zero.
DATA	15:12	**Data** — Associated data is in this algorithm.
Reserved	11:4	Read as zero.
Reserved	3:0	Read as zero.

The supported table linkage combinations are listed in Table 63 on page 286, in which the Associated Data column lists the type of Data table to which the linkage points.

All other linkage combinations are illegal, and only Key and Data tables can be used as the "data" table in a linkage.

Table 63 Legal LNK1, LNK2, and Data Types for *Lookup_Algorithm_Configuration1* Register

Legal LNK1 Types	Legal LNK2 Types	Legal Associated Data Types
Index	Trie	Key
Index	VP Trie	Data
Hash	Trie	Key
Trie	No table	Key
VP Trie	No table	Data

Up to eight (8) algorithms can be configured using up to the three (3) fields (*LNK1*, *LNK2*, and *DATA*) contained within this register to perform various lookups. A restriction is that the last table in a linked-table configuration (algorithm) must always be a Key or Data type table. Additionally, you can have just two (2) tables configured in a linked-table algorithm. Refer to Table 64 on page 286.

Table 64 Algorithm Configuration Examples

Algorithm #	Lookup_Algorithm_Configuration1 Register		
ALG#0	LNK1[31:28]= VTB7=TBL0=Hash	LNK2[31:28]= VTB5=TBL2=Trie	DATA[15:12]= VTB6=TBL1=Key
ALG#1	LNK1[31:28]= VTB4=TBL0=VP Trie	LNK2[31:28]= VTB5=TBL2=Index	DATA[15:12]= VTB6=TBL1=Key
ALG#2	LNK1[31:28]= VTB5=TBL2=Index	LNK2[31:28]= Not Used	DATA[15:12]= VTB3=TBL1=Data
ALG#3	LNK1[31:28]= Not Used	LNK2[31:28]= VTB2=TBL2=Trie	DATA[15:12]= VTB3=TBL1=Data
• • •	• • •	• • •	• • •
ALG#7	LNK1[31:28]= VTB3=TBL0=Hash	LNK2[31:28]= VTB2=TBL4=Index	DATA[15:12]= VTB3=TBL1=Data

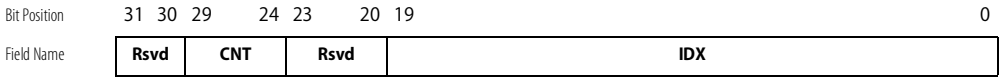
Table 64 *on page 286 is only an example, virtual tables and physical tables can be mapped in many different ways.*

Lookup_Algorithm_Configuration2 Register

This register assigns virtual tables (*VBT#*) to an algorithm *only* when *LNK1* field bits [31:28] in *Lookup_Algorithm_Configuration1* register is either a Trie table or VP Trie table. This register contains the root index for VP Trie and Trie tables.

Purpose Assign virtual tables to an algorithm, only when the VP Trie or Trie table is used in *LNK1* field of the *Lookup_Algorithm_Configuration1* register.

Address 0x500 - 0x507



Field Name	Bit Position	Description
Reserved	31:30	Read as zero.
CNT	30:24	**Skip Count** — Initial skip count for “Patricia” style tries.
Reserved	23:20	Read as zero.
IDX	19:0	**Index** — Initial index for all VP Tries.

TLU Format and Examples of Table Types

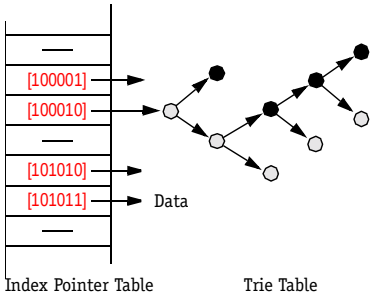
This section describes the proper data entry format and gives an example of each of the seven (7) supported table types. These examples are provided to show the difference in data structure between each table type.

Indexed Pointer Tables

Indexed pointer tables contain entries that point to an entry (via an index) in another table. Refer to Figure 58 on page 288. This type of table can be used to evaluate a portion of a lookup key. For instance, you can use an Indexed Pointer table to evaluate the first 8 or 16bits of a 32bit key and use a another table to lookup the rest of the key.

A practical example would be using a 16bit indexed pointer lookup on the first 32bits of a variable-prefix IP lookup, and then using a VP Trie table to evaluate the remaining 16bits of the address.

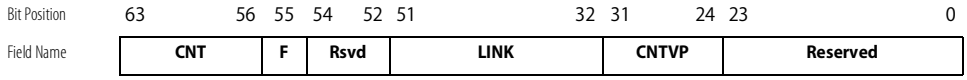
Figure 58 Indexed Pointer Table Data Structure



CNT [0]	F [1]	Rsvd	LINK [100001]	Rsvd
CNT [4]	F [1]	Rsvd	LINK [100010]	Rsvd
CNT [0]	F [0]	Rsvd	LINK [0]	Rsvd
CNT [0]	F [1]	Rsvd	LINK [101010]	Rsvd
CNT [0]	F [0]	Rsvd	LINK [101011]	Rsvd

Index Pointer Table Entries

Index Pointer Data Entry Format



Field Name	Bit Position	Description
CNT	63:56	**Count** — The total number of bits that match traversing down the branch.
F	55	**Chain Flag** — A 1 indicates that this node points to a new table in which to resume the search. A 0 indicates that this node points to the associated data.
Reserved	54:52	Read as zero.
LINK	51:32	**Link** — Index of associated data if F = 0 or index to an element in the next table in the algorithm if F = 1.
CNTVP	31:24	**Count VP** — The number of bits that match in the parent of the node pointed to by the LINK field. Note: This field is only required when the new table is a VP table, otherwise this field is reserved.
Reserved	23:0	Read as zero.

Figure 64 on page 301 provides an examples of how to set up the *Table_Configuration2* register for 8 and 16bit index pointer tables.

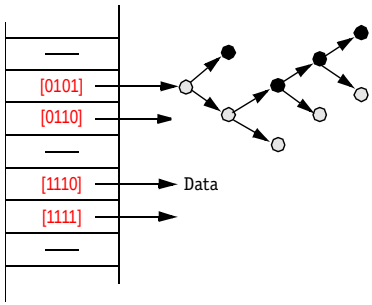
Table 65 Table_Configuration2 Register Setup Examples

Total Key Size (bits)	Index Size (bits)	Start Byte	Mask
48	8	0	0x00FF
48	16	1	0xFFFF
32	8	2	0x00FF
32	16	3	0xFFFF

Hash Tables

Hash Tables are used for *exact match* algorithms. That is, algorithms that require a lookup key to be matched exactly. Hash tables evaluate the lookup key via a *hash function*. The hash function returns an index into the Hash table that contains an entry that points to another entry in another table. The table entry *pointed to* by the Hash table can be either an entry in a Trie table (used for collision resolution), a data entry in a data table, or a key to be used for comparison in a key table.

Figure 59 Hash Table Data Structure



CNT [0]	FL [1]	Rsvd	LINK [0101]	Rsvd
CNT [0]	FL [1]	Rsvd	LINK [0110]	Rsvd
CNT [0]	FL [0]	Rsvd	LINK [0]	Rsvd
CNT [0]	FL [0]	Rsvd	LINK [1110]	Rsvd
CNT [0]	FL [1]	Rsvd	LINK [1111]	Rsvd

Hash Table Entries

Calculating Collisions

You can calculate the number of collisions using a Chi^2 distribution. Thus, the probability of a collision based on random data is:

```
# real entries/# buckets
```

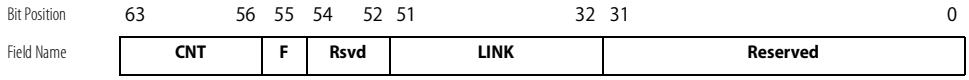
For example, if you have 64k real 32bit keys and hash to a 18bit index, your hash table would have 256k buckets. Dividing the 64k entries by 256k indicates that there would be an average of one (1) collision for every four (4) entries. At most one (1) bucket would have eight (8) entries, and all of the other buckets would have something less than eight (8).

TLU Hash Function

The TLU hash function is a fixed function that produces a hash index of the lookup key. Typically, the hash function exhibits good randomness such that changing one (1) bit in a key causes approximately half of the bits (in the hash index) to change as a result.

An example of a Hash table is an 802.1D Bridge Forwarding Table. Collisions can be resolved by having entries that collide point to a Trie table. Ultimately, the entry key and associated data are stored in a data table.

Hash Data Entry Format



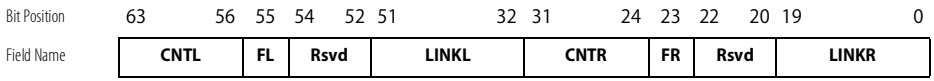
Field Name	Bit Position	Description
CNT	63:56	**Count** — The total number of bits that match traversing down the branch.
F	55	**Chain Flag** — A 1 indicates that this node points to a new table in which to resume the search. A 0 indicates that this node points to the associated data.
Reserved	54:52	Read as zero.
LINK	51:32	**Link** — Index of associated data of F = 0, or index to an element in the next table in the algorithm if F = 1.
Reserved	31:0	Read as zero.

Trie Tables

Trie tables are binary trees that provide an efficient means for resolving Hash table collisions. A benefit of using Trie tables for collision resolution in conjunction with a Hash table is that for *n* keys that collide in the table, only $\log_2(n)$ entries need to be checked to resolve the collisions. For collision resolution, only the bits that differ in the collided keys are checked. Leaf nodes point to a Key table entry that does a complete key match of the input key. The size of each node of this table structure is 8Bytes or one (1) SRAM location.

The TLU uses a modified "Patricia" tree to store Trie tables.

Trie Data Entry Format



Field Name	Bit Position	Description
CNTL	63:56	**Count Left** — Specifies the position of the bit in the key that is used to determine whether to take the left or right branch in the next left branch node. If the CNTL value is a non-zero, then the CNTL value refers to an individual bit in the Key, and if that bit value is 0 take left branch and if the bit is 1 take right branch. If the CNTL value is 0x00, then this field indicates that the next node is a leaf node.
FL	55	**Chain Flag Left** — If 0, continue to next "left" node of this table if the associated count field is non-zero, or to the Key table entry if the associated count field is 0. If 1, this node points to the next table in the algorithm linked-table chain.
Reserved	54:52	Read as zero.
LINKL	51:32	**Link Left** — Index to the left branch of the Trie if FL = 0, or to the next table if FL = 1.
CNTR	31:24	**Count Right** — Specifies the position of the bit in the key that is used to determine whether to take the left or right branch in the next right branch node. If the CNTL value is a non-zero, then the CNTL value refers to an individual bit in the Key, and if that bit value is 0 take left branch and if the bit is 1 take right branch. If the CNTL value is 0x00, then this field indicates that the next node is a leaf node.
FR	23	**Chain Flag Right** — If 0, continue to next "right" node of this table if the associated count field is non-zero, or to the Key table entry if the associated count field is 0. If 1, this node points to the next table in the algorithm linked-table chain.

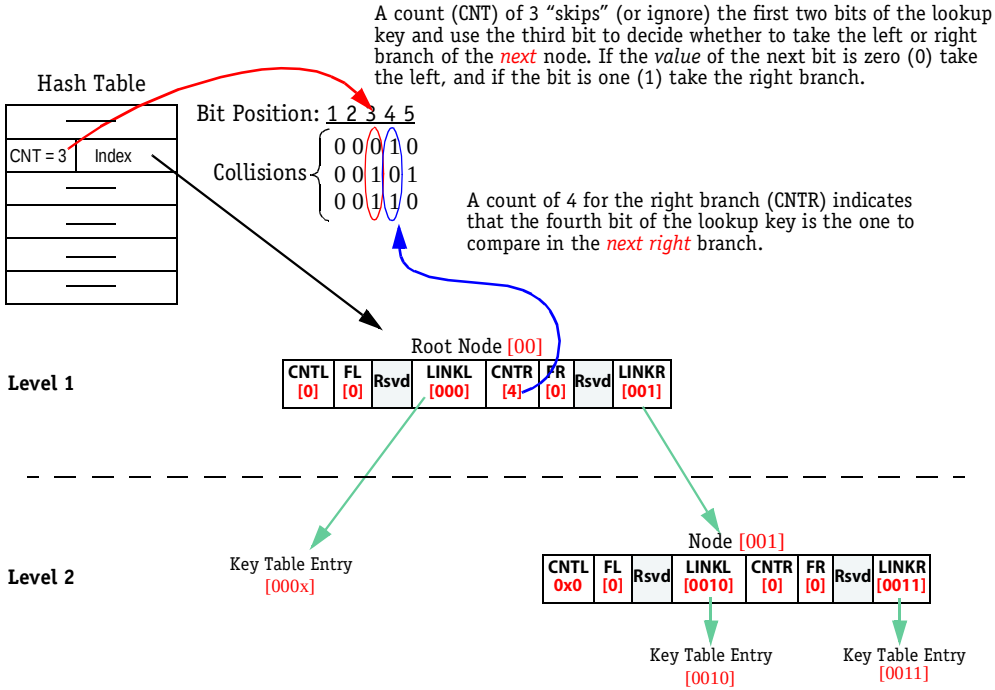
Field Name	Bit Position	Description
Reserved	22:20	Read as zero.
LINKR	19:0	**Link Right** — Index to the right branch of the Trie if FL = 0, or the next table if FR = 1.

Trie tables evaluate the lookup key one bit at a time. As the table is traversed, the count fields in the previous node are used to specify which bit of the key is to be evaluated in the current node. The value (0 or 1) of the bit being evaluated indicates whether to take the left branch or the right branch (0=left branch, 1=right branch) of the tree.

The use of the *count* field is shown in Figure 60 on page 294. The *3* in the Hash table's CNT field means that the third MSB of the three collision entries is to be used when evaluating the lookup key in the next "node". The *index* value of the entry in the Hash table points the next entry. In this case, the index points to the "root" node in a new table (a Trie table) to be used for collision resolution. Examining the third MSB of the collided values shows that one (1) entry has a bit value of (0) and the other two (2) entries have bit values of (1). The (0) value indicates to take the left branch from the root node and the (1) value indicates to take the right branch from the root node.

Since there is only one (1) entry with the third MSB equal to zero (0), the next left branch node (level 2) in the Trie table can point to the Key table entry that contains the data.

Figure 60 Trie Table Showing Skip Function



However, there are two (2) "collision" entries with a value of (1) for their third bit (00101 and 00110). Thus, the program must move to a lower level in the tree to obtain the correct indexes (to Key table entries) for each lookup key. Notice that the root node's *CNTR* (count right) field is set to four (4). This means that in the next right node (level 2), the fourth bit of the lookup key should be evaluated. Again, a bit value of (0) indicates branch left and a value of (1) indicates branch right, (0=left branch, 1=right branch).

Since there are only two (2) remaining collision entries and their fourth bits are different (one is a 0 and the other is a 1), both the left and right nodes will match the first four bits of one of the two unresolved entries and since the CNT is zero (0), each link points to the correct index into the associated Key table.

The exact match on the lookup key takes place in the Key table. The Trie is used to resolve search key values that collided in the Hash table. Thus, the depth of the Trie table needs to be only deep enough to distinguish two (2) or more keys (depending on the number of collisions) from each other.

The *CNTL* (count left) and *CNTR* fields also allow you to "ignore or skip" one (1) or more bits in a lookup key and thus omit what would be the corresponding nodes on a branch of the tree. Thus, if the values for a number of bits in the Trie entries are the same across a number of entries, the TLU can perform an "exact" match without traversing a level of the Trie for every bit in the lookup key.

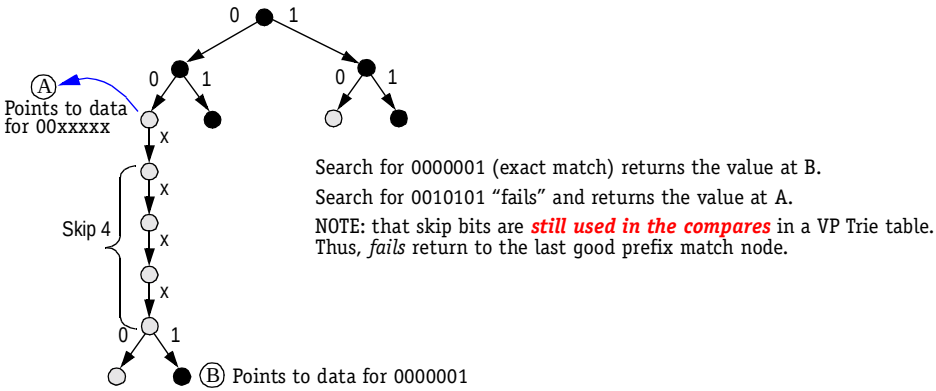
Variable Prefix (VP) Trie Tables

VP Trie tables are used for *most-specific match* algorithms. An entry in a VP Trie table contains sets of pointers (*LINKL*, *LINKR*, and *LINKD*) and partial key information (*VMASK* and *PKEY*) that allows a TLU lookup to compare a table entry with a key and return the most-specific match to the lookup key in the table. The last entry (or, in a branch, each entry) in the table points to the Data table entry where the associated data is stored.

A VP Trie table is typically used in conjunction with an Indexed Pointer table. In this case, the Indexed Pointer table evaluates either the first 8 or 16bits of a lookup key and the VP Trie table evaluates the rest of the key. This is a good method for a most-specific match of a 32bit key value like an IP destination address in the IP routing table.

VP Trie tables can also "skip or ignore" one or more bits in a lookup key in the same way as Trie tables. But an important difference is their comparison algorithm. Each leaf of a VP Trie table contains the partial key (PKEY) to that point. If the partial key compare fails, the previous node in the table that was the "best partial match" is returned. Refer to Figure 61 on page 296.

Figure 61 VP Trie Table Data Structure Showing Skip Function



Variable Prefix (VP) Tries Data Entry Format

Bit Position	63–56	55	54	53–52	51–32	31–24	23–16	15–0
Field Name	CNTL	Rsvd	C	Rsvd	LINKL	Reserved	VMASK	PKEY
Field Name	Reserved	Rsvd	C	Rsvd	LINKR	CNTR	VMASK	PKEY
Field Name	Reserved				LINKD	Reserved	VMASK	PKEY

Field Name	Bit Position	Description
CNTL	63:56	**Count Left** — Specifies the position of the bit in the key that is used to determine whether to take the left or right branch in the next left branch node. If the CNTL value is a non-zero, then the CNTL value refers to an individual bit in the Key, and if that bit value is 0 take left branch and if the bit is 1 take right branch. If the CNTL value is 0x00, then this field indicates that the next node is a leaf node.
Reserved	55	Read as zero.
C	54	**Compare Node** — If this is set to 0, then the LINKD field in this entry points to an entry in the associated data table in the algorithm linked-table chain. If this field is set to 1, then do not use the associated data linked-table (this is a dummy node).
Reserved	53:52	Read as zero.
LINKL	51:32	**Link Left** — Index to the left branch of the VP Trie.
CNTR	31:24	**Count Right** — Specifies the position of the bit in the key that is used to determine whether to take the left or right branch in the next right branch node. If the CNTL value is a non-zero, then the CNTL value refers to an individual bit in the Key, and if that bit value is 0 take left branch and if the bit is 1 take right branch. If the CNTL value is 0x00, then this field indicates that the next node is a leaf node.
VMASK	23:16	**Variable Mask** — The number of valid bits in this partial key (PKEY).
PKEY	15:0	**Partial Key** — The 16bit word that contains the bit indicated by the count field (CNTL or CNTR) this nodes parent.
LINKR	51:32	**Link Right** — Index to the right branch of the VP Trie.
LINKD	51:32	**Link Data** — Points to data associated with this node.

Each VP Trie table entry is 24Bytes long and occupies three (3) consecutive 8Byte SRAM *offset* locations.

A node's count field (CNTL or CNTR) generates the offset into the next data structure, and that it is the previous node's count field (CNTL or CNTR) that generates the offset into the current data structure.

When traversing through a node, the TLU compares *PKEY* with the corresponding bits of the search key as indicated by the *VMASK* field. If the *PKEY* compare passes, the node replaces the last "best match" node and then moves to the next node in the table. If the PKEY compare fails, then the last *"best match"* node is passed to the calling function.

Figure 62 VP Trie Table Example 1: Root Node Pointed to by Alg2 Register

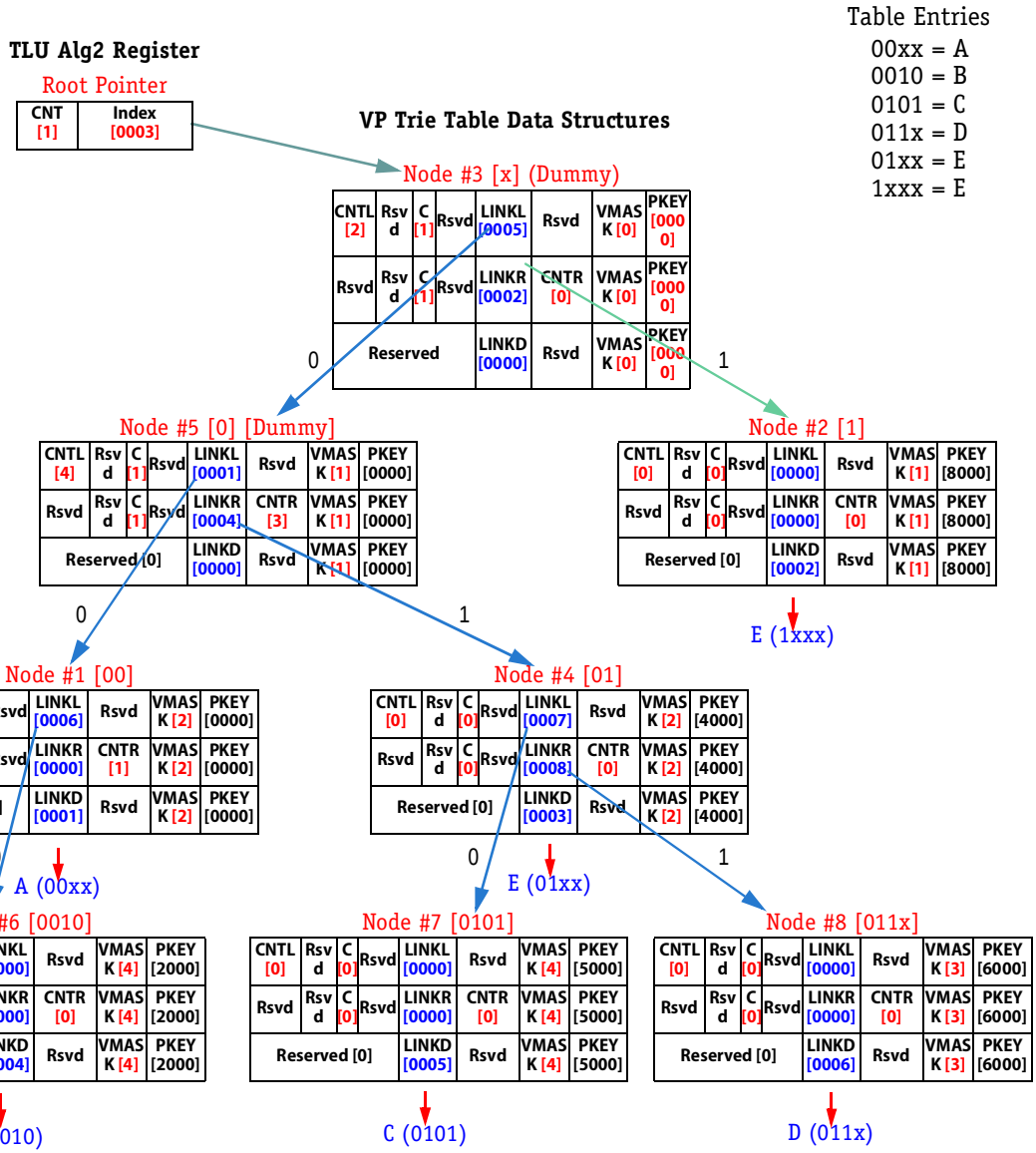
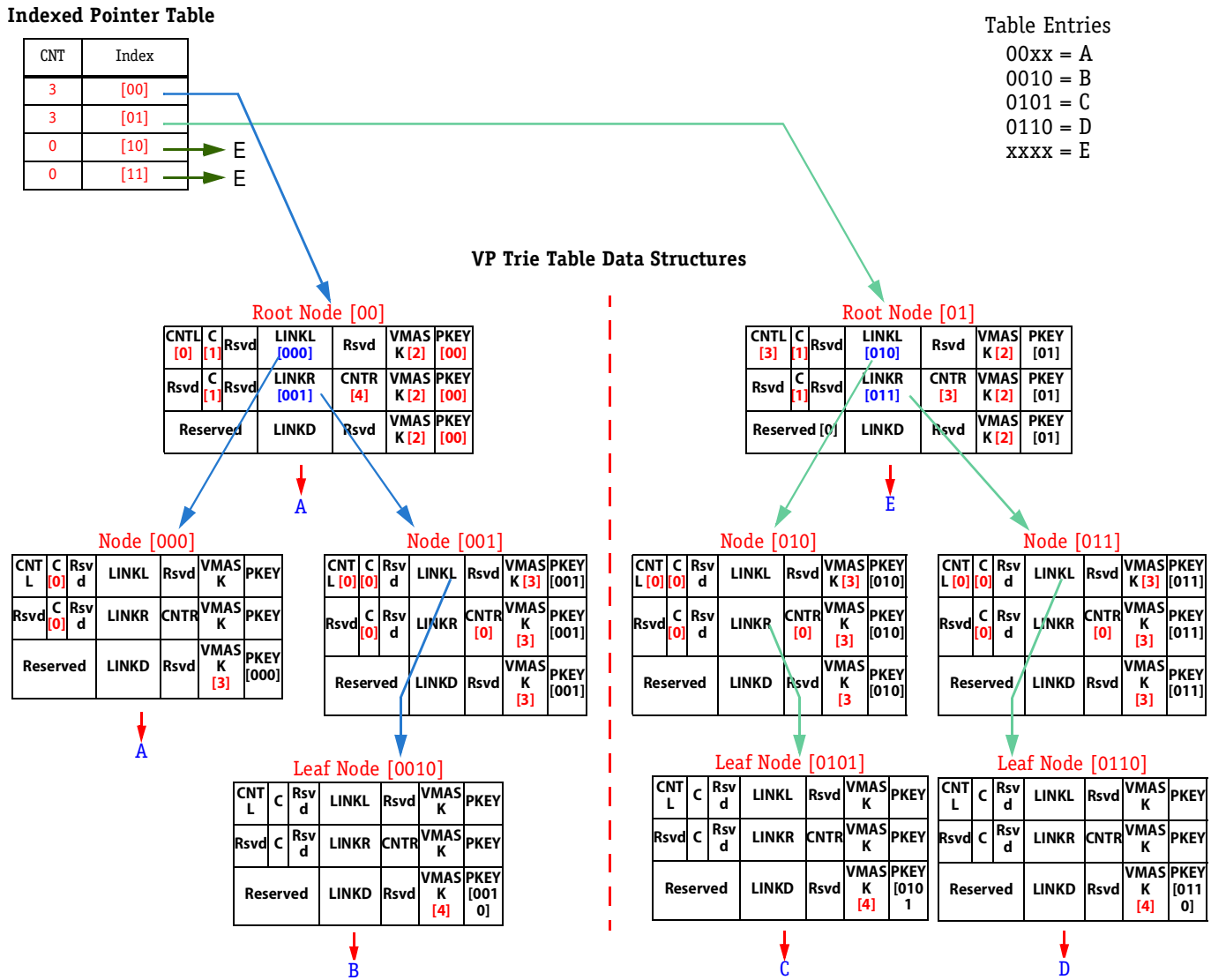


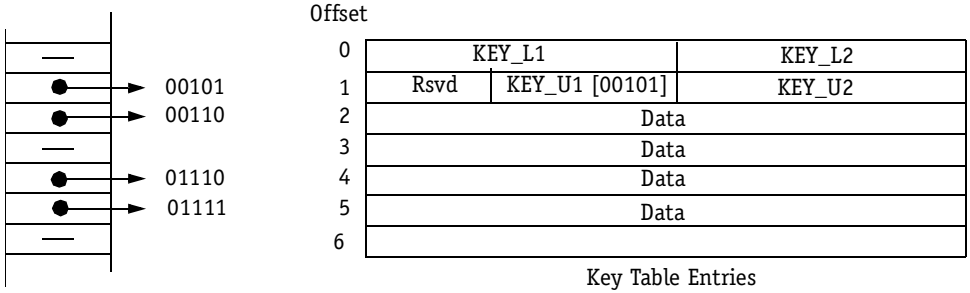
Figure 63 VP Trie Table Example 2: Root Node Pointed to by Indexed Pointer Table Entry



Key Tables

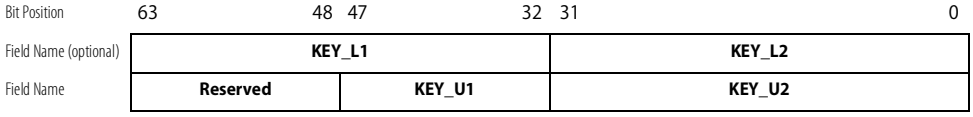
Key tables are used in *exact match* algorithm, and contain both the key of an entry and the associated data. The last step of an exact match algorithm compares the lookup key with the key from the entry in the Key table. The TLU supports key sizes of 32, 48, 96, and 112bits. The associated data portion of the entry in a Key table is typically returned to the requesting node (CP or XP) via the Ring Bus.

Figure 64 Key Table Data Structure



An example of an exact match algorithm that would be implemented using the TLU is an 802.1D forwarding table for layer 2 bridging applications. In this application, you would use a Hash table for the initial index evaluation, a Trie table to resolve collisions (if necessary), and a Key table to store both the key and associated data.

Key Data Entry Format



Field Name	Bit Position	Description
KEY_L1*	63:32	**Key Lower 1** — Contains the lower middle portion for 112 and 96bit keys
KEY_L2*	31:0	**Key Lower 2** — Contains the LSB for 112 and 96bit keys.
Reserved	63:52	Read as zero.
KEY_U1*	51:32	**Key Upper 1** — The MSB of 48bit and 112bit keys.
KEY_U2*	31:0	**Key Upper 2** — The MSB of 32bit and 96bit keys. Used also for the upper middle portion of 112 and 48bit keys.

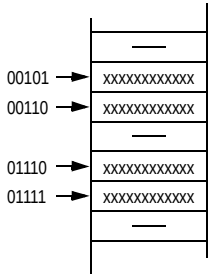
* For key sizes up to 48 bits, only one entry is used (KEY_U1 & KEY_U2). For larger key sizes, both fields are used.

A Key Data structure occupies the first one (1) or two (2) entries of a data entry. It is used by the TLU to verify that a node's data matches the search key. It is typically the termination of Trie and Hash tables. For key sizes up to 48bits, only one (1) entry is used. For larger key sizes, both fields are used.

Data Tables

Data tables contain the data associated with an entry. Data tables can be used as a stand-alone table or as the last table (that contains the associated data for a previously evaluated key) in a set of linked tables.

Figure 65 Data Table Data Structure



A common example of a stand-alone Data table would be the ATM VC table. Data can be read and written to this table by an index. Typically the concatenated VPI/VCI makes up the index. Another more specific example would be a implementing Partial CRC-32's for ATM Adaptation Layer-type 5 (AAL-5) reassembly that are stored in a Data table type structure. Refer to "Partial CRC-32 Support" on page 310.

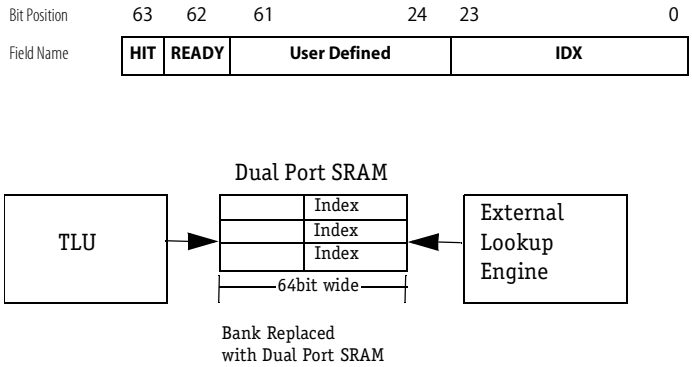
External Tables

The SRAM interface may alternatively be used to communicate with third party lookup engines. One or more SRAM banks are replaced by dual ported memories accessible to both the external lookup engine and the TLU. This memory serves as go-between for the two (2) devices. An external lookup table is constructed in at least one (1) of the shared memories. The table contains one (1) or more entries. Each entry must comply with the "external table interface format". These entries serve as mailboxes to synchronize data transfer between the two (2) devices.

When a *Find* or *Findr* lookup command is used to access the external table, the least significant 24 bits of the Key (32 or 48 bit keys only) are used to index the appropriate entry. The TLU then proceeds to continuously poll the entry until the *READY* field bit [62] is set by the external lookup engine. The TLU then checks the contents of the *HIT* field bit [63], if set to one (1), the TLU returns the appropriate lookup data, if set to zero (0), a lookup miss error (type 1) is returned.

The TLU polls for the *READY* field bit [62] assertion for up to 255 times before quitting and returning a watchdog time error (type 2).

Figure 66 External Table Interface Format



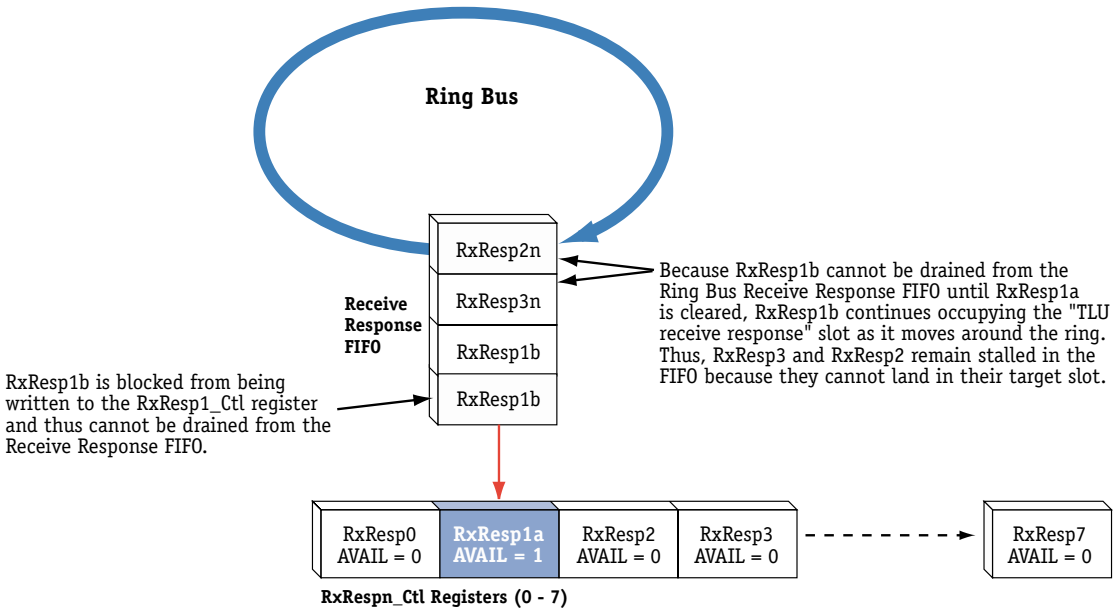
TLU Application Considerations

The following section covers issues that are important to application design. For more information about how implement tables in applications see the *C-Ware Reference Library* document in the *C-Ware Application Development Guide*.

TLU/Ring Bus Control Register Response Slot Usage

TLU lookup results are returned via the Ring Bus and put into the *RxRespn_Ctl* register that was specified during the Ring Bus launch. If a TLU response slot is currently "occupied" with a previous response (Resp1) and a second lookup result (Resp2) destined for the same slot is ready, the second response (Resp2) is placed in the eight (8) slot receive response FIFO and remains in the FIFO until the first response is released and response slot is cleared. A response slot is released by deasserting the *AVAIL* bit [31] in the Ring Bus' *RxRespn_Ctl* register. Refer to Figure 67 on page 304.

Figure 67 TLU/Ring Bus Control Register Response Slot Usage



Responses move into the receive response FIFO on the destined processor (CP or XP). If the response at the head of the FIFO can't get into a response register slot because it's occupied by another response, then it is "trapped" at the head of the FIFO, causing head-of-line blocking. When the entire response FIFO is full, then other responses destined for this CP start circling the Ring Bus. Refer to "Ring Bus Overview" on page 370.

TLU Performance

Table 66 on page 305 describes the TLU latency and the number of SRAM cycles for various operations. All performance estimates are relative to the SRAM clock. For 100MHz SRAMs a clock cycle is 10ns. Performance estimates are based on a ZBT SRAM. The TLU operation frequency is assumed to be synchronous with the SRAM clock frequency. Synchronization occurs using the input and output FIFOs.

Table 66 TLU Performance Estimates

TLU Command	SRAM Cycles	TLU Latency
Write()	Roundup (# bytes written/8)	FIFO delay + 5 + SRAM cycles * 4
Read()	Roundup (# bytes read/8)	FIFO delay + 5 + SRAM cycles * 4
Add() or XOR()	2	FIFO delay + 5 + SRAM cycles * 4
Find()	Refer to Table 67 on page 305.	FIFO delay + 5 + SRAM cycles * 4
Findw()	Find + Roundup (# bytes written/8 +1)	FIFO delay + 5 + SRAM cycles * 4
Findr()	Find + Roundup (# bytes read/8 +1)	FIFO delay + 5 + SRAM cycles * 4

The FIFO delay is dependent on the FIFO backlog. The backlog is a function of the input command rate versus the SRAM access rate. If every CP is limited to four (4) lookups at one time, then the maximum FIFO delay is 16 * 4 = 64 clocks. When the fabric interface is added it increases the maximum delay to about 80 clocks.

The number of SRAM cycles to issue a *Find* type command is highly dependent on the algorithm being used and upon the table entries themselves. Estimated values are given in Table 67 on page 305. For the Trie structures, assume an equally balanced tree with 64k nodes. These latencies do *not* include round trip time on the Ring Bus (18-90) clock cycles. For a Hash structure with ((*n* real entries < 2) x *n* buckets), the number of collisions is almost always less than eight (8).

Table 67 TLU SRAM Accesses by Table Format

	SRAM Cycles			TLU Latency (ns)		
Table Type	Min.	Typical	Max.	Min.	Typical	Max.
Hash with no collisions	1	1	1	20	20	20
Hash with n collisions	2	$2 + \log_2 n$	2 + n	30	$30 + 10^* \log_2 n$	30 + 10* n
VP Trie	2	20	32	30	220	340
VP Trie using 16bit index	2	10	18	30	120	200
Trie	2	10	18	30	120	200

Table Sizing Examples

To aid in sizing tables, examples using two (2) typical applications are described in this section. Both examples are for a Layer 2/3 switch and list the required tables types, number of entries, sizes of entry, and sizes for the tables.

As the numbers indicate in the sizing examples shown in this section, an application implementing both a Bridge Address table and an IP Routing table uses about 8.5M of SRAM space. Refer to Table 68 on page 306, and Table 69 on page 306.

These examples do not include any other types of statistics or QoS tables, nor do they count an external TLU device out one bank.

Bridge Address Table Sizing Example

The Bridge Address table consists of an Index table to map VIDs to FIDs for 802.1Q, a Hash table, a Trie table, and a Key table for the associated data. A common size is anywhere from 64k to 256k entries. This example uses a 128k entry table.

Table 68 Bridge Address Table Sizing Example

Table Type	Number of Entries	Entry Size	Table Size
Index	4k	8	32k
Hash	256k	8	2.1M
Trie	32k	8	262k
Key	128k	32	4.1M

IP Routing Table Sizing Example

The IP Routing table consists of an Index, a VP Trie, and a Data table. A common size for an IP routing table is between 32k and 256k. This example uses 64k.

Table 69 IP Routing Table Sizing Example

Table Type	Number of Entries	Entry Size	Table Size
Index	64k	8	524k
VP Trie	32k	24	786k
Data	64k	16	1.0M

TLU Special Applications

This section describes two (2) specific applications that are supported: long lookups and partial CRC-32 for ATM Adaptation Layer-type 5 (AAL-5).

Using the RxByte Processor for Long Lookups

Some protocols that are implemented in the RxByte processor require that sophisticated, long lookups be launched from the SDP to hide latency.

Since the RxByte processor only has access to two (2) of the *TxMsg* registers to launch lookups from, the SDP can only launch lookups of keys that are up to 112bits in length. The two (2) lookup slots are referred to as *TxMsg0* and *TxMsg1*. The size of these registers are listed in Table 70 on page 307.

Table 70 TxMsg*n* Registers and Their Size

Register	Size (bits)
TxMsg*n*_Ctl	32
TxMsg*n*_Data_H	32
TxMsg*n*_Data_L	32

When launching a lookup from the SDP, the *TxMsg*n*_Ctl* register contains control information for putting the request onto the Ring Bus. Such information would contain the source Ring Bus node, destination Ring Bus node, message length, return message slot, and so on.

To use multi-slots, you must launch the lookup using the *TxMsg0_Ctl* register in addition to the *TxMsgn_Data_n* registers. The keys are contained in some or all of *TxMsg0_Data_H*, *TxMsg0_Data_L*, *TxMsg1_Data_H*, and *TxMsg1_Data_L* registers.

The *TxMsgn_Data_n* registers contain the actual data that the destination node (in this case the TLU) processes. Information contained in the *TxMsgn_Data_n* registers would include: the lookup table in the TLU, the command (*Read*, *Findr*, *XOR*, etc.) and most importantly the lookup key.

For keys that are 48bits, 96bits, and 112bits, two (2) sets of the *TxMsgn_Data_n* registers must be used. The formats of these registers are shown in Table 71 on page 308.

Table 71 Large Key Data Format, >48bits

	MSB			LSB
Field	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**
TxMsg0_Data_H	XXX	XXX	KU1	KU1
TxMsg0_Data_L	KU2	KU2	KU2	KU2
TxMsg1_Data_H	KL1	KL1	KL1	KL1
TxMsg1_Data_L	KL2	KL2	KL2	KL2

XXX = Reserved by TLU.
KU1 = Key upper 1 (upper 16b of key 1)
KU2 = Key upper 2 (lower 32b of key 1)
KL1 = Key lower 1 (upper 32b of key 2)
KL2 = Key lower 2 (lower 32b of key 2)

Depending on the configuration of the TLU table's key size, the information looked up as the key from the set of registers varies as listed in Table 72 on page 308.

Table 72 Key Size versus Key Match

Key Size	Key Match
32bit	Key match is done on contents of KU2 fields
48bit	Key match is done on contents of KU1 + KU2 fields
96bit	Key match is done on contents of KU2 + KL1 + KL2 fields
112bit	Key match is done on contents of KU1 + KU2 + KL1 + KL2 fields

Long Lookup Example for an Ethernet Application

For IEEE 802.1Q tagged frames, the Ethernet application must lookup the MAC address and the VLAN ID of the frame which is: 48bits + 12bits = 60bits. Since there is no 60bit key size available, use the next larger key size available which is (96bit). The format for the lookup is listed in Table 73 on page 309.

Table 73 Ethernet Application Lookup Format

	MSB			LSB
Field	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**
TxMsg0_Ctl	control information + length, indicating lookup uses both slots			
TxMsg0_Data_H	XXX	XXX	UUU	UUU
TxMsg0_Data_L	VVV	VVV	MM0	MM1
TxMsg1_Data_H	MM2	MM3	MM4	MM5
TxMsg1_Data_L	PPP	PPP	PPP	PPP

XXX = Reserved by the TLU
UUU = Unused
VVV = VLAN ID
MM*n* = MAC address
PPP = Not used by application (padded). These bits should be cleared by the application when doing lookups and when installing entries into the TLU.

From the SDP's perspective, the *TxMsgn_Data* words are mapped as shown in Table 74 on page 309.

Table 74 TxMsgn_Ctl Mapping

Bit	Mapping
TxMsg0_Data7	XXX
TxMsg0_Data6	XXX
TxMsg0_Data5	UUU
TxMsg0_Data4	UUU
TxMsg0_Data3	VVV
TxMsg0_Data2	VVV
TxMsg0_Data1	MM0
TxMsg0_Data0	MM1
TxMsg1_Data7	MM2

Table 74 TxMsgn_Ctl Mapping (continued)

Bit	Mapping
TxMsg1_Data6	MM3
TxMsg1_Data5	MM4
TxMsg1_Data4	MM5
TxMsg1_Data3	PPP
TxMsg1_Data2	PPP
TxMsg1_Data1	PPP
TxMsg_Data0	PPP

In this case, the fields that are matched (given that this is a 96bit key) are the VVV + MM*n* fields, resulting in the lookup of the MAC address + VLAN ID according to the 802.1Q specification.

Ethernet Application Example Implementation Notes

In the Ethernet application example, the SDP lookups just the VID in the RxByte processor if the frame is VLAN tagged. The CPRC gets the result of that looks up and looks up the returned FID + MAC address using the described method above to lookup the rest of the frame.

If the frame is untagged, then the MAC addresses are used with the Port VLAN ID (PVID) of the port from the SDP to only do one (1) lookup and therefore saves the extra latency of having the CPRC program do a lookup.

Only the TxMsg0_Ctl register can be used for multi-slot lookups.

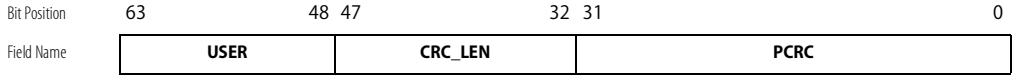
Partial CRC-32 Support

Partial CRC-32's for ATM Adaptation Layer-type 5 (AAL-5) reassembly can be stored and accumulated in a Data table type structure. The format and implementation steps are described in this section.

This is achieved using the SDP in conjunction with the TLU to accumulate the partial CRC's for many Virtual Connections (VC). Specifically, this is supported using the TLU's *XOR* command, a VC data table, a CRC-32 Correction Table, the *CRC-32_Correction_Table_Base_Address* register, and the *CRC-32_Checkvalue* register. The *XOR* command automatically increments the *CRC_Len* field bits [47:32] after adding a new partial CRC. Refer to "XOR Command" on page 270.

Partial CRC-32 Data Entry Format

The Cyclic Redundancy Check (CRC) entry is used for CRC error checking as part of a Virtual Connection (VC) data table. The entry must conform to this format:



Field Name	Bit Position	Description
USER	63:48	**User Defined Data** — The user can insert anything here.
CRC_LEN	47:32	**CRC length** — Number of cells holding current packet.
PCRC	31:0	**Partial CRC** — The current partial CRC.

Partial CRC-32 General Setup

Follows these steps to implement the Partial CRC-32 Operation:

1. Create and initialize CRC-32 Correction Table. (Each entry contains a CRC calculated for 0xFF plus 48 times the number of cells; that is, 4Bytes of zeros).
2. Set up *CRC-32_Correction_Table_Base_Address* register to the start address of the newly created table (CRC-32 Correction Table).
3. In the *SDPMode3* and *SDPMode5* registers, set both the *RxByteCRCinit* field and *TxByteCRCinit* field= 0 in order to initialize the CRC-32 block to zero when the *CRCinit* command is executed by the RxByte and TxByte programmable processor's microcode.

Partial CRC-32 Rx Setup and Operation

Follow this step to implement the Rx side of the Partial CRC-32 Operation:

1. Set up Rx CRC data entry in the receive VC table as follows: *CRC_Len*= 1, *PCRC*= 0.

All cells except for the last end of message (EOM) cell are handled via the RxByte programmable processor, as follows:

1. RxByte programmable processor calculates *PCRC* on current cell.
2. RxByte programmable processor sends TLU XOR command with current PCRC.

TLU's *XOR* command settings are indicated here:

Bit Position	63 61	60 57	56 55	54 48	47 32	31 24	23 20	19 0
Field Name/Setting	CMD=1	VTB#= User Defined	CRC= 0x01	OFF=User Defined Offset	N/A	MSK=0x0F	Rsvd	IDX=User Defined Index
Optional	Reserved					PCRC=From RxByte Calculation		

3 TLU *XORs* PCRC with (accumulated) PCRC and obtains a (new accumulated) PCRC. Then this value is shifted by 48Bytes and stored as the new PCRC in the VC table's CRC-32 Entry.

Last cell end of message (EOM) is handled via the RxByte programmable processor, as follows:

1 RxByte programmable processor calculates PCRC on last cell.

2 RxByte programmable processor sends TLU *XOR* command with PCRC and sets the *XOR* command bits [56:55] *CRC* field= 11 (Rxlast).

TLU's *XOR* command settings are indicated here:

Bit Position	63 61	60 57	56 55	54 48	47 32	31 24	23 20	19 0
Field Name/Setting	CMD=1	VTB#= User Defined	CRC= 0x11	OFF=User Defined Offset	N/A	MSK=0x0F	Rsvd	IDX=User Defined Index
Optional	Reserved					PCRC=From RxByte Calculation		

3 TLU receives request and *XORs* PCRC with (accumulated PCRC); TLU does a table lookup based on PDU length to PCRC and adds the value looked up with the (accumulated PCRC). TLU compares (accumulated PCRC) with *CRC-32_Checkvalue* register. If the *CRC_Checkvalue* register does not match, the TLU sets a Ring Bus error indication for the PCRC. TLU re-sets accumulated *PCRC*=0, and *CRC_Len*= 1.

Partial CRC-32 Tx Setup and Operation

Follow this step to implement the Tx side of the Partial CRC-32 Operation:

1 Initialize Tx CRC data entry in the transmit VC Table as follows: *CRC_Len*=0, and *PCRC*=0.

All cells except for the last end of message (EOM) are implemented via the SDP, as follows:

1 CPRC make the cell available to SDP for transmission.

2 TxByte programmable processor calculates PCRC as it transmits the cell.

3 TxByte programmable processor sends TLU *XOR* command request containing PCRC; TLU *XORs* with the PCRC, performs 48Byte shift and sends command to store this in the table as the (new accumulated PCRC).

TLU's *XOR* command settings are indicated here:

Bit Position	63 61	60 57	56 55	54 48	47 32	31 24	23 20	19 0
Field Name/Setting	CMD=1	VTB#= User Defined	CRC= 0x01	OFF=User Defined Offset	N/A	MSK=0x0F	Rsvd	IDX=User Defined Index
Optional	Reserved					PCRC=From TxByte Calculation		

Last cell end of message (EOM) is implemented, as follows:

1 CPRC sends TLU *XOR* command (prior to giving last cell to SDP). The TLU does a lookup into the CRC-32 Correction Table indexed by the *CRC_Len* field (which is equivalent to the number of cells in the PDU -1). The TLU XORs this with the (accumulated) PCRC and returns the PCRC (accumulated for all cells except the last) to the CPRC via the Ring Bus. The TLU sets *PCRC*= 0, *CRC_Len*= 0 (based upon data field) after returning the (accumulated PCRC) in the Partial CRC-32 Data Entry Format. The TLU sets the error bit in the Ring Bus register.

The Ring Bus error bit should be ignored.

TLU's *XOR* command settings are indicated here:

Bit Position	63 61	60 57	56 55	54 48	47 32	31 24	23 20	19 0
Field Name/Setting	CMD=1	VTB#= User Defined	CRC= 0x10	OFF=User Defined Offset	N/A	MSK=0x0F	Rsvd	IDX=User Defined Index
Optional	Reserved					PCRC=0		

2 CPRC writes (accumulated) PCRC to its merge register space.

3 The TxByte programmable processor accumulates a PCRC for the last cell. It then *XORs* this value with the (accumulated PCRC) from its CPRC's merge space, performs the 1's complement and appends to the frame. SDP transmits cell with correct CRC.

Single cell packets are implemented, as follows:

1 CPRC writes PCRC Accumulated= 0xFFFFFFFF to merge register space.

Chapter 7

Queue Management Unit

Chapter Overview

This chapter covers the following topics:

Queue Management Unit (QMU) Overview

The Queue Management Unit (QMU) provides queuing service to all the processors (CPs, XP and FP) on the C-5 NP. The QMU queues are used by the processors to switch *payload descriptors* from input processors (CPs, XP and FP) to output processors (CPs, XP and FP) using Control Blocks (WrCB0_ or RdCB0_) via the Payload Bus.

The C-5 NP processors generate the descriptor data in their respective (DMEM), then write the data into a queue stored in the SRAM. The configurable QMU performs queue management while simply passing the descriptor data through without modification; it does not parse the data records that it enqueues.

The QMU provides up to five-hundred-twelve (512) queues using an on-chip memory (internal SRAM) for control structures and off-chip memory (external SRAM) for descriptor storage.

Queues can be allocated asymmetrically to processors, such that one (1) CP could have zero (0) to one-hundred-twenty-eight (128) queues. Up to 16,384 descriptor buffers can be enqueued simultaneously across all queues.

Generally, the data types enqueued in the QMU are either:

- A payload descriptor including a payload buffer tag (BTag), or
- A user-defined inter-processor message.

Payload Descriptors Enqueued to the QMU

Payload Descriptors are small fixed-size (12, 16, 24, or 32Bytes) data structures that contain all the information required to complete the forwarding of a received payload data unit from the ingress processor, for example, the information required to build a header at the output interface. Payload descriptors are created by the application program running on the ingress processor, generally a Channel Processor RISC Core (CPRC).

Typically, the QMU queues are used as egress queues associated with specific data for egress processors, (CPs, the XP, or the FP). Only one (1) egress processor can drain each queue, but any of the ingress processors can write into any queue.

Used-Defined Inter-processor Messages Enqueued to the QMU

Inter-processor messages are also small fixed-sized (12, 16, 24, or 32Bytes) data structures that contain user defined information. Generally, inter-processor messages are used to orchestrate control plane activities such as flow control, statistics gathering, or table maintenance. For example, an ingress processor could build a message then enqueue it to a queue serviced by the XP for off-line processing.

QMU Major Components

The major components of the QMU are listed in Table 75 on page 317. In addition, Figure 68 on page 319 shows the QMU Block Diagram.

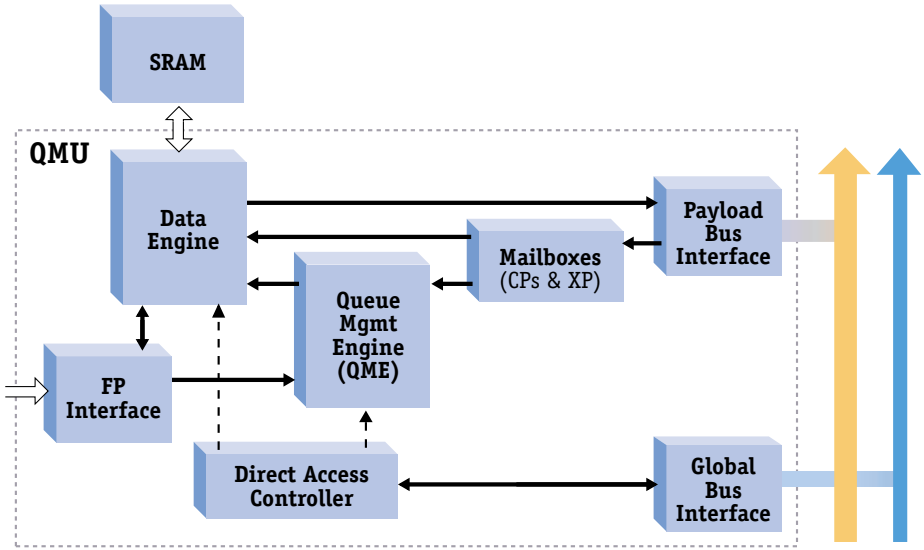
Table 75 Major Components of the QMU and Their Functions

Item	Function
Queue Management Engine	The Queue Management Engine (QME) manages all descriptor queues in SRAM and maintains per-queue status information (descriptor weight, queue weight, and queue length) that is delivered with each descriptor to the dequeuing processor (CPs, XP, or FP). The QME provides multicast elaboration for descriptors that must be replicated to more than one queue. A single buffer descriptor can be enqueued to any number of output processors (CPs, XP, or FP) with a single command (multicast enqueue) from the descriptor generator (CPs, or XP). The descriptor is enqueued at a specified queue level at each listed output port so that each port receives a copy of the descriptor. The QME implements and operates on a link-list of descriptors. Every descriptor buffer in the QMU starts out as an entry on a free-descriptor buffer list. Then during an enqueue operation, a descriptor buffer is removed from the free-descriptor buffer list and the payload descriptor is copied into a queue. The reverse process happens for dequeue operations. The QME must perform (2) two functions to respond to a processor's (CPs, XP, or FP) enqueue or dequeue request. First, the descriptor must be stored in or retrieved from the external QMU SRAM array. Second, the internal queue controls inside the internal SRAM must be updated to add the entry onto or remove the entry from the desired queue.
Direct Access Controller	Initialization, queue configuration changes, and read/write of the QMU internal registers and QMU external memory are performed using loads and stores on the Global Bus. If the transaction is a read, the Direct Access Controller (DAC) gathers the value from the addressed location in the QMU and returns it to the requesting processor on the Global Bus. When one of the QMU's queues state changes from empty to non-empty as the result of an enqueue operation, the Queue Ready Generator (QRG) takes the queue number, determines the processor to notify, then generates the appropriate message.
Mailboxes	Multi-Use Control Block (WrCB0_, RdCB0_) transfers from CPs and the XP arrive at the QMU over the Payload Bus and are held in a Mailbox. Each processor (CPs, XP) has its own Write Mailbox (CP*n* Wr MailBox) and its own Read Mailbox (CP*n* Rd Mailbox). Each mailbox holds a single command used to perform specific operations. Queue status and dequeue operations are performed using (RdCB0_). Unicast enqueue, multicast enqueue, and configure queue operations are performed using (WrCB0_). Each of the two (2) mailboxes for each CP or the XP operate independently of the other. They each have their own available/busy status and success/fail status information reported to the CP or XP independently.

Table 75 Major Components of the QMU and Their Functions (continued)

Item	Function
Fabric Port Interface	The Fabric Port has a separate interface to the QMU because of its very high throughput requirement. When the FP has queuing operations pending, it receives no less than half the QMU's descriptor throughput. A dedicated path is used to write FP queuing operations from the FP to the QMU's FP command FIFO. The QMU returns descriptors to the FP over the Payload Bus. Queue Ready Generator (QRG) reports from the QMU are sent to the FP, just like those for any of the CPs or the XP. The QMU buffers up to eight (8) enqueue operations and eight (8) dequeue operations for the FP.
Data Engine	The Data Engine moves payload descriptors or user-defined inter-processor messages on and off the queues stored in external SRAM. This interface adjusts the timing of the internal information (address and data) so that the setup and hold times on the external memory interface are met for both memory writes and reads.
External SRAM	The QMU uses external ZBT SRAM to hold the data enqueued in the QMU queues. The interface is 32bits wide and runs at half of the core clock rate. The memory is organized in power-of-2 sized blocks big enough to hold the configured descriptors.
Internal SRAM	Contains queue configuration information that includes descriptor link-list, descriptor weight, descriptor dynamic memory, queue head-tail, queue length, queue weight, queue parameters, and dynamic descriptor Pool*n* usage.
Configuration Registers	Used for mapping of queues to CPs, XP and FP, configuration of the QMU, QMU debugging, and collecting QMU statistics.

Figure 68 QMU Block Diagram



QMU Flow Process

The QMU flow is described in this section.

Flow Details for CPs/XP Inputs and FP Inputs

The front part of the input flow is a little different between the CPs/XP and the FP. However, the activities occurring from the Command Processor on are the same for both the CPs/XP and FP. The entire CPs/XP input flow is detailed first, then the front part of the FP later.

CPs and XP Input Flow

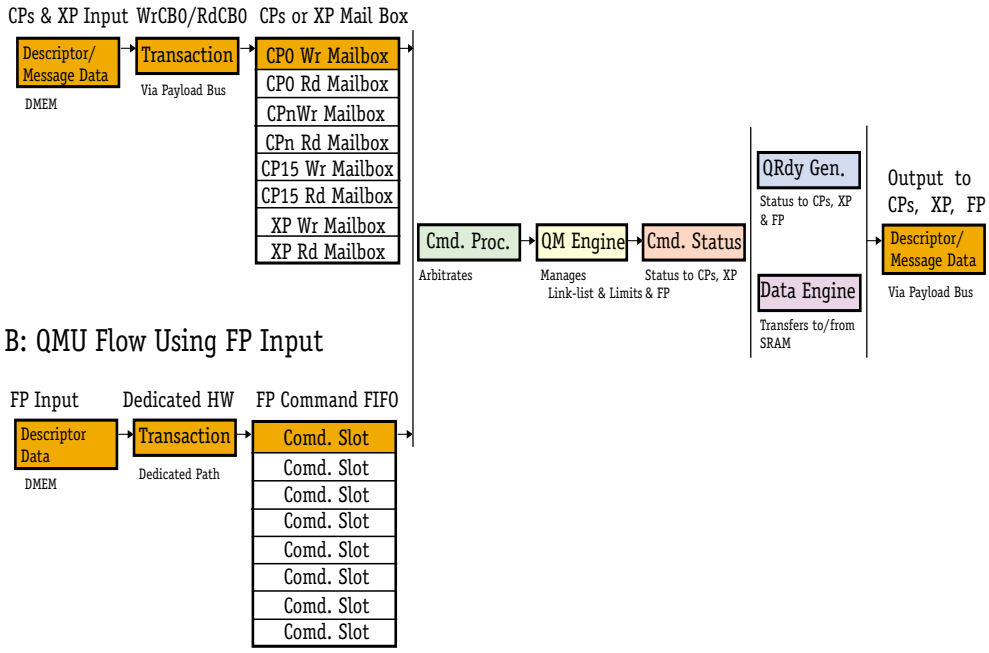
Each processor (CPs, XP) writes the descriptor or message data (user-defined inter-processor message) into its DMEM, then writes the data and a command using a Control Block (WrCB0_) to the processors' dedicated mailbox (CP*n* Wr Mailbox) via the Payload Bus. This data remains in this holding register until the command processor (Cmd. Proc.) arbitrates to determine which mailbox to execute next. The command status generator (CSG) monitors the status of each mailbox throughout the process and reports that status back to the processor. The mailbox empty or non-empty state is used to determine when the applicable mailbox is available for a new operation.

The queuing management engine (QME), upon receipt of the command, manages the free descriptor buffer list, queueing, and storage of queue data in the SRAM using a link-list. A link-list tracks the free descriptor buffers, used descriptor buffers for queueing, and the location of the data (descriptor data) in queues in the SRAM. The queue ready generator (QRG), upon receipt of the data, notifies the processor that the queue is non-empty. It determines the correct processor using the queue number and sends the message over the Payload Bus. The data engine (DE), upon direction of the queue engine (QE), physically transfers the data to/from the external SRAM. During a write (enqueue operation), the data engine (DE) reads the content of the (CP*n* Wr Mailbox) that holds the data, then writes that data into SRAM per the queuing engine (QE) and its link-list. During a read (dequeue operation), the data engine (DE) reads the data (descriptor data) from the SRAM per the queuing engine (QE) link-list, then FIFOs them to transmit back to the CPs/XP using the (RdCB0_) via Payload Bus.

FP Input Flow

The FP writes the descriptor data and a command into its (DMEM), then writes the data and a command using a dedicated interface to the FP command FIFO in the QMU. This data is held in a FP command FIFO (FPCFIFO) until the command processor (Cmd. Proc.) arbitrates to determine which holding register to execute next. At this point, the rest of the FP input flow is identical to the CPs/XP flow starting with the command status generator (CSG), as described in "CPs and XP Input Flow" on page 320.

Figure 69 QMU Flow Diagram



Queue Organization

The queue memory consists of both external and internal SRAM. Each is described here.

External SRAM

The queue memory's external SRAM is organized in the following manner:

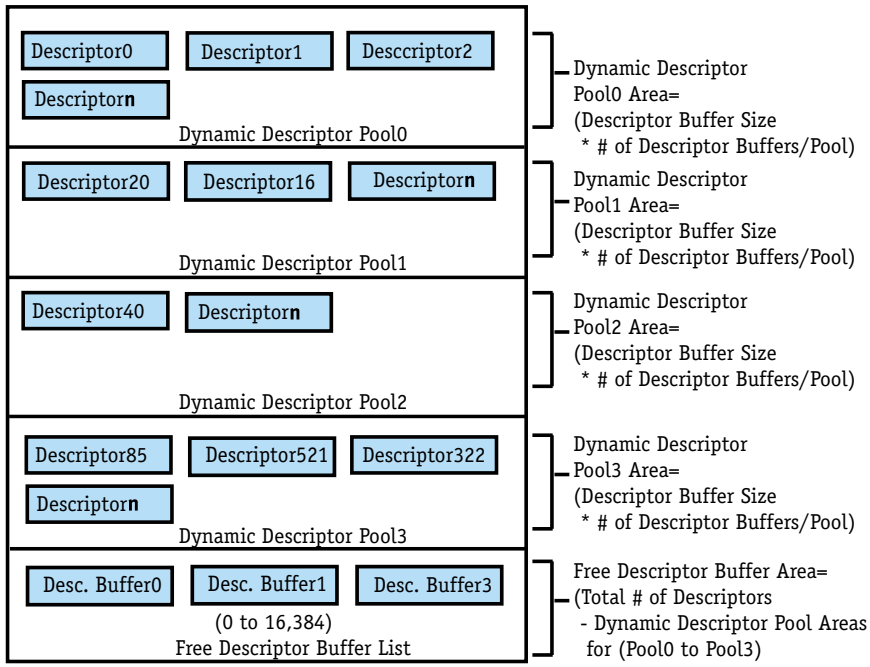
Descriptor Buffer

Descriptor buffers are fixed-sized data structures that are written with payload descriptors and linked to a queue during an enqueue operation. Descriptors are stored in the descriptor buffers located in the QMU's external (SRAM). The QMU allows (0 to 16,384) total descriptor buffers using the *Num_Descriptors* register. The Descriptor buffer sizes supported are (12, 16, 24, or 32Bytes), using the *Descriptor_Size* register. Refer to Figure 70 on page 323, and Table 77 on page 326.

Dynamic Descriptor Pools

The QMU maintains sections of SRAM called *dynamic descriptor pools* containing descriptors. Pools are intended to provide protection among the many users of the queues. Up to four (4) dynamic descriptor pools can be configured. Each Dynamic Descriptor Pool Area= (Descriptor Buffer Size * Number of Descriptors per Pool). Each Dynamic Descriptor Pool*n* can grow to contain a number of descriptors, the legal range is (0 to 16K). A limit is used to guarantee a maximum number of descriptor buffers a queue may hold within the (0 to 16K) range. This limit is implemented using a Dynamic Descriptor Usage Limit Pool. There are four (4) of these registers: *Dyn_Des_Usage_Lim_Pool0* to *Dyn_Des_Usage_Lim_Pool3*. Each Dynamic Descriptor Pool has an associated Dynamic Descriptor Usage Limit Pool. Refer to Figure 70 on page 323 and Table 77 on page 326.

Figure 70 External SRAM Storage Space for Descriptor Buffer Data



Dynamic Descriptor Usage Limit PooIn

During a unicast enqueue to a queue, permission to take the next free descriptor buffer and enqueue it on the queue is based on the state of that queue's Dynamic Descriptor Usage Limit. Each queue is assigned to one (1) of four (4) registers: *Dyn_Des_Usage_Lim_Pool0* to *Dyn_Des_Usage_Lim_Pool3*. Each pool has a usage counter and a usage limit parameter. The usage count is not allowed to exceed the usage limit during unicast enqueues.

The idea behind the Dynamic Descriptor Buffer Usage Limit pools is to provide separation between service classes that need to use dynamic buffering. For example, the use of dynamic buffers by ATM's Variable Bit Rate (VBR) service should not impact the availability of dynamic buffers used by the Constant Bit Rate (CBR) service. The limit on each dynamic pool's descriptor usage is individually configurable because different services require different degrees of traffic variability (burstyness).

When an enqueue is requested to a queue that is in its dynamic range, the queue's pool descriptor usage is compared to that pool's usage limit. If the current pool usage is below the pool's limit, the enqueue is done and the pool usage count is incremented. When that descriptor is later dequeued, the pool usage count is decremented. Refer to "Queue Length Allowance and Length Limit Parameters" on page 330.

When the QMU is initialized, the pool usage limits, the total number of descriptors, and the allowances of all the initialized queues must be configured to work together. The total number of Descriptors allocated among all four (4) pools of the *Dyn_Des_Usage_Lim_Pool0* to *Dyn_Des_Usage_Lim_Pool3* registers should be < the number of dynamically enqueued descriptors. This way, there will be no depletions of the supply of free descriptor buffers.

All queues that are members of a single multicast group should share the same pool. See "Multicast Operation Throughput Considerations" on page 352 for more information.

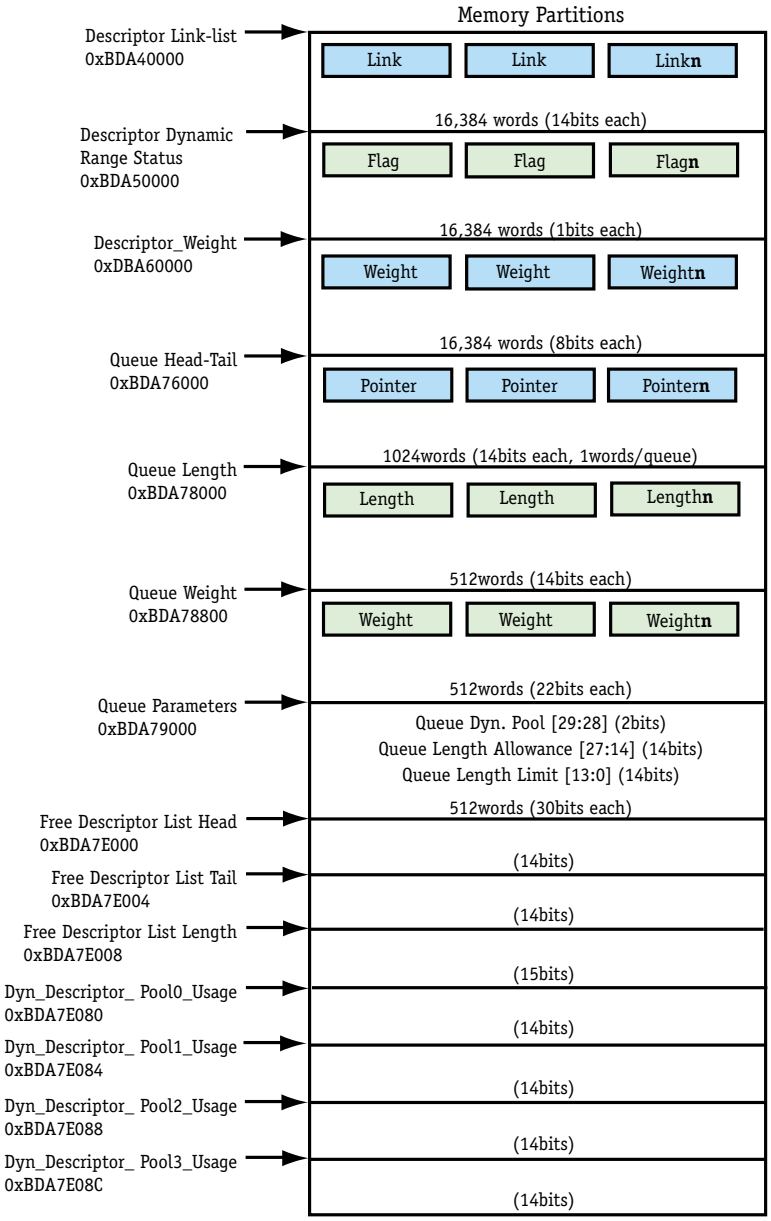
Internal SRAM

The queue memory's internal SRAM contains all the data structures required to build the queues, maintain queue status, and provide descriptor buffer management. The internal memory is divided into the following sub-sections. Refer to Figure 71 on page 325, and Table 76 on page 324.

Table 76 QMU Internal SRAM Sub-Sections and Their Functions

Memory Sub-Sections	Function
Descriptor Link-list	Descriptor links that form the queues.
Descriptor Dynamic	One (1) bit per descriptor that flags wether the descriptor was allocated in a queue's dynamic range.
Descriptor Weight	The weight of each descriptor.
Queue Head-Tail	Pointers to each queue's head and tail.
Queue Length	The length of each queue.
Queue Weight	The accumulated weight of each queue.
Queue Parameters	Configuration for each queue.
Free Descriptor List Head	Pointer to free descriptor list head.
Free Descriptor List Tail	Pointer to free descriptor list tail.
Free Descriptor List Length	The length of free descriptor list.
Dynamic Descriptor Pool0 Usage to Dynamic Descriptor Pool3	The number of dynamically enqueued descriptors in each pool.

Figure 71 Internal SRAM Space



QMU Variables

The QMU uses the following variables shown in Table 77 on page 326.

Table 77 Legal Ranges for SRAM Variables

<table>
<tr><th>Item</th><th>Range</th></tr>
<tr><td>Number of Dynamic Descriptor Buffer Pools</td><td>Up to 4</td></tr>
<tr><td>Number of Descriptor Buffers per Pool</td><td>0 to 16K</td></tr>
<tr><td>Number of Descriptors allowed in the QMU</td><td>0 to 16,384 as detailed here:
<table><tr><th>Programmed Value</th><th>Number of Descriptors</th></tr><tr><td>0
•
•
•
16,383</td><td>1
•
•
•
16,384</td></tr></table></td></tr>
<tr><td>Individual Descriptor Size</td><td><table><tr><th>Size (Bytes)</th><th>Encoded Value</th></tr><tr><td>12</td><td>0</td></tr><tr><td>16</td><td>1</td></tr><tr><td>24</td><td>2</td></tr><tr><td>32</td><td>3</td></tr></table></td></tr>
<tr><td>Number of Descriptors Enqueued Dynamically to the Queues Associated with a Pool</td><td>0 to 16K</td></tr>
<tr><td>Number of Queues Allowed in the QMU</td><td>0 to 511 as detailed here:
<table><tr><th>Programmed Value</th><th>Number of Queues</th></tr><tr><td>0
•
•
•
511</td><td>1
•
•
•
512</td></tr></table></td></tr>
</table>

Table 77 Legal Ranges for SRAM Variables (continued)

Item	Range
Number of Queues Mapped per Processor	Up to 128
Queue Level Number (for Multicast Enqueue Operations Only)	0 to 7

Queue Mapping and Parameter Characteristics

The basic queue characteristics are described in this section.

Queue to Processor Mapping

A queue is a FIFO that contains descriptor data.The QMU supports (0 to 511) queues available to the entire C-5 NP. Up to 128 (of those 512) queues can be mapped to a given processor, (CPs, XP, or FP).

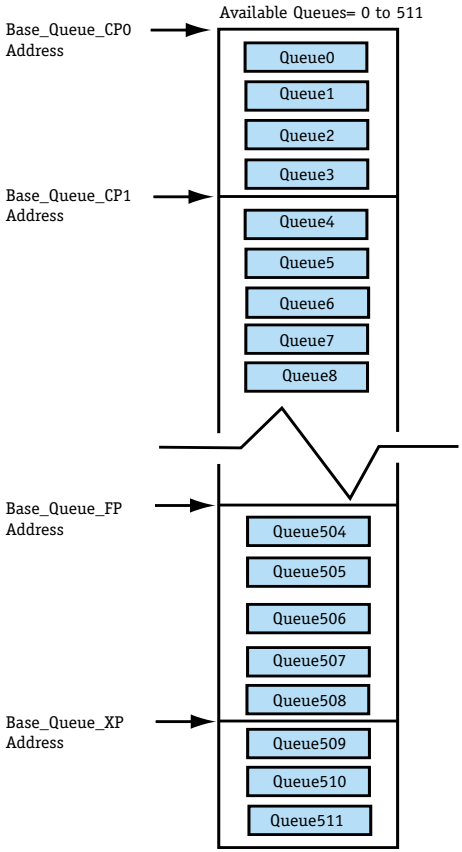
The QMU uses a simple mapping scheme to establish a flexible association between processors and their queues for both unicast enqueue and multicast enqueue operations.

The QMU maps individual queue numbers (0 to 511) to their respective processors (CPs, XP, or FP) using the *Base_Queue_CP0* to *Base_Queue_CP15*, *Base_Queue_FP*, and *Base_Queue_XP* registers. Specifically, using the *Base_Queue_Number* bits [8:0] field contained in the *Base_Queue_CP0* to *Base_Queue_CP15*, *Base_Queue_FP*, and *Base_Queue_XP* registers.

The *Base_Queue_nnn* is the lowest numbered queue owned by that processor. The rest of a processor's queues are located at offsets above the *Base_Queue_nnn*. The offset range cannot be larger than one-hundred-twenty-eight (128). Each processor is assigned its queues sequentially within the (0 to 511) queue number range.

These queues are the ones targeted by unicast and multicast enqueues to that processor, and are the queues that stimulate queue non-empty transition notifications to the processor. This scheme allows an arbitrary number of queues to be assigned to each processor (the *Base_Queue_ Number* bits [8:0] fields are unconstrained). Refer to Figure 72 on page 329.

Figure 72 Mapping Queues to Processors for Unicast/ Multicast Enqueue Operations Example



Queue to Processor Mapping Rules

The assignment of queues to processors must follow these rules:

- The ranges of queues starts with queue 0 for CP0.
- The range for CP1 follows immediately after the range for CP0 with no gap. This pattern continues through the remaining CPs in increasing CP number, and then to the FP and the XP.
- The last XP queue must not exceed the total number queues initialized.
- There can be no more than one-hundred-twenty-eight (128) queues per processor.

When there are no queues assigned to a processor, its *Base_Queue_nnn* register must still be programmed. Its value should be the offset for the "next available queue" following the last queue of the next lower numbered processor.

If a processor is assigned zero (0) queues, the next higher processor (based on Processor ID) is assigned the same Base_Queue_Number bits [8:0] as the lower processor. Therefore, two (or more) processors can have the same Base_Queue_Number bits [8:0] value if the lower processor(s) has been assigned zero (0) queues.

If any descriptors are enqueued for a processor with no queues, those descriptors are enqueued in a queue of the higher numbered processor. When one of the next-higher numbered processor's queues is non-empty, the notification is sent to the highest numbered processor sharing the *Base_Queue_Number* bits [8:0] value, but the higher numbered processor becomes the owner of the queue.

Queue Length Allowance and Length Limit Parameters

The supply of Descriptor Buffers within the QMU must be properly managed to prevent the depletion of Descriptor Buffers for queues not at the "traffic hot-spots". To avoid this situation, a minimum number of Descriptor Buffers always needs to be available to each queue regardless of the use of Descriptor Buffers by other queues. Two (2) parameters are used to set the minimum and maximum amount of Descriptor Buffers used to provide a free flow of Descriptor Buffers to the queues that need them at a given point in time. They are: *Queue Length Allowance* and the *Queue Length Limit*. Allowance is the minimum amount of the (0 to 16K) range, whereas, Limit is the maximum of the (0 to 16K) range. Thus, each queue is initialized with these two (2) parameters:

- **Queue Length Allowance** — Is the guaranteed minimum allocation of Descriptor Buffers of (0 to 16K) range. Buffers are implicitly reserved so that a queue can always fill to its allowance. The sum of the allowances of all the queues is always smaller than the total supply of Descriptor Buffers, typically much smaller.
- **Queue Length Limit** — Is the guaranteed maximum number of unicast Descriptor Buffers a queue is allowed to hold in the range of (0 to 16K). The sum of the limits of all the queues is typically much greater than the supply of Descriptor Buffers.

These parameters (Allowance and Limit) refer to the number *of Descriptors enqueued in a queue; they do not specify which Descriptor Buffers are enqueued in any given queue.*

When a queue has filled to an amount between its Allowance and its Limit, it is in its *dynamic range*. In this *dynamic range*, Descriptor Buffers may or may not be available to the queue for enqueuing depending on the use of Descriptor Buffers by other queues. Whether or not a Descriptor Buffer is available in this situation depends on the number of Descriptor Buffers currently enqueued in other queues.

Descriptor Buffer availability is dynamic in that it depends on the current state of the QMU that results from the dynamic traffic pattern. If other queues are not using many Descriptor Buffers at a given moment in time, a queue at the “traffic hot-spot” can get more than its share. If there are a lot of enqueued Descriptor Buffers, each queue gets at least its Allowance, (that is, its fair share under congestion). This gives the QMU the appearance of many more Descriptor Buffers than it actually has if they had been dedicated to specific queues.

Queuing Operations

The QMU receives queue operation requests via the Payload Bus. Enqueue Operations use a control block (WrCB0) to write the descriptor data into a queue in the QMU's external SRAM from the DMEM of either the requesting CP, or XP via the Payload Bus. In addition, Dequeue Operations use a control block (RdCB0) to read descriptor data from a queue in the QMU's SRAM into the DMEM of the requesting CP, or XP via the Payload Bus. Descriptor data is transferred to/from the QMU in 16Byte units; up to two (2) units (32Bytes) can be transferred in one (1) Payload operation along with other information contained in the payload transaction.

QMU Run Enable

The QMU must be enabled to process queuing operation requests from the processors (CPs, XP, or FP). The *QMU_Run_Enable* register has one (1) bit that enables the QMU to process Payload Bus operations. This bit must be set after initialization to allow the QMU to go online.

This bit can be cleared at any time. Once cleared, this bit disables the QMU from starting the processing of any subsequent queue operations. A queue operation currently being executed is not interrupted by the clearing of this bit, but no subsequent queue operations are started. This allows a clean shutdown of the QMU so that its internal state is not corrupted. Once the QMU is shut down, a processor (CPs, XP, or FP) can access all of the internal state of the QMU (both registers and memory). Any queue commands left in the mailboxes (Wr Mailbox/Rd Mailbox) are executed in normal order when the QMU is re-enabled, just as if the QMU had never been disabled. Refer to "QMU_Run_Enable Register (QMU Enable Queue Function)" on page 501.

Enqueue Operation

Processors (CPs, XP, or FP) enqueue payload descriptors to designate to the output processor how to forward packets/cells. Upon a successful enqueue, the requesting processor is freed from having to keep track of the packet/cell buffer. It has effectively handed this packet/cell buffer off to the output function by way of the QMU.

Payload (Wr/Rd) Servicing Order During Enqueue Operation

Queue operations arriving from the CPs and the XP are serviced from the mailboxes by type (Wr/Rd) in the order of their arrival at the QMU over the Payload Bus. Payload write (Wr) operations from any CP or the XP are executed in arrival order before any waiting payload read (Rd) operations are executed in their arrival order. In order to facilitate the FP operating at full capacity, the QME gives FP commands at least one half the queuing bandwidth in proportion to its traffic bandwidth. The CP/XP operations, either read or write operations, are interleaved with FP operations.

Since only one (1) command of each type (Wr/Rd) can be buffered in the QMU from each of the CPs and the XP, a mailbox status for each ensures that commands are not lost. Direct wiring of mailbox status and command execution status information from the QMU to the *CP_Mode0* register ensures that the CPs do not need to spend system-level resources to determine if the previous command completed successfully or not.

An attempt to over-write a busy mailbox causes a Payload Bus NACK. An attempt does not disturb the previously written command.

Causes of Enqueue Failure

When an enqueue operation failure occurs, the requesting processor (CPs, XP, or FP) must either retry the enqueue at a later time or drop the packet/cell and reuse the Buffer Tag (BTag). An enqueue operation can fail for the following reasons:

- A *Dyn_Descriptor_Pooln_Usage* shows that the pool is empty when a queue is in its dynamic range.
- The *Free_Descriptor_Buffer_List* has no descriptor buffers left.
- The unicast target queue has exceeded its buffer-use limit.
- The target queue does not exist (that is, it was never configured).

Dequeue Operation

Because the QMU queues are output queues, there must be a single processor that owns each active queue. That owner is responsible for draining the queue. A successful dequeue operation triggers the entire forwarding process. Therefore, when a dequeue failure occurs, there is no descriptor to drive a packet/cell transmission.

Queue Servicing Policy During Dequeuing Operation

The dequeuing processors (CPs, XP, or FP) determine the queue service policy, not the QMU. In addition, several queues can be assigned to each CP for Quality of Service (QoS) purposes. The dequeuing processor is free to implement any priority, service weighting, or scheduling policy.

Dequeuing processors are responsible for scheduling their own workloads. To aid in this, the QMU provides *weights* for each queue. Each queue entry has a weighting factor associated with it. In typical frame-switching applications, each descriptor is assigned a weight which is related to the length of the frame stored in the associated BMU buffer. The sum of the enqueued descriptor weights is kept for each QMU queue. Each enqueue and dequeue operation updates the value for its queue. For multicast enqueue operations, the queue weight is adjusted for all the specified queues.

When a descriptor is dequeued from a queue, the descriptor's weight, the queue's new weight, and the queue's length are returned to the processor's DMEM along with the descriptor itself. Refer to "Dequeue Operation" on page 348.

Causes of Dequeue Failures

A dequeue operation can fail for these reasons:

- The target queue is empty.
- The target queue does not exist (that is, it was never configured).

Status Reporting

Status is reported on Mailboxes and Queues. Queue status information is made up of three (3) types: Empty to Non-Empty State Notification, Dequeue Operation, and Buffer Management.

Mailbox Availability and Status Reporting of Mailboxes

The QMU maintains a write (CP*n* Wr Mailbox) and a read (CP*n* Rd Mailbox) mailbox for each CP and the XP (XP Wr Mailbox, and XP Rd Mailbox). These mailboxes are invisible to application developers except that the state of each is reported via the *CP_Mode0* register for the processor. Specifically, the *CP_Mode0* register, bits [23:22] *QMURdMbxStatus* field and bits [21:20] *QMUWrMbxStatus* field are visible. Using these two (2), two (2) bit fields, status states reported include: operation success, operation error, busy-wait, or busy-executing. The mailbox scheme provides a single holding register per source (that is, only one (1) command is outstanding per mailbox at a time).

This allows success/fail operation information to be conveyed back to the requesting processor (CPs or XP) in an orderly fashion. Errors reported (set to 1) in the *CP_Mode0* register occur when the previous queue operation encountered an error and should be repeated. Generally, it means the queue operation failed, because of a resource error such as queue non-empty or descriptor buffer pool limit was exceeded.

Mailbox status is also reported through both *Event0* and *Event1* registers. Specifically, *Event0* register bit [60] *QMUError* field, as well as the *Event1* register bit [31] *QRdMbxAvail* field, bit [26] *QRdMbxBusy* field, bit [15] *QWrMbxAvail* field, and bit [10] *QWrMbxBusy* field are used to determine when the applicable mailbox is available for a new queue. The avail bits indicate the mailbox made a state transition from busy to empty. The busy bits indicate a transition from empty to busy. The error bit indicates a transition from busy to error.

Queue Status Information

There are several types of queue status information that can be accessed to inform the processors (CP, XP, or FP) in their forwarding tasks:

- Queue non-empty transition notifications presented via *Queue_Status0, Queue_Status1*, *Queue_Status2* and *Queue_Status3* registers, as well as, the *Event0* and *Event1* registers.
- Dequeue status information (the Queue's weight and length) that is included with a descriptor when it is dequeued. Refer to "Dequeue Operation" on page 348.
- Buffer management status that is included in the *Free_Descriptor_Buffer_List* and *Dyn_Descriptor_Pooln_Usage* registers.

Queue Empty to Non-empty State Notification Process Information

Queue status is made visible to the processors (CPs, XP, or FP) via the *Event1* register and the *Queue_Status0*, *Queue_Status1*, *Queue_Status2*, *Queue_Status3* registers in each processor's configuration space. Payload operations (RdCB0) can be used to read the complete queue status from the QMU when the normal updates that come with a dequeue are insufficient. Refer to "Queue Status Operation" on page 342. Autonomous announcements of queue status by the QMU are made on the Global Bus and are handled by CP and XP hardware.

When a queue becomes non-empty, its dequeuing processor (CP, XP, or FP) can begin to drain it. The QMU announces this change of state (empty to non-empty) to the queue's dequeuing processor indicating the queue is now ready to be serviced. The dequeuing processor's software is then responsible to keep track of the ready status of the queue until the queue is completely drained.

Queue change of state (empty to non-empty) notifications are issued whether the enqueued descriptor came from a unicast or a multicast enqueue. Each multicast target queue is treated individually in the generation of these notifications.

Queue status non-empty transitions are automatically loaded into the four (4) queue status registers, where they can be read by the CPRC. *Queue_Status0*, *Queue_Status1, Queue_Status2, Queue_Status3* bits are set to one (1) when a queue state changes from empty to non-empty. The dequeuing processor is responsible for clearing these bits when it handles the status change (that is, before it begins to read descriptors from the queue). The logical OR of the bits in these status registers also provides a level-sensitive event for input to the *Event1* register. Refer to "Queue_Status0 Register (CP Queue Status Function)" on page 433,Table 131 on page 433.

Dequeue Status Information

The dequeuing processor (CPs, XP, or FP) of the descriptors is given a queue and descriptor status with each dequeue operation. The QMU sends the following three (3) types of status information to DMEM along with the descriptor being delivered:

- Descriptor Weight, Queue Weight, and Queue Length
  - The Descriptor Weight — The weight of that descriptor.
  - The Queue Weight — Accumulated weight of the queue after the dequeue of this descriptor.
  - Queue Length — The queue length after the dequeue of this descriptor.

The queue status information can also be obtained using the Queue Status Operation. Refer to"Queue Status Operation" on page 342.

Buffer Management Information

The processor receives an update of the queue's length with each descriptor read (RdCB0 for Dequeue Operation and Queue Status Operations). When the reported queue length drops to zero (0), the processor has just received the last enqueued descriptor and should stop issuing descriptor reads until it receives a queue non-empty transition notification for that queue.

Additional QMU status information can be read during on-line operation via Global Bus transactions using Load operations from an egress processor. These Global reads deliver the values of these registers instantaneously. Their values typically vary (very rapidly and widely) when the QMU is active. Some form of sampling and time-averaging is necessary to get an accurate sense of the congestion level.

The QMU makes available status information about the processors level of congestion. That status information is contained in the *Free_Descriptor_Buffer_List* and *Dyn_Descriptor_Pooln_Usage* registers. These two (2) registers can be read using Global reads when the QMU is on-line to access the status information. Refer to "Free_Descriptor_Buffer_List Register (QMU Status Function)" on page 510, "Dyn_Descriptor_Pool0_Usage Register (QMU Status Function)" on page 511, and Table 157 on page 511.

Processors can also read the instantaneous contents of the various statistics counters using Loads over the Global Bus. There are sixteen (16) QMU Statistics registers that are available. Refer to "QMU Registers" on page 355.

Types of Transactions

The QMU supports five (5) Queuing functions. The different functions are initiated by CPs or the XP using the Multi-Use Control Blocks by just changing the fields. Multi-Use Control Blocks use the following registers: WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB0_SDP_Addr; and RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB0_SDP_Addr. Refer to Table 78 on page 337, Table 79 on page 338, and Table 80 on page 339.

Table 78 Multi-Use Control Blocks (for Wr and Rd)

Mode	Category	Function	Fields Used	Details
• CP to/from QMU • XP to/from QMU	Queue Management Transactions	Configure Queue	Mail Box#, Queue#, Cmd, PoolID	See "Configure Queue Operation" on page 340.
		Queue Status		See "Queue Status Operation" on page 342.
		Unicast Enqueue		See "Unicast Enqueue Operation" on page 344.
		Multicast Enqueue	Mail Box#, QueueLevel#, Cmd, PoolID	See "Multicast Enqueue Operation" on page 346.
		Dequeue	Mail Box#, Queue#, Cmd, PoolID	See "Dequeue Operation" on page 348.

Table 79 WrCB0_ Variables per Field for QMU

<table>
<tr><th>Register</th><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>WrCB0_DMA_Addr</td><td>PoolID</td><td>20:16</td><td>PoolID —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Configure Queue</td><td>31</td></tr><tr><td>Unicast Enqueue</td><td></td></tr><tr><td>Multicast Enqueue</td><td></td></tr></table></td></tr>
<tr><td rowspan="4">WrCB0_Sys_Addr</td><td>Mail Box#</td><td>27:24</td><td>Mail Box Number —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Configure Queue</td><td rowspan="3">Enter the number of the processor’s mailbox. Legal range=0 to 16. Generally, the initiating processors ID.</td></tr><tr><td>Unicast Enqueue</td></tr><tr><td>Multicast Enqueue</td></tr></table></td></tr>
<tr><td>Queue#</td><td>20:12</td><td>Queue Number —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Configure Queue</td><td>Queue to configure. Legal range=0 to 511.</td></tr><tr><td>Unicast Enqueue</td><td>Queue to write to. Legal range=0 to 511.</td></tr></table></td></tr>
<tr><td>QLevel#</td><td>18:16</td><td>Queue Level Number —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Multicast Enqueue</td><td>Index to the Queue Number to write to. Legal range=0 to 7.</td></tr></table></td></tr>
<tr><td>CMD</td><td>10:8</td><td>Command —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Configure Queue</td><td>1</td></tr><tr><td>Unicast Enqueue</td><td>4</td></tr><tr><td>Multicast Enqueue</td><td>6</td></tr></table></td></tr>
</table>

Table 80 RdCB0_ Variables per Field for QMU

<table>
<tr><th>Register</th><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>RdCB0_DMA_Addr</td><td>PoolID</td><td>20:16</td><td>PoolID —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Queue Status</td><td rowspan="2">31</td></tr><tr><td>Dequeue</td></tr></table></td></tr>
<tr><td rowspan="3">RdCB0_Sys_Addr</td><td>Mail Box#</td><td>27:24</td><td>Mail Box Number —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Queue Status</td><td rowspan="2">Enter the number of the processor’s mailbox. Legal range=0 to 16. Generally, the initiating processors ID.</td></tr><tr><td>Dequeue</td></tr></table></td></tr>
<tr><td>Queue#</td><td>20:12</td><td>Queue Number—<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Queue Status</td><td>Queue Status being read. Legal range=0 to 511.</td></tr><tr><td>Dequeue</td><td>Queue being Dequeued. Legal range=0 to 511.</td></tr></table></td></tr>
<tr><td>CMD</td><td>10:8</td><td>Command —<table><tr><th>Operation Type</th><th>Value</th></tr><tr><td>Queue Status</td><td>2</td></tr><tr><td>Dequeue</td><td>5</td></tr></table></td></tr>
</table>

Queue Management Transactions

Queue transactions consist of five (5) different functions (Configure Queue, Queue Status, Unicast Enqueue, Multicast Enqueue, and Dequeue). Queue transactions are invoked using Control Blocks (WrCB0, and RdCB0).

Queue Transaction Functions (Operation and Examples)

Each is described here along with examples.

Configure Queue Operation

Configure Queue uses a control block (WrCB0) to send configuration information from the (DMEM) of either the requesting CP or XP to the QMU's Queue Management Engine (QME).

Configure Queue Example

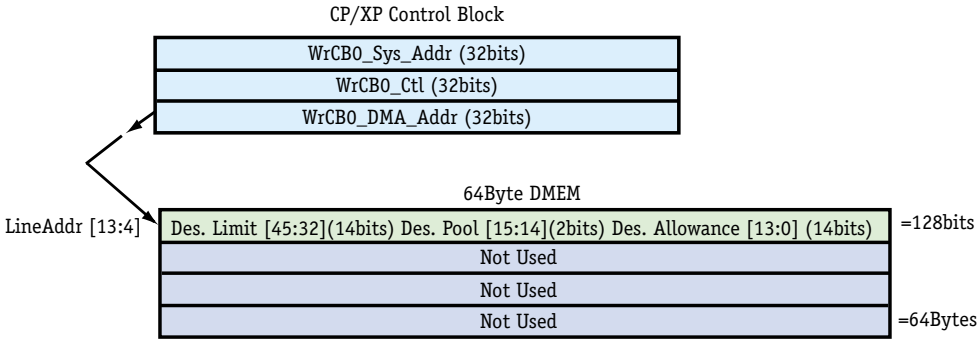
The bits for WrCB0_Sys_Addr, WrCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 81 on page 340.

Table 81 WrCB0_ Settings for Configure Queue

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. **Length (Bytes)** 64
WrCB0_Sys_Addr	Mail Box#	27:24	**Mail Box Number** — Enter the number of the processor's mailbox. Legal range=0 to 16.
	Queue#	20:12	**Queue Number** — Queue to write to configure. Legal range=0 to 511.
	Cmd	10:8	**Command** — Enter 1 for Configure Queue Operation.
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 31 for Queue Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 64Byte line address for DMA.

The WrCB0_DMA_Addr bits [13:0] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. The descriptor limit [45:32] (14bits), descriptor pool [15:14] (2bits) and descriptor allowance [13:0] (14bits) are located in the first 128bit line inside the 64Byte DMEM as shown in Figure 73 on page 341. The second, third and fourth 128bit lines are not used.

Figure 73 Configure Queue Implementation



Queue Status Operation

Queue Status uses a control block (RdCB0) to read a single queue's length and weight from the QMU into the (DMEM) of the requesting CP, or XP.

Queue Status Example

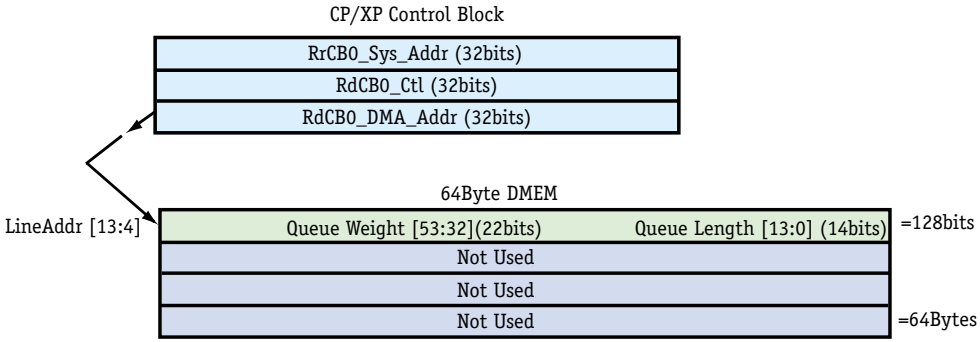
The bits for RdCB0_Sys_Addr, RdCB0_Ctl and RdCB0_DMA_Addr are set as shown in Table 82 on page 342.

Table 82 RdCB0_ Settings for Queue Status

Register	Field Name	Bit Position	Description
RdCB0_Ctl	Length	13:4	**Length** — Length of DMA transfer in Bytes. **Length (Bytes)** 64
RdCB0_Sys_Addr	MailBox#	27:24	**Mail Box Number** — Enter the number of the processor's mailbox. Legal range=0 to 16.
	Queue#	20:12	**Queue Number** — Queue Status being read. Legal range=0 to 511.
	Cmd	10:8	**Command** — Enter 2 for Queue Status Operation.
RdCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 31 for Queue Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 64Byte line address for DMA.

The RdCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. The queue's weight [53:32] (22bits) and its length [13:0] (14bits) are located in the first 128bit line inside the 64Byte DMEM as shown in Figure 74 on page 343. The second, third and fourth 128bit lines are not used.

Figure 74 Queue Status Implementation



Unicast Enqueue Operation

Unicast Enqueue uses a control block (WrCB0) to write the Descriptor data into a queue in the QMU's (SRAM) from the (DMEM) of either the requesting CP or XP.

Unicast Enqueue Example

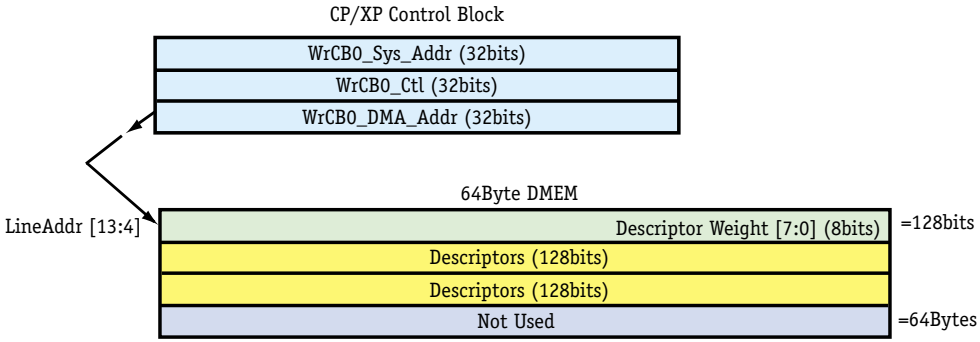
The bits for WrCB0_Sys_Addr, WrCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 83 on page 344.

Table 83 WrCB0_ Settings for Unicast Enqueue

Register/	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. **Length (Bytes)** 64
WrCB0_Sys_Addr	Mail Box#	27:24	**Mail Box Number** — Enter the number of the processor's mailbox. Legal range=0 to 16.
	Queue#	20:12	**Queue Number** — Queue to write to. Legal range=0 to 511.
	Cmd	10:8	**Command** — Enter 4 for Unicast Enqueue Operation.
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 31 for Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 64Byte line address for DMA.

The WrCB0_DMA_Addr bits [13:0] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. The descriptor weight [7:0] (8bits) is located in the first 128bit line followed with descriptors (depending on 12, 16, 24, or 32Byte sizes) located in the second and third 128bit lines inside the 64Byte DMEM as shown in Figure 75 on page 345. The fourth 128bit line is not used.

Figure 75 Unicast Enqueue Implementation



Multicast Enqueue Operation

Multicast Enqueue uses a control block (WrCB0) to write a Descriptor's data into multiple queues in the QMU's SRAM from the DMEM of either the requesting CP, or XP.

Multicast Enqueue Example

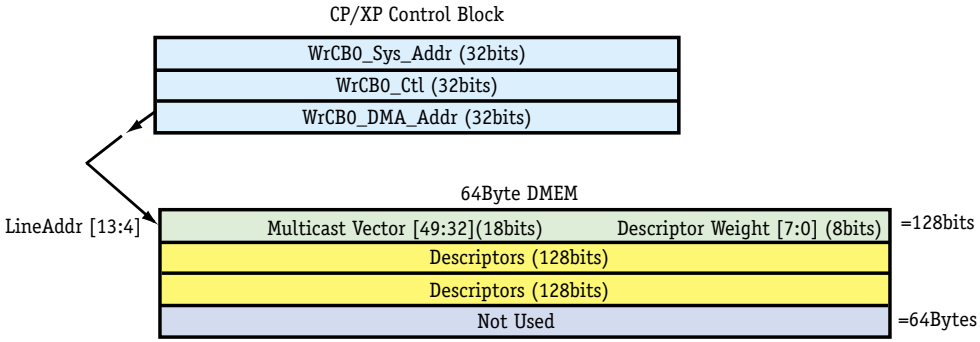
The bits for WrCB0_Sys_Addr, WrCB0_Ctl, and WrCB0_DMA_Addr are set as shown in Table 84 on page 346.

Table 84 WrCB0_ Settings for Multicast Enqueue

Register	Field Name	Bit Position	Description
WrCB0_Ctl	Length	13:0	**Length** — Length of DMA transfer in Bytes. **Length (Bytes)** 64
WrCB0_Sys_Addr	Mail Box#	27:24	**Mail Box Number** — Enter the number of the processor's mailbox. Legal range=0 to 16.
	QLevel#	18:16	**Queue Level Number** — Index to the Queue Number to write to. Legal range=0 to7.
	Cmd	10:8	**Command** — Enter 6 for Multicast Enqueue Operation.
WrCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 31 for Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 64Byte line address for DMA.

The WrCB0_DMA_Addr bits [13:0] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. The multicast vector [49:32] (18bits) and descriptor weight [7:0] (8bits) are located inside the first 128bit line followed with descriptors (12, 16, 24, or 32Byte sizes) located in the second and third 128bit lines inside the 64Byte DMEM as shown in Figure 76 on page 347. The fourth 128bit line is not used.

Figure 76 Multicast Enqueue Implementation



Dequeue Operation

Dequeue uses a control block (RdCB0) to read Descriptor data from a queue in the QMU's SRAM into the DMEM of the requesting CP, or XP. Dequeue frees a Descriptor Buffer from a queue.

Dequeue Example

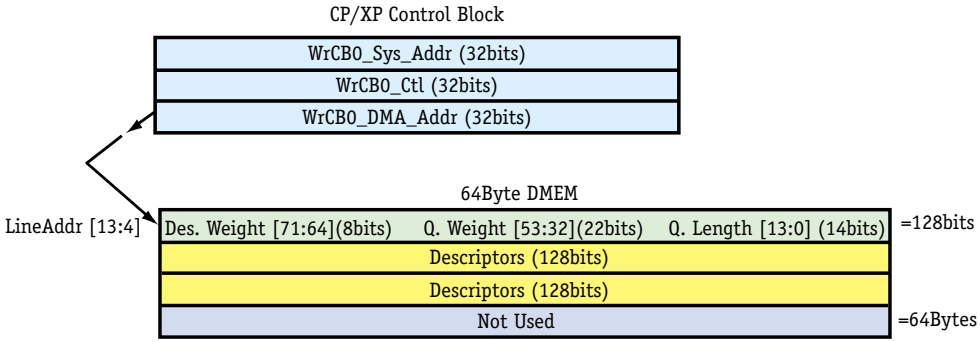
The bits for RdCB0_Sys_Addr, RdCB0_Ctl and RdCB0_DMA_Addr are set as shown in Table 85 on page 348.

Table 85 RdCB0_ Settings for Dequeue

Register	Field Name	Bit Position	Description
RdCB0_Ctl	Length	13:4	**Length** — Length of DMA transfer in Bytes. **Length (Bytes)** 64
RdCB0_Sys_Addr	MailBox#	27:24	**Mail Box Number** — Enter the number of the processor's mailbox. Legal range=0 to 16.
	Queue#	20:12	**Queue Number** — Queue being Dequeued. Legal range=0 to 511.
	Cmd	10:8	**Command** — Enter 5 for Dequeue Operation.
RdCB0_DMA_Addr	PoolID	20:16	**PoolID** — Enter 31 for Queue Operation.
	LineAddr	13:4	**DMEM Line Address** — DMEM 64Byte line address for DMA.

The RdCB0_DMA_Addr bits [13:4] *LineAddr* field refers to the Local DMEM buffer address that is generally 64Byte aligned. The descriptor weight [71:64] (8bits), queue weight [53:32] (22bits) and queue length [13:0] (14bits) are located inside the first 128bit line followed with descriptors (depending on 12, 16, 24, or 32Byte sizes) located in the second and third 128bit lines inside the 64Byte DMEM as shown in Figure 77 on page 349. The fourth 128bit line is not used.

Figure 77 Dequeue Implementation



QMU Multicast Support (Non-System Level)

Multicast enqueue operations take a single multicast descriptor from the DMEM of a processor (CPs, or XP) and copy it to the designated target queue numbers. To get from the DMEM to the selected targeted queue number requires certain items. Two (2) basic formulas can be used to illustrate the required items. They are:

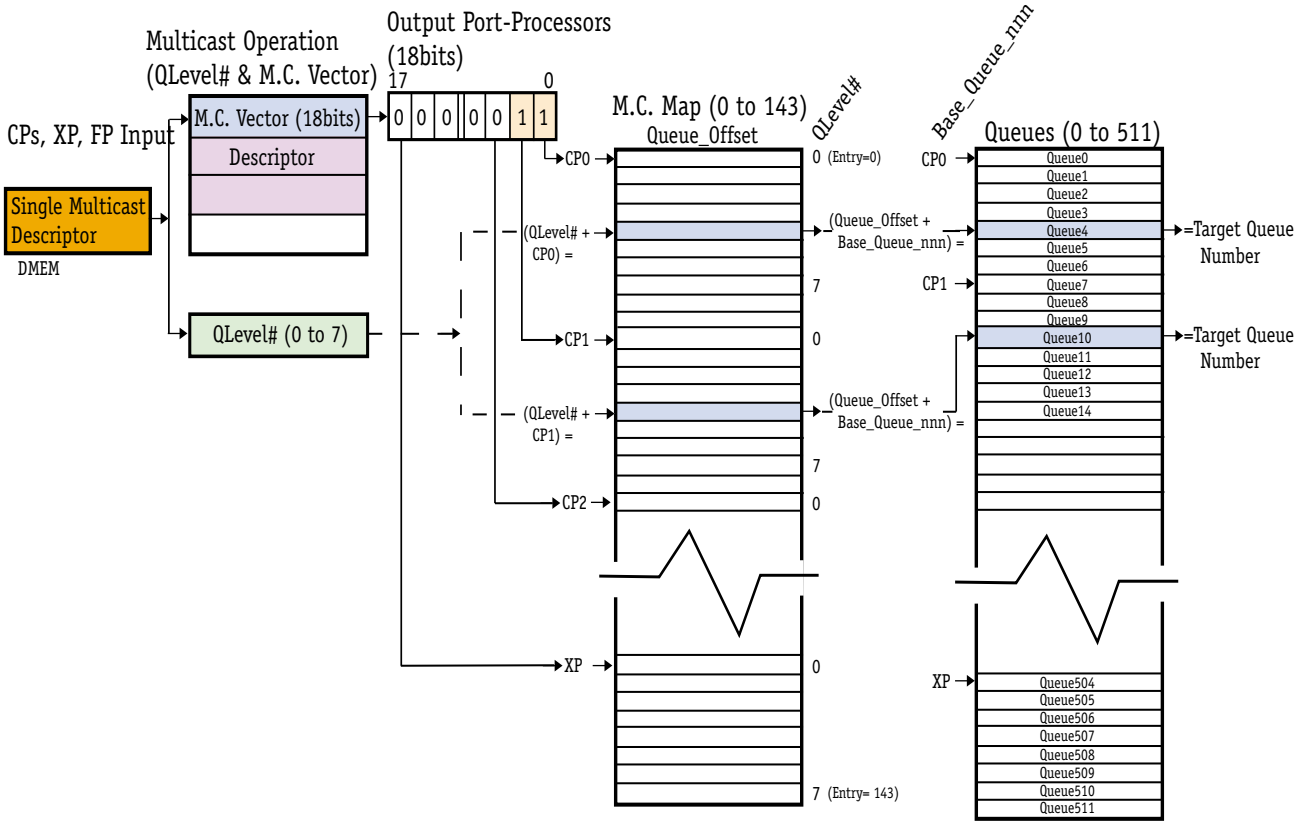
- (QLevel# + (Multicast Vector Bit * 8)) = Queue_Offset
- (Queue_Offset + Base_Queue_nnn) = Target Queue Number

The multicast operation is implemented using the Control Blocks (WrCB0_). Specifically, the *WrCB0_Sys_Addr* register, bits [18:16] *QLevel#* field that are used to select the queue level number (0 to 7). This allows eight (8) multicast queuing levels to be mapped to as many as eight (8) queues for each processor. Processors can have < 8 multicast-enabled queues if desired. Additionally, the *multicast vector* (18bits), a bitmask, determines which processors (CPs, XP, or FP) are the output ports for this multicast operation. Generally, the multicast vector is fetched from a multicast table in the TLU by the processor that built the descriptor.

The multicast vector's 18bits correspond to the processors. For example, 0bit= CP0, 1bit= CP1, 15bit= CP15, 16bit= FP, 17bit=XP. These two (2) items combined provide the *Queue_Offset* address (0 to 143) into the Multicast Destination Table. Use the *Multicast_Destination0* to *Multicast_Destination143* registers bits [6:0] *Queue_Offset* field for selecting applicable processors (CPs, XP, or FP) for programming output ports using their ID's (0=CP0, 1=CP1, 15=CP15, 16=FP, 17=XP) and the applicable queue level number (0 to 7) (QLevel#). Refer to Table 156 on page 508 for a complete listing of all the multicast mapping addresses including: processor ID numbers, queue level number, and queue offset address.

One of the QMU's setup steps is to map individual queue numbers (0 to 511) to their respective processors (CPs, XP, or FP) using the *Base_Queue_CP0* to *Base_Queue_CP15*, *Base_Queue_FP*, and *Base_Queue_XP* registers. This already provides the association between the queue numbers and their mapped processors. Take the *Queue_Offset* address result from the first formula and add the applicable base queue address (*Base_Queue_CPnn*, *Base_Queue_XP*, or *Base_Queue_FP*) to determine the targeted queue number that the single multicast descriptor needs to be copied for each output port.

Figure 78 Multicast Enqueue Operation and Mapping Example



1) (QLevel# + (M.C. Vector Bit * 8)) = Queue_Offset
2) (Queue_Offet + Base_Queue_nnn) = Target Queue Number

Multicast elaboration is performed by copying the descriptor into a descriptor buffer for each of the appropriate output queues. Other than the availability of *Free_Descriptor_Buffer_List* and *Dyn_Descriptor_Pooln_Usage*, the QMU imposes no limit on the number of descriptors that can simultaneously be in the queues as the result of multicast operations. However, the BMU has a fixed number (1024 8bit) of Multi-Use Counters (MUC) that it can maintain that can be used to track buffer accesses when a buffer has multiple targets (CPs, XP or FP). Refer to "Multi-Use Counter (MUC) Management Transactions" on page 233. Each of the sixteen (16) CPs, the XP, and the FP can receive a copy of a descriptor. This totals a maximum of eighteen (18) copies. However, the external switching fabric must be capable of multicast replication for multiple fabric ports from a single multicast queue.

Multicast Operations Success or Failure

Descriptors enqueued in the QMU correspond to payload buffers located in the BMU. If a descriptor does not make it to a destination processor, the corresponding payload buffer is effectively lost as well because it is not deallocated. In a multicast operation, a payload buffer is assigned a destination count equal to the number of destination processors to receive a copy of the descriptor. When all the destination processors have received their copy of the descriptor, the payload buffer is released because the destination-count reaches zero (0).

It is critical that the QMU succeed at multicasting the descriptor to all the destination processors. If it does not, one or more processors never receive a copy of the buffer descriptor, and the payload buffer's destination count never reaches zero (0) and the payload buffer is never released. This would cause a "memory leak". If the QMU cannot successfully enqueue a descriptor to all the specified destinations, it rejects the entire multicast enqueue request. When a multicast enqueue request fails, the requesting processor can either retry the multicast enqueue, or drop the payload.

Multicast Operation Throughput Considerations

An N-way multicast enqueue operation cannot take longer than N unicast enqueues if the C-5 NP is to maintain its throughput. Therefore, the QMU cannot afford to take the time to visit each target queue to determine if its length is over its limit before actually beginning to do the series of enqueues. Since the QMU cannot check ahead of time and it cannot fail to enqueue to each queue, it cannot check the queue limit while it is doing the enqueues. This means that a multicast operation can push a queue above its length limit.

Before the QMU begins the individual enqueues of a multicast enqueue operation, it checks that there are enough free descriptor buffers (*Free_Descriptor_Buffer_List* register) and enough room within the dynamic pool (*Dyn_Des_Usage_Lim_Pooln* register) to complete the series of enqueues. As with the queue length limit, the QMU cannot know ahead of actually doing the individual enqueues how many of the target queues will be above their allowances and therefore need a dynamic descriptor buffer. The initial pool usage check insures that there will be enough dynamic buffers no matter what. However, the pool usage check might find that there are not enough buffer credits left if all the target queues need one, when, in actuality, there are enough pool credits available for the number of queues that actually need them. In this case, the multicast operation will be rejected when it could have succeeded.

Because the QMU only has time to check the dynamic pool of the first queue, all member queues of a multicast group must share the same dynamic buffer pool.

Queue Levels Supported in Multicast Operations

The QMU supports eight (8) levels (0 to 7) of multicast queuing. The purpose of using levels is to assign a specific queue to each level in order to implement a multicast operation. In a multicast operation, a single payload descriptor is simply copied to each of the targeted queues specified using the Queue Level Number field (QLevel#) in the *WrCB0_Sys_Addr*. The Queue Level Number is an index to the Queue Number to write to.

Although levels are available only through multicast commands, unicast descriptors can also use queuing levels if enqueued via a multicast command with a fanout of 1. A multicast descriptor specifies only the target processors and the queuing level; the QMU then maps that onto the specific queue corresponding to the given level for each target processor. The presumption is that a given descriptor is associated with a class-of-service that applies at each output port (CPs, XP, or FP). When different queuing levels are needed at different output ports, multiple multicast enqueues can be issued so that each one covers the destinations at each of the required queuing levels.

Multicast mapping uses a configurable look-up table to store the queue number for each combination of processor and queuing level. For eighteen (18) processors and eight (8) queueing levels, this requires one-hundred-forty-four (144) entries. Each entry holds a 7bit queue offset that when added to the processor's base queue number, gives the number of the target queue. With this scheme, each processor with eight (8) or more queues can have up to eight (8) queues for transmitting multicast traffic. These queues could also receive unicast traffic. Target processors can accept < 8 queues. Refer to "Multicast_Destination0 to Multicast_Destination143 Registers (QMU Configuration Function)" on page 507 and Table 156 on page 508.

If a processor is to accept multicast traffic in < 8 queues, each of its eight (8) look-up table entries must still be configured to point at one (1) of its queues. For example, if there are to be only two (2) queues receiving multicast traffic, at queue offsets 3 and 4, the eight (8) look-up table entries could be programmed. Refer to Table 86 on page 353.

Table 86 Multicast Queue Mapping for <8 Queues Example

M. C. QLevel#'s Points to Single Queue	SIngle Queue's Receive of M. C. Traffic	Single Queue Target#
QLevel0	Queue_Offset3	Queue25
QLevel1	Queue_Offset3	
QLevel2	Queue_Offset3	
QLevel3	Queue_Offset3	

Table 86 Multicast Queue Mapping for <8 Queues Example (continued)

M. C. QLevel#'s Points to Single Queue	SIngle Queue's Receive of M. C. Traffic	Single Queue Target#
QLevel4	Queue_Offset4	Queue83
QLevel5	Queue_Offset4	
QLevel6	Queue_Offset4	
QLevel7	Queue_Offset4	

The actual mapping depends on the particular relationship of queuing levels to service classes in each application.

Multicasting to the Fabric Processor

The FP multicast vector bit in the descriptor is set by the originating processor and directs the descriptor to one FP queue at the appropriate queuing level.

QMU Configuration Space

The QMU has memory-mapped Configuration Space that contains a number of registers. The registers are used for four (4) purposes: mapping of queues to CPs, XP and FP, configuration of the QMU, QMU debugging, and collecting QMU statistics. Refer to Table 87 on page 355.

Table 87 QMU Registers

QMU Register Types	Register Function	Specific Register Details
CP's Queue Mapping	These registers specify the base address for a CP's queues.	See "Base_Queue_CP0 to Base_Queue_CP15 Registers (QMU CP's Queue Allocation Function)" on page 501.
XP's Queue Mapping	These registers specify the base address for a XP's queues.	See "Base_Queue_XP Register (QMU XP's Queue Allocation Function)" on page 502.
FP's Queue Mapping	These registers specify the base address for a FP's queues.	See "Base_Queue_FP Register (QMU FP's Queue Allocation Function)" on page 502.
QMU Configuration	Specifies the number of descriptor buffers to be available in the QMU.	See "Num_Descriptors Register (QMU Configuration Function)" on page 503.
	Specifies the maximum number of descriptors that can be enqueued dynamically to the queues associated with Pool*n*.	See "Dyn_Des_Usage_Lim_Pool0 Register (QMU Configuration Function)" on page 503.
	Specifies the operating mode of the QMU.	See "Operation_Mode Register (QMU Configuration Function)" on page 504.
	Specifies the size of the data stored for each descriptor in an encoded form.	See "Descriptor_Size Register (QMU Configuration Function)" on page 504.
	Provides the mapping of the multicast destination port and queue level to a target queue number for each leaf of a multicast elaboration.	See "Multicast_Destination0 to Multicast_Destination143 Registers (QMU Configuration Function)" on page 507.
QMU Status	Designates the total number of free descriptors.	See "Free_Descriptor_Buffer_List Register (QMU Status Function)" on page 510.

Table 87 QMU Registers (continued)

QMU Register Types	Register Function	Specific Register Details
QMU Status (continued)	Designates how many buffers are in use in Pool*n*.	See "Dyn_Descriptor_Pool0_Usage Register (QMU Status Function)" on page 511.
QMU Statistics	Count of Queue Configuration Operations.	See "Config_Q_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Read Status operations.	See "Rd_Q_Status_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Unicast Enqueues from the CPs.	See "CP_Uni_Enq_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Multicast Enqueues from the CPs.	See "CP_Multi_Enq_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Total Multicast Enqueues Targets from the CPs.	See "CP_Multi_Enq_Target_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Dequeue operations from the CPs.	See "CP_Dequeue_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Unicast Enqueues from the FP.	See "FP_Uni_Enq_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Multicast Enqueues from the FP.	See "FP_Multi_Enq_Cnt Register (QMU Statistics Function)" on page 505.
	Count of Total Multicast Enqueues Targets from the FP.	See "FP_Multi_Enq_Target_Cnt Register (QMU Statistics Function)" on page 506.
	Count of Dequeue Operations from the FP.	See "FP_Dequeue_Cnt Register (QMU Statistics Function)" on page 506.
	Count of QMU Idle Clock Cycles.	See "QMU_Idle_Cycles Register (QMU Statistics Function)" on page 506.
	Count of Payload NACKs.	See "Payload_NACK_Cnt Register (QMU Statistics Function)" on page 506.

Table 87 QMU Registers (continued)

QMU Register Types	Register Function	Specific Register Details
QMU Statistics (continued)	Count of Global NACKs.	See "Global_NACK_Cnt Register (QMU Statistics Function)" on page 506.
	Count of Payload Read Failures.	See "Payload_Read_Failures_Cnt Register (QMU Statistics Function)" on page 506.
	Count of Command Processor Errors, illegal opcodes and out of range queue numbers.	See "Cmd_Processor_Err_Cnt Register (QMU Statistics Function)" on page 506.
	Count of Queue Engine Errors.	See "Q_Engine_Err_Cnt Register (QMU Statistics Function)" on page 507.

For complete details about specific registers go to their reference. Refer to "Queue Management Unit (QMU) Configuration Registers" on page 498.

QMU Setup

The QMU must be initialized to operate. Using Global Bus operations, various registers and memories must be written with appropriate values to allow the QMU to function. Initialization must be completed before the QMU can be put online. See Table 87 on page 355 for more complete descriptions of the registers listed below.

As with most C-5 NP components, QMU initialization is completely handled by C-Port's programming interface, so that programmers never directly deal with the registers listed in this section.

Initializing the QMU involves writing the following:

1 Configure the pools using the following registers:

- *Operation_Mode* register (Internal=0x1)
- *Descriptor_Size* register (12, 16, 24, 32Bytes)
- *Num_Descriptors* register (0 to 16, 384)
- *Dyn_Des_Usage_Lim_Pool0* to *Dyn_Des_Usage_Lim_Pool3* registers for each of the four (4) *Dyn_Descriptor_Pooln_Usage* registers

2 Map the queues to their applicable processors (CPs, XP, or FP):

This maps specific processors to specific queues to be used for Unicast Enqueue Operations, as well as mapping specific processors to specific queues to be used for Multicast Enqueue Operations.

- *Base_Queue_CP0 to Base_Queue_CP15*, *Base_Queue_XP* and *Base_Queue_FP* registers for each of the eighteen (18) processors need to be initialized.
- 144 mapping table entries need to be written for Multicast Enqueue Operations.

3 Establish the *Free_Descriptor_Buffer_List*:

- This requires coordinated initialization of the *Free Descriptor Buffer List* register (head pointer, tail pointer, and length) and the internal descriptor linkage-list pointers. If there are 16,384 descriptors buffers, 16,383 linkage pointers must be initialized to point one to the next. The last descriptor's buffer linkage pointer is a *don't-care*.

4 Set the queue parameters:

- Queue Length Allowance — The guaranteed minimum amount of allocated descriptor buffers of the (0 to 16K) range. Implemented using the Configure Queue Operation.
- Queue Length Limit — The guaranteed maximum number of allocated descriptor buffers a queue is allowed to hold of the (0 to 16K) range. In other words, Allowance is the minimum amount of the (0 to 16K) range, where as, Limit is the maximum of the (0 to 16K) range. Implemented using the Configure Queue Operation. Refer to "Configure Queue Operation" on page 340.

5 Set the queue Link-lists:

- The queue lengths must be initialized to zero. With a length of zero, the queue head and tail pointers are don't-cares.

6 Enable the *QMU_Run_Enable* bit:

- The QMU must be run enabled before it can process queuing operations. The *QMU_Run_Enable* register is written with a "1" to do this.

QMU Performance

The QMU uses pipelining, so that the execution latency of the queuing operations (i.e., the time between the request, at the QMU, until the descriptor is returned to the requester) is greater than the time interval between the beginning of the execution of the descriptors.

Execution Speed and Descriptor Size Relationship

The actual maximum speed achievable by the QMU is a function of both the descriptor size and the speed of the Queue Management Engine (QME). The Queue Management Engine (QME) uses seven (7) clock cycles to do a unicast enqueue, a dequeue, or a single destination of a multicast enqueue. The descriptor data storage/retrieval process requires two (2) core clock cycles for each word of four (4) descriptor bytes. The actual execution period is the larger of the two (2) of these intervals.

For example, 12Byte descriptors need six (6) core-clock cycles to be transferred to and from the external SRAM. Since the queuing-engine needs seven (7) clock cycles, the overall execution interval is seven (7) clock cycles. For 16Byte descriptors, the descriptor transfer rate is eight (8) core-clock cycles, so it is the dominant time for overall execution. With 24Byte descriptors, the execution interval is twelve (12) clock cycles. For 32Byte descriptors, the minimum execution rate is sixteen (16) clock cycles.

Table 88 Execution Rates Using a 200MHz Core-Clock Rate Example

Descriptor Size	Core-Clock Cycles	Queue Operations/Second
12Byte	7	28.6M
16Byte	8	25.0M
24Byte	12	16.7M
32Byte	16	12.5M

Multicast Support (System Level)

The C-5 NP supports multicasting Ethernet packets and multi-pointing ATM frames or cells. Several C-5 NP components are involved in the multicast process.

Multicast elaboration from the Fabric Processor (FP) to the QMU is not supported for this version of the C-5 NP.

Multicast Flow in the C-5 NP

The overall multicast transaction flow is described below. It has been separated into the receive and the transmit portions for clarity.

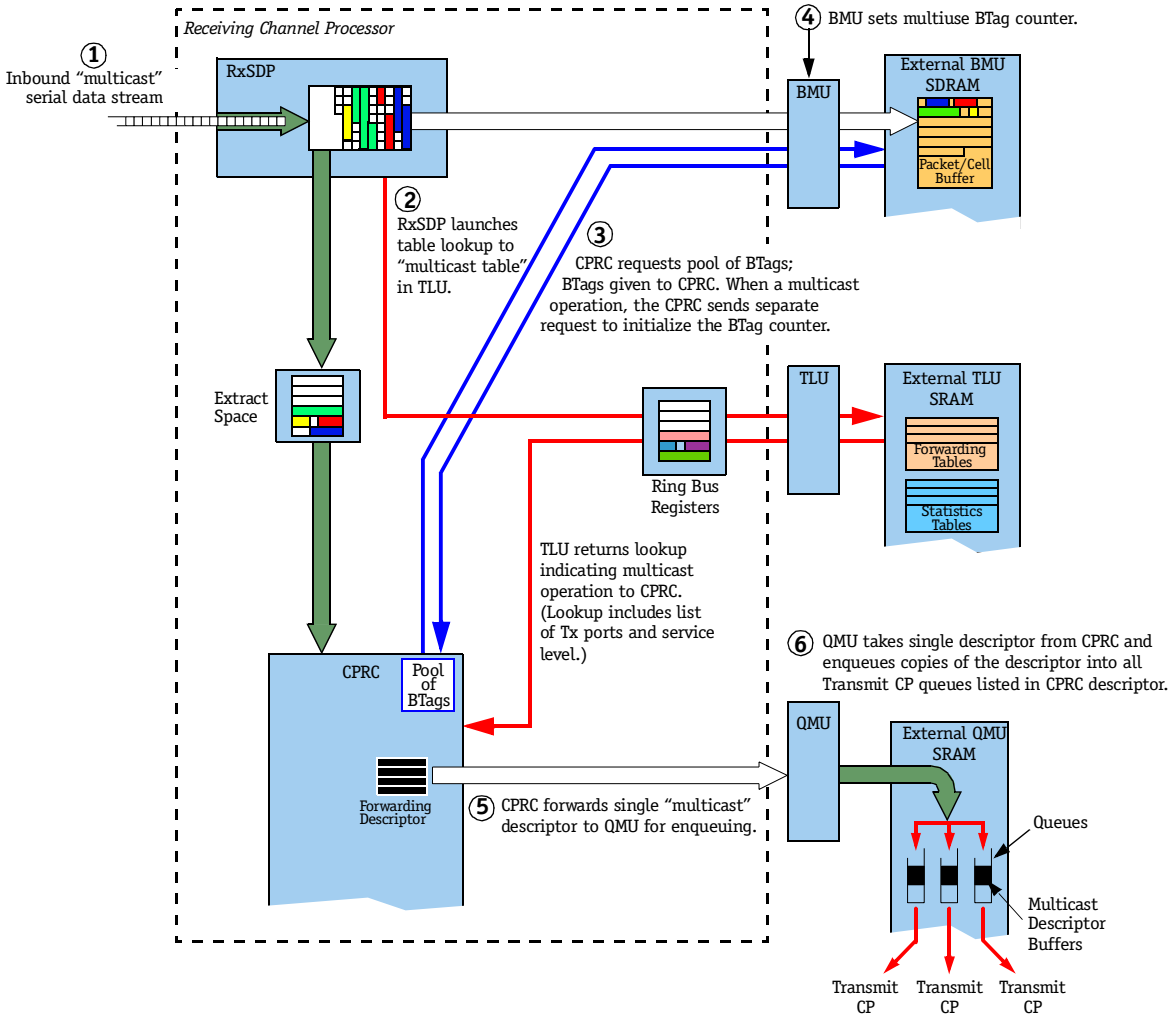
Multicast Receive Flow Transaction Process

The following describes the receive flow for a multicast transaction (see Figure 79).

1. RxSDP receives the "multicast/multipoint" packet/cell.
2. Based on a combination of SDP processing, table lookup, and CPRC processing, the CP determines that the packet/cell requires a multicast forwarding operation.
3. The CPRC requests and assigns a multiuse Buffer Tag (BTag) to the packet/cell and requests the BMU to associate a multi-use counter (MUC) BTag with the BTag.
4. The BMU assigns a multi-use counter (MUC) BTag (the count equals the number of transmit ports) and associates the counter with a buffer.
5. The CPRC can perform additional processing and then sends the descriptor to the QMU.
6. The QMU removes the multicast vector and queue level information and then tries to enqueue the descriptor to the specified output queues. It assesses the supply of descriptor buffers and if there are enough, it proceeds to do the enqueues.

   Each descriptor queue maintains an allocated descriptor count and an overall descriptor usage limit. The QMU checks the dynamic buffer pool for the first output queue, and if that pool has enough buffers to match the number of transmit ports, the QMU proceeds. Otherwise, it signals a failure of the enqueue operation and does not complete the multicast operation. Thus, all members of a multicast group should share the same dynamic buffer pool.

Figure 79 Multicast Application Receive Process Flow



Multicast Transmit Flow Transaction Process

The multicast transmit process is identical to a unicast operation. The following describes the transmit flow for a multicast transaction (see Figure 80).

1 The CPRC requests the descriptor in its assigned queue.

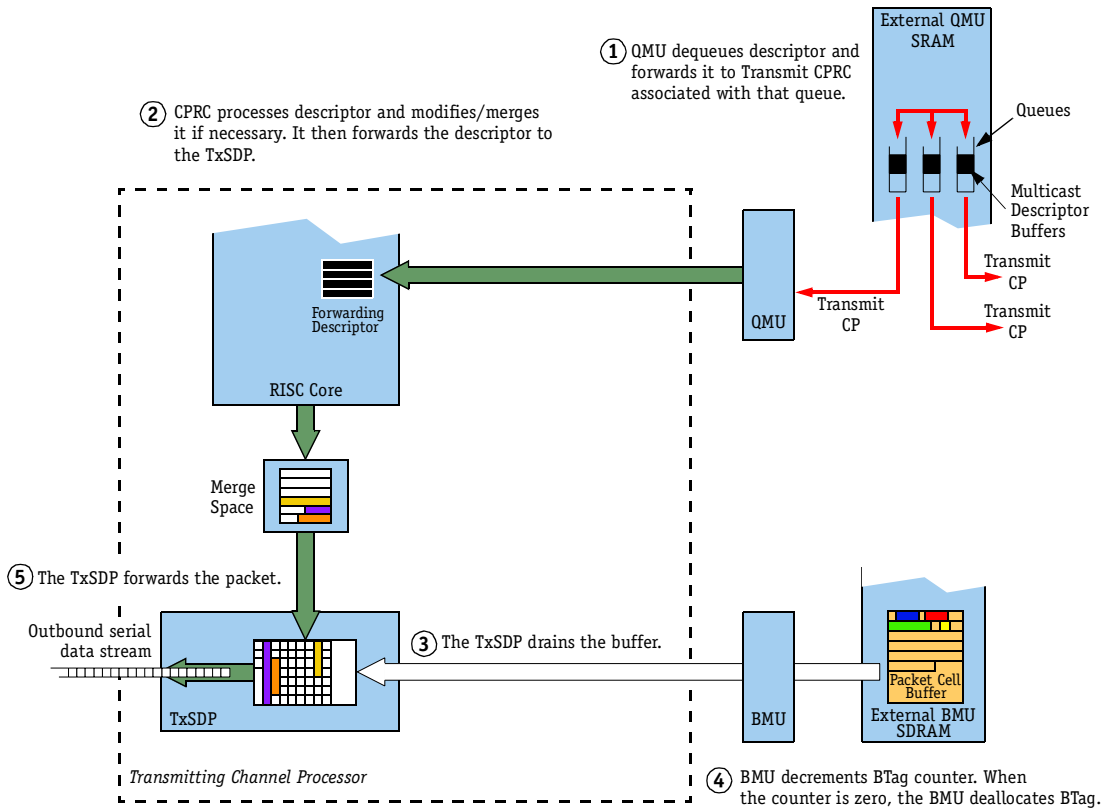
2 The CPRC processes the descriptor (if necessary) and then forwards it to the TxSDP.

3 The TxSDP drains the corresponding buffer and appends the header.

4 The CPRC drains the buffer and then sends a message to the BMU to decrement the counter. When the multi-use counter (MUC) BTag reaches 0 (all buffers associated with this multicast operation have been drained), the BTag is deallocated.

5 The TxSDP forwards the packet. Each port sends the packet/cell out when it is able, based on the amount of traffic that is queued for that port.

Figure 80 Multicast Application Transmit Process Flow



Chapter 8

Internal Buses

Chapter Overview

This chapter covers the following topics:

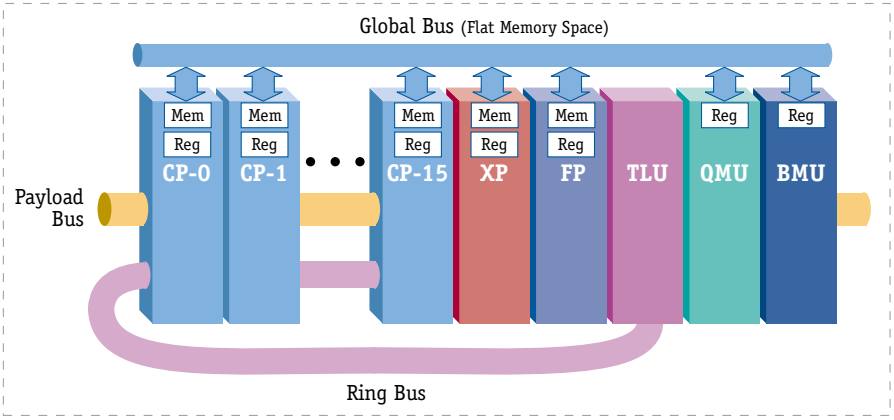
Internal Buses Overview

The C-5 NP contains three (3) independent data buses that provide internal communication paths between the C-5 NP's eighteen (18) processors (16CPs, XP, and FP) and three (3) coprocessors (TLU, QMU, and BMU), thus supporting concurrent processing. Refer to Table 89 on page 366. In addition, Figure 81 on page 366 shows the internal buses.

Table 89 C-5 NP Interconnect Components

Item	Device Type	Function
Payload Bus	Slotted, multichannel, shared, arbitrated bus	Carries payload data and payload descriptors between the processors and the BMU and QMU.
Ring Bus	Slotted ring-topology bus	Provides bounded latency transactions between the processors and the TLU. It also supports inter-processor communication.
Global Bus	Slotted, multichannel, shared, arbitrated bus	Supports inter-processor communication via a conventional flat memory-mapped addressing scheme.

Figure 81 Internal Custom Buses



Internal Buses Characteristics

The three (3) buses have the bandwidth, bus width, and transfer size characteristics defined in Table 90 on page 367.

Table 90 Bus Characteristics

Bus	Bandwidth	Bus Width	Transfer Size
Payload Bus	34.1Gbps	128bits	64Bytes
Ring Bus	21.2Gbps	64bits	8Byte to 32Bytes
Global Bus	4.2Gbps	32bits	4Bytes

Payload Bus Overview

The Payload Bus is a slotted, multichannel, shared, arbitrated bus that provides a high bandwidth path for C-5 NP's (16CPs, XP, and FP) to shared services in the BMU and QMU. It has a guaranteed arbitration latency to satisfy CP programming constraints, a retry feature, and a bus acknowledgment to indicate when a transaction is complete.

Payload Bus Operation

The Payload Bus uses a 128bit wide data path in four-cycle bursts to transfer up to 64 Bytes of payload data, descriptors, buffer tags, and other information to or from a processor on each of two (2) independent channels. To achieve high bus utilization, operations are pipelined and reads are split into a read request and a write response. Typical payload operations are described in Table 91 on page 368.

Table 91 Typical Payload Operations

Operation	Type of Information	Quantity/PDU
Rx Transactions		
Payload read	BTags (32 BTags are passed together)	1 per 32 PDUs
Payload write	Data	PDU size/64Bytes
Payload write	Descriptor	1
Tx Transactions		
Payload read	Descriptor	1
Payload read	Data	PDU size/64Bytes
Payload write	BTag	1

Payload Bus Latency

The Payload Bus arbitrates differently for the FP than for other clients (CPs and XP). This behavior is configurable by a *ZBFP* bit [9] in the *XP Miscellaneous Control* register. Refer to "XP Miscellaneous Control Register (XP Configuration Function)" on page 476. Setting this bit provides the FP with three (3) additional slots on the Payload Bus. Thus, you can optimize for greater FP access to the Payload Bus by setting this bit to 1, *or* optimize for better CP/XP access to the Payload Bus by setting this bit to 0 (the default configuration).

Payload Bus Latency (Default Mode)

In default mode (*ZBFP* = 0), the bus assigns/reserves one (1) bus slot for each processor. The default mode latency is shown in Table 92 on page 369.

Table 92 Payload Bus Arbitration Delay in Default Mode

Latency	Reads	Writes
Minimum	10 cycles	10 cycles
Maximum	CPs, XP, and FP = 110 cycles	CPs, XP, and FP = 110 cycles

Payload Bus Latency (FP Mode)

In FP mode (ZBFP = 1), additional bus slots are allocated to the TxFP for reads and to the RxFP for writes. This ensures that the FP can maintain a high data flow rate to the fabric. The FP mode latency is shown in Table 93 on page 369.

Table 93 Payload Bus Arbitration Delay in FP Mode

Latency	Reads	Writes
Minimum	10 cycles	10 cycles
Maximum	• TxFP = 40 cycles • CPs, XP, and RxFP = 140 cycles	• RxFP = 40 cycles • CPs, XP, and TxFP = 140 cycles

Ring Bus Overview

The C-5 NP implements a ring-topology bus for communication between the TLU and the eighteen (18) processors (16CPs, XP, and FP), each of which is a *node* on the ring. It uses a 64bit wide data path and is clocked at the C-5 NP core clock rate. The Ring Bus supports the following types of message transactions:

- Unacknowledged transaction
- Hardware acknowledged transaction
- Software acknowledged transaction

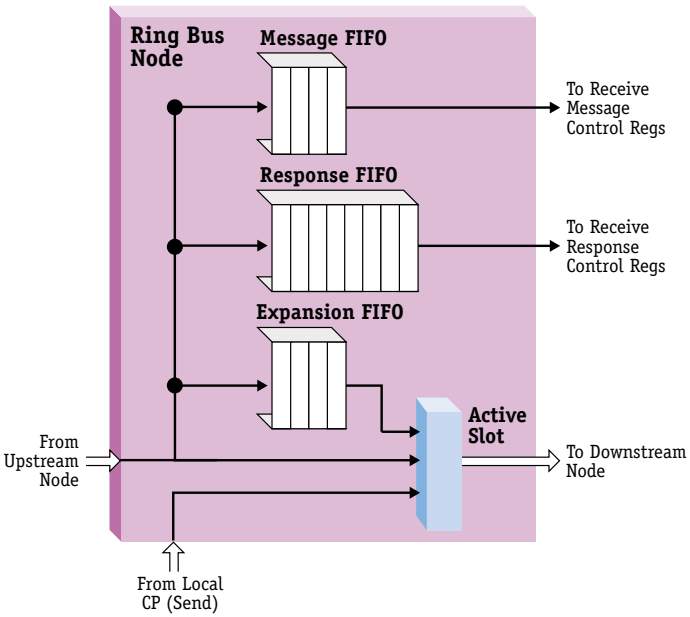
Ring Bus Major Components

A Ring Bus node consists of four (4) items. Refer to Table 94 on page 370 and Figure 82 on page 371.

Table 94 Ring Bus Components

Item	Function
Message FIFO	To pass messages to the *Receive Message* registers.
Response FIFO	To pass messages to the *Receive Response* registers.
Expansion FIFO	Where messages/responses passed from upstream are temporarily held when the active slot is busy.
Fixed-Size Active Slot	Where the upstream messages/responses or the local node's messages/responses are forwarded on the Ring Bus to the downstream node. The active slot comprises one (1) clock cycle, and the Ring Bus supports simultaneous node transmission and reception on each clock cycle.

Figure 82 Ring Bus Node Block Diagram



Ring Bus Node Operation

A Ring Bus node can perform the following actions:

- Sending Downstream
  - Send a new message or response to a downstream node.
- Receiving from Upstream
  - Receive a message destined for the local node.
  - Receive a response destined for the local node.
  - Pass through a message or response to a downstream node.

A node can send on the Ring Bus as long as the active slot (the slot currently available to the node) is free. Table 95 on page 372 lists the Ring Bus node IDs for the CPs, XP, FP, and TLU.

Table 95 Ring Bus Node IDs

Unit	Node ID	Unit	Node ID
CP0	0	CP10	10
CP1	1	CP11	11
CP2	2	CP12	12
CP3	3	CP13	13
CP4	4	CP14	14
CP5	5	CP15	15
CP6	6	XP	24
CP7	7	FP*	30
CP8	8	TLU	31
CP9	9		

* Transmit only. The FP cannot read messages on the Ring Bus. Thus any messages sent to the FP cannot be removed from the Ring Bus, eventually filling up the Ring Bus.

Sending Downstream

A local node uses its active slot to send downstream. When the local node is in the process of sending and then receives an upstream message targeted for a downstream node, the upstream message is placed in one of the local node's four (4) Expansion FIFO slots until the local node completes sending. Then the node passes the upstream message to the next node on the bus.

Contiguous "multi-slot" messages can be transmitted on the Ring Bus. Multi-slot messages are treated as one (1) message and are not divided as they move on the Ring Bus. The local node can send contiguous "multi-slot" messages (2 or 4 slots in length) as long as there are sufficient slots in the Expansion FIFO to hold the upstream messages. For example, if the local node wants to send a message requiring two slots, there must be two slots available in the Expansion FIFO. A four (4) slot message requires the expansion FIFO to be completely empty.

Receiving from Upstream

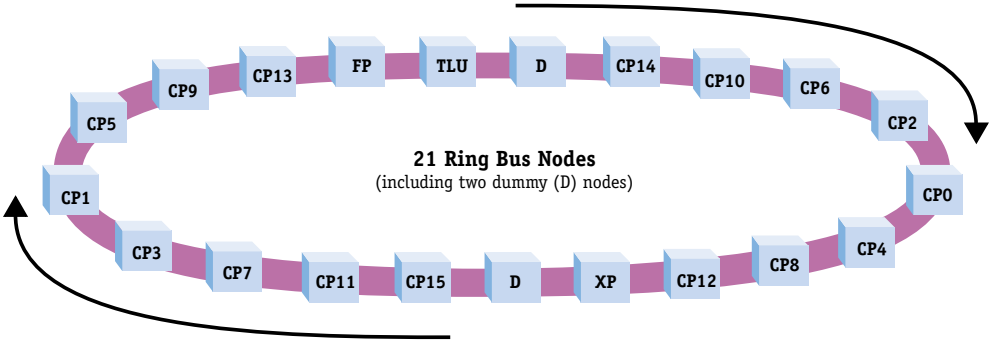
When a node receives a message or response from an upstream neighbor, it processes that message or response in one of three (3) ways:

- If a message is destined for the local node, it is forwarded through the four-slot (4) Message FIFO to the *Receive Message* registers. When the program reads these registers, the FIFO is popped. Refer to "Ring Bus (Rx) Receive Message Registers" on page 375.
- If a response is destined for this local node, it is forwarded through the eight-slot (8) Response FIFO to the *Receive Response* registers. A sequence number in the response dictates how the *Receive Response* registers are filled. If the target receive response block is already used in the register, then the FIFO can become blocked, possibly filling up the entire ring if incoming messages continue. The program must clear the *Receive Response* Register. Refer to "Ring Bus Receive (Rx) Response Registers" on page 375.
- If the node is simply forwarding a message/response downstream, it can send if the local node is not trying to send a message simultaneously. If its local node is sending a message, then the upstream message/response is placed in the local node's four-slot (4) Expansion FIFO. If this FIFO reaches capacity, the first item on the stack takes priority over the local node sending its own message/response and gets forwarded to the active slot.

Ring Bus Latency

When describing the Ring Bus's latency, it is important to understand that a "complete" transaction usually requires a round trip of the entire Ring Bus. For example, CP13 is located two nodes upstream from the TLU. If CP13 sends a request to the TLU, the "request" latency is two (2) clock cycles assuming that the TLU node is not busy. However, since the Ring Bus is unidirectional, the minimum latency to return data from the TLU to CP13 is 19 clock cycles. Thus round trip latency is 21 cycles, best case. Refer to Figure 83 on page 374.

Figure 83 Nodes on the Ring Bus



Because of the unidirectional nature of the Ring Bus and the fact that transactions on the Ring Bus are usually request/response, latency is not affected by which node delivers messages to the ring first. Therefore, the position of nodes on the Ring Bus should not be a consideration when designing your program.

The latency is also affected by the fact that the Ring Bus is expandable. Nodes (with the exception of the two (2) dummy nodes) can expand from one (1) slot to four (4) additional slots to increase Ring Bus node accessibility. The expansion is automatic. When upstream data is injected into a node, the node can expand to guarantee that the receiving node can still output data to the Ring Bus. This expansion enables the node to send four (4) slots of contiguous data.

Taking into account that Ring Bus message transactions are one-way operations and that nodes can expand up to four (4) additional slots, we can see that the worst case round trip latency is:

((19 nodes x 5 slots) + (the two dummy nodes)) = 97 clock cycles (assuming that the target node is not busy when the message arrives).

In rare cases, the latency might increase due to the target node being busy. In this case, the message continues around the bus and until it arrives at the target node again. If the target node is free, the transaction is completed.

A TLU response to the FP does not use the Ring Bus. Rather, these responses are sent via a dedicated path between the TLU and the FP.

Ring Bus Interface Registers

This section describes three (3) type of Ring Bus functions and their registers as shown in Table 96 on page 375.

Table 96 CP Registers by Function

CP Function	Specific Registers
Ring Bus Tx Message Control	TxMsg0_Ctl, TxMsg0_Data_H, TxMsg0_Data_L; TxMsg1_Ctl, TxMsg1_Data_H, TxMsg1_Data_L; TxMsg2_Ctl, TxMsg2_Data_H, TxMsg2_Data_L; TxMsg3_Ctl, TxMsg3_Data_H, TxMsg3_Data_L
Ring Bus Rx Response Control	RxResp0_Ctl, RxResp0_DataH, RxResp0_DataL; RxResp1_Ctl, RxResp1_DataH, RxResp1_DataL; RxResp2_Ctl, RxResp2_DataH, RxResp2_DataL; RxResp3_Ctl, RxResp3_DataH, RxResp3_DataL; RxResp4_Ctl, RxResp4_DataH, RxResp4_DataL; RxResp5_Ctl, RxResp5_DataH, RxResp5_DataL; RxResp6_Ctl, RxResp6_DataH, RxResp6_DataL; RxResp7_Ctl, RxResp7_DataH, RxResp7_DataL
Ring Bus Rx Message Control	RxMsg_Ctl, RxMsg_FIFO

For complete details about specific registers go to their references. Refer to: "TxMsg0_Ctl Register (CP Ring Bus Tx Message Control Function)" on page 401, "TxMsg0_Data_H Register (CP Ring Bus Tx Message Control Function)" on page 403, "TxMsg0_Data_L Register (CP Ring Bus Tx Message Control Function)" on page 403, "RxResp0_Ctl Register (CP Ring Bus Rx Response Control Function)" on page 404, "RxResp0_Data_H Register (CP Ring Bus Rx Response Control Function)" on page 405, "RxResp0_Data_L Register (CP Ring Bus Rx Response Control Function)" on page 405, "RxMsg_Ctl Register (CP Ring Bus Rx Message Control Function)" on page 406, and "RxMsg_FIFO Register (CP Ring Bus Rx Message Control Function)" on page 407.

Ring Bus Transmit (Tx) Message Registers

Configuration Space includes four (4) sets of registers used to transmit messages on the Ring Bus. Refer to "Ring Bus Transmit (Tx) Messages Registers" on page 106.

Ring Bus (Rx) Receive Message Registers

Configuration Space includes a set of registers used to receive unsolicited messages.Refer to "Ring Bus (Rx) Receive Message Registers" on page 107.

Ring Bus Receive (Rx) Response Registers

Messages initiated by the CPRC as a request type expect to receive a subsequent response type message, for example TLU requests. Configuration space includes eight (8) sets of registers used to receive responses. Refer to "Ring Bus Receive (Rx) Response Registers" on page 108.

Global Bus Overview

The Global Bus is a slotted, multichannel, shared, arbitrated bus that supports inter-processor I/O using a single, flat memory model and provides direct access (via load/store operations) to all C-5 NP memory regions except CPRC IMEM, TLU table storage memory, SDP control stores, PCI configuration registers, and some XP configuration registers.

The Global Bus uses a 32bit wide data path with separate control and address path. It can transfer 4Bytes of data on each of two (2) independent data channels. It also has a guaranteed arbitration latency and a bus acknowledgment feature to indicate transaction completion. To achieve high bus utilization, operations are pipelined and reads are split into a read request and a write response. Retry selection is per CP (not per transaction).

Table 97 Global Bus Latency

Worst Case	Average Case	Best
110 cycles	(Total global bandwidth/2) / (Num. of active processors)	10 cycles

Appendix A

C-5 NP Registers

Appendix Overview

This appendix covers the following topics:

Although specific ranges of memory are allocated to specific functions, the entire area may not be used.

Channel Processor (CP) Configuration Registers

Configuration Space in the CPs is an area that contains a number of registers. The CPRC uses these registers to communicate with the SDP and the bus controllers (Payload Bus and Global Bus). The CP's registers can also be accessed by other components of the C-5 NP (XP and other CPs).

CP Registers

The following is a list of each CP register along with its address, function, and reference to its detailed parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions. Refer to Table 98 on page 378.

Table 98 CP Registers

Address	Register Name	Function	Detailed Parameters
0xBC*n*00000	DMEM_Base	CP DMEM	See "Data Memory (DMEM)" on page 81
0xBC*n*04000 to 0xBC*n*0403C	RxSDP0_Ext0 to RxSDP0_Ext15	CP Rx Extract Space0	See page 383
0xBC*n*04080	RxCB0_Sys_Addr	CP Rx Control Block0	See page 384
0xBC*n*04084	RxCB0_Ctl		See page 385
0xBC*n*04088	RxCB0_DMA_Addr		See page 388
0xBC*n*0408C	RxCB0_SDP_Addr		See page 389
0xBC*n*04090	RxCtl0_Status		See page 486
0xBC*n*04100 to 0xBC*n*0413C	TxSDP0_Merge0 to TxSDP0_Merge15	CP Tx Merge Space0	See page 383
0xBC*n*04180	TxCB0_Sys_Addr	CP Tx Control Block0	See page 396
0xBC*n*04184	TxCB0_Ctl		See page 397
0xBC*n*04188	TxCB0_DMA_Addr		See page 398
0xBC*n*0418C	TxCB0_SDP_Addr		See page 399
0xBC*n*04190	TxCtl0_Status		See page 400
0xBC*n*04200 to 0xBC*n*0423C	RxSDP1_Ext0 to RxSDP1_Ext15	CP Rx Extract Space1	See Table 99 on page 383

Table 98 CP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBC*n*04280	RxCB1_Sys_Addr	CP Rx Control Block1	See Table 101 on page 384
0xBC*n*04284	RxCB1_Ctl		See Table 103 on page 387
0xBC*n*04288	RxCB1_DMA_Addr		See Table 104 on page 388
0xBC*n*0428C	RxCB1_SDP_Addr		See Table 105 on page 389
0xBC*n*04290	RxCtl1_Status		See Table 116 on page 400
0xBC*n*04300 to 0xBC*n*0433C	TxSDP1_Merge0 to TxSDP1_Merge15	CP Tx Merge Space1	See Table 100 on page 384
0xBC*n*04380	TxCB1_Sys_Addr	CP Tx Control Block1	See Table 112 on page 396
0xBC*n*04384	TxCB1_Ctl		See Table 113 on page 398
0xBC*n*04388	TxCB1_DMA_Addr		See Table 114 on page 398
0xBC*n*0438C	TxCB1_SDP_Addr		See Table 115 on page 399
0xBC*n*04390	TxCtl1_Status		See Table 116 on page 400
0xBC*n*04400	WrCB0_Sys_Addr	CP Wr Control Block0	See page 390
0xBC*n*04404	WrCB0_Ctl		See page 391
0xBC*n*04408	WrCB0_DMA_Addr		See page 392
0xBC*n*04410	WrCB1_Sys_Addr	CP Wr Control Block1	See Table 106 on page 390
0xBC*n*04414	WrCB1_Ctl		See Table 107 on page 392
0xBC*n*04418	WrCB1_DMA_Addr		See Table 108 on page 392
0xBC*n*04420	RdCB0_Sys_Addr	CP Rd Control Block0	See page 393
0xBC*n*04424	RdCB0_Ctl		See page 394
0xBC*n*04428	RdCB0_DMA_Addr		See page 395
0xBC*n*04430	RdCB1_Sys_Addr	CP Rd Control Block1	See Table 109 on page 393
0xBC*n*04434	RdCB1_Ctl		See Table 110 on page 395
0xBC*n*04438	RdCB1_DMA_Addr		See Table 111 on page 395

Table 98 CP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBC*n*04440	TxMsg0_Ctl	Ring Bus Tx Message Control	See page 401
0xBC*n*04448	TxMsg1_Ctl		See Table 118 on page 402
0xBC*n*04450	TxMsg2_Ctl		
0xBC*n*04458	TxMsg3_Ctl		
0xBC*n*04460	TxMsg0_Data_H		See page 403
0xBC*n*04464	TxMsg0_Data_L		See page 403
0xBC*n*04468	TxMsg1_Data_H		See Table 119 on page 403
0xBC*n*0446C	TxMsg1_Data_L		See Table 120 on page 403
0xBC*n*04470	TxMsg2_Data_H		See Table 119 on page 403
0xBC*n*04474	TxMsg2_Data_L		See Table 120 on page 403
0xBC*n*04478	TxMsg3_Data_H		See Table 119 on page 403
0xBC*n*0447C	TxMsg3_Data_L		See Table 120 on page 403
0xBC*n*04480	RxResp0_Ctl	Ring Bus Rx Response Control	See page 404
0xBC*n*04484	RxResp1_Ctl		See Table 121 on page 404
0xBC*n*04488	RxResp2_Ctl		
0xBC*n*0448C	RxResp3_Ctl		
0xBC*n*04490	RxResp4_Ctl		
0xBC*n*04494	RxResp5_Ctl		
0xBC*n*04498	RxResp6_Ctl		
0xBC*n*0449C	RxResp7_Ctl		
0xBC*n*044A0	RxResp0_Data_H		See page 405
0xBC*n*044A4	RxResp0_Data_L		See page 405
0xBC*n*044A8	RxResp1_Data_H		See Table 122 on page 405
0xBC*n*044AC	RxResp1_Data_L		See Table 123 on page 405

Table 98 CP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBC*n*044B0	RxResp2_Data_H	Ring Bus Rx Response Control (continued)	See Table 122 on page 405
0xBC*n*044B4	RxResp2_Data_L		See Table 123 on page 405
0xBC*n*044B8	RxResp3_Data_H		See Table 122 on page 405
0xBC*n*044BC	RxResp3_Data_L		See Table 123 on page 405
0xBC*n*044C0	RxResp4_Data_H		See Table 122 on page 405
0xBC*n*044C4	RxResp4_Data_L		See Table 123 on page 405
0xBC*n*044C8	RxResp5_Data_H		See Table 122 on page 405
0xBC*n*044CC	RxResp5_Data_L		See Table 123 on page 405
0xBC*n*044D0	RxResp6_Data_H		See Table 122 on page 405
0xBC*n*044D4	RxResp6_Data_L		See Table 123 on page 405
0xBC*n*044D8	RxResp7_Data_H		See Table 122 on page 405
0xBC*n*044DC	RxResp7_Data_L		See Table 123 on page 405
0xBC*n*044E0	RxMsg_Ctl	Ring Bus Rx Message Control	See page 406
0xBC*n*044E4	RxMsg_FIFO		See page 407
0xBC*n*04500 to 0xBC*n*0457C	Rx_SONETOH0 to Rx_SONETOH31	SONET Rx Control	See page 408
0xBC*n*04580 to 0xBC*n*045FC	Tx_SONETOH0 to Tx_SONETOH31	SONET Tx Control	See page 408
0xBC*n*04600	RxCtl_ByteSeq0	SDP Rx Control	See page 408
0xBC*n*04604	RxCtl_ByteSeq1		See Table 124 on page 408
0xBC*n*04608	RxCtl_SyncSeq		See page 409
0xBC*n*0460C	RxCtl_BitSeq0		See page 409
0xBC*n*04610	RxCtl_BitSeq1		See Table 125 on page 409
0xBC*n*04620	TxCtl_ByteSeq0	SDP Tx Control	See page 410
0xBC*n*04624	TxCtl_ByteSeq1		See Table 126 on page 410
0xBC*n*0462C	TxCtl_BitSeq0	SDP Tx Control (continued)	See page 410
0xBC*n*04630	TxCtl_BitSeq1		See Table 127 on page 410

Table 98 CP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBC*n*04640	CP_Mode0	CP Mode Configuration	See page 411
0xBC*n*04644	CP_Mode1		See page 414
0xBC*n*04648	SDP_Mode2		See page 417
0xBC*n*0464C	SDP_Mode3		See page 418
0xBC*n*04650	SDP_Mode4		See page 422
0xBC*n*04654	SDP_Mode5		See page 424
0xBC*n*04658	Debug_Mode		See page 429
0xBC*n*0465C	PIN_Mode		See page 431
0xBC*n*04660	Queue_Status0	CP Queue Status	See page 433
0xBC*n*04664	Queue_Status1		See Table 131 on page 433
0xBC*n*04668	Queue_Status2		
0xBC*n*0466C	Queue_Status3		
0xBC*n*04670	Queue_Update0		See page 433
0xBC*n*04674	Queue_Update1		See Table 132 on page 433
0xBC*n*04678	Queue_Update2		
0xBC*n*0467C	Queue_Update3		
0xBC*n*04684	Event_Timer	CP Miscellaneous Control	See page 434
0xBC*n*04688	Cycle_Count_H		See page 434
0xBC*n*0468C	Cycle_Count_L		See page 434
0xBC*n*046A0	Event0	CP Event and Interrupt Control	See page 435
0xBC*n*046A4	Event1		See page 437
0xBC*n*046A8	Event_Mask0		See page 439
0xBC*n*046AC	Event_Mask1		See Table 133 on page 439
0xBC*n*046B0	Event_Access		See page 439
0xBC*n*046B4	Mask_Access		See page 441
0xBC*n*046B8	Interrupt_Mask0		See page 441
0xBC*n*046BC	Interrupt_Mask1		See Table 134 on page 441
0xBC*n*046C0	SONET_Event		See page 442
0xBC*n*046C4	SONET_Mask		See page 445

CP Detailed Descriptions

The following is a detailed description of each of the CP registers and their individual parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

RxSDP0_Ext0 to RxSDP0_Ext15 Registers (CP Rx Extract Space0 Function)

Purpose — Used to pass fields extracted from the receive data stream by the RxSDP to the CPRC. These registers are used only for receive datascope0. See Table 99 on page 383 for similar registers.

Address — 0xBC*n*04000 to 0xBC*n*0403C

Access — CPRC Read, CPRC Write only writable for test purposes when SDP_Mode3 RxResetx==0, SDP RxByte Processor Write - byte addressable.

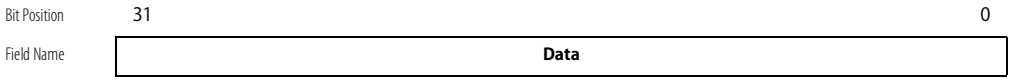


Table 99 RxSDP1_Ext0 to RxSDP1_Ext15 Registers (for Datascope1)

Register Name	Purpose	Address
RxSDP1_Ext0 to RxSDP1_Ext15	Same as registers *RxSDP0_Ext0* to *RxSDP0_Ext15*, but for datascope1.	0xBC*n*04200 to 0xBC*n*0423C

TxSDP0_Merge0 to TxSDP0_Merge15 Registers (CP Tx Merge Space0 Function)

Purpose — Used to pass fields from the CPRC to the TxSDP to merge in with the transmit data stream. These registers are used only for transmit datascope0. See Table 100 on page 384 for similar registers.

Address — 0xBC*n*04100 to 0xBC*n*0413C

Access — CPRC Read, CPRC Write, SDP TxByte Processor Read - byte addressable

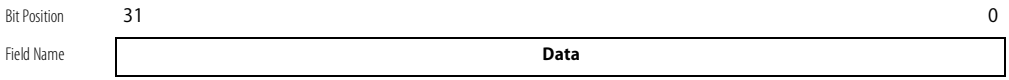


Table 100 TxSDP1_Merge0 to TxSDP1_Merge15 Registers (for Datascope1)

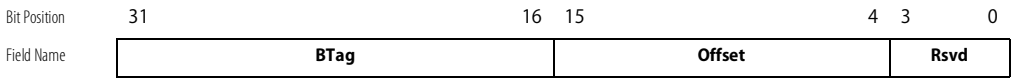
Register Name	Purpose	Address
TxSDP1_Merge0 to TxSDP1_Merge15	Same as registers *TxSDP0_Merge0* to *TxSDP0_Merge15*, but pertains to transmit datascope1.	0xBC*n*04300 to 0xBC*n*0433C

RxCB0_Sys_Addr Register (CP Rx Control Block0 Function)

Purpose Provides an address consisting of a Pool ID, BTag and offset for datascope0. See Table 101 on page 384 for similar register.

Address 0xBC*n*04080

Access CPRC Read/Write



Field Name	Bit Position	Description
BTag	31:16	**Buffer Tag** — Address. Legal range is a physical limit= 0 to 65,532Bytes or 0 to 0xFFFF.
Offset	15:4	**Offset** — Address or Command. Refer to Table 20 on page 118 Legal range= 0 to 65,520Bytes, or 0 to 0xFFF0. Values must be 16Byte aligned.
Reserved	3:0	Read as zero.

Table 101 RxCB1_Sys_Addr Register (for Datascope1)

Register Name	Purpose	Address
RxCB1_Sys_Addr	Same as*RxCB0_Sys_Addr*, except for datascope1.	0xBC*n*04280

RxCB0_Ctl Register (CP Rx Control Block0 Function)

Purpose Controls DMA for payload receive operation for datascope0. See Table 103 on page 387 for similar register.

Address 0xBC*n*04084

Access CPRC Read/Write

Bit Position	31	30	29	28	27 24	23	22	21 20	19	18	17 16	15 0
Field Name	Avail	NoRetry	EOP	Protect EOP	Error	Own 1	Own 0	Ctx	SDP state	Rsvd	DMA state	RxLength
Reset Value	1	x	0	raz	x	x	x	x	0	raz	00	x

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — 1=RxCB is available to the CPRC, 0=Start the DRAM DMA engine.
NoRetry	30	**No Retry** — 1=Do not retry the transaction on bus NACK, 0=Retry, up to 16 (Max.) times before reporting an error.
EOP	29	**End-of-Packet** — Typically this is set by the SDP when scope is switched and cleared by the DMA engine when a transfer completes successfully.
Protect EOP	28	**Protect End-of-Packet** — When set during a RxCB_Ctl write, the EOP bit contained in field [29] is not written. When cleared during a RxCB_Ctl write, the EOP bit is written.
Error	27:24	**Error** — When a DMA operation completes and the *Avail* bit [31] =1, then a non-zero value indicates that the DMA operation completed with an error. Refer to Table 102 on page 386 for error code definitions.
Own1	23	**Block Ownership Bit** — 0=SDP owns, 1= DMA owns.
Own0	22	**Block Ownership Bit** — 0=SDP owns, 1= DMA owns.
Ctx	21:20	**Context** — Two bit field that software can use to provide context or identifying information per requests. This filed has no impact on the operation.
SDP state	19	**SDP State** — Shows the state of the SDP engine: 0=Ready, 1=Wait for line update. During a RxCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set, and not changed if *Avail* bit [31] is clear.
Reserved	18	Read as zero.

Field Name	Bit Position	Description
State	17:16	**DMA State** — Shows the state of the DMA engine: 00 = Idle,10 = Request, 01 = Grant. During a RxCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set, and not changed if *Avail* bit [31] is clear.
RxLength	15:0	**Receive Length** — Count of bytes in the receive payload.

Table 102 Transfer Control Block Error Codes

Error Type	Encoding	Read/ Write	Target	Description
Success	0	Wr, Rx, Rd, Tx	Buffer memory: PID== 0-29 BMU:PID==30 QMU:PID==31	The payload bus transaction completed successfully.
RxSDP Error	8	Rx	Not Applicable	The SDP RxByte sequencer indicated an error by writing to the RxCtl_Status bit [30].
NACK Retry Limit	9	Wr, Rx, Rd, Tx	Buffer memory: PID== 0-29	The BMU was unable to accept a buffer memory transaction. The payload bus transaction was attempted until the NACK retry limit was reached.
NACK Retry Limit	9	Wr	BMU: PID==30	The BMU was unable to accept BTag command, or the multi-use counter table was full on an allocate command. The payload bus transaction was attempted until the NACK retry limit was reached.
NACK Retry Limit	9	Rd	BMU: PID==30	The BMU was unable to accept BTag or multi-use counter command. The payload bus transaction was attempted until the NACK retry limit was reached.
NACK Retry Limit	9	Wr	QMU:PID==31	The QMU was unable to accept a command because the write mailbox was full. The payload bus transaction was attempted until the NACK retry limit was reached.
NACK Retry Limit	9	Rd	QMU:PID==31	The QMU was unable to accept a command because the read mailbox was full. The payload bus transaction was attempted until the NACK retry limit was reached.
Bad Pool	A	Rd	BMU:PID==30	BTag allocate command requested a non-existent pool.

Table 102 Transfer Control Block Error Codes

Error Type	Encoding	Read/ Write	Target	Description
Bad BMU Command	A	Wr	BMU:PID==30	Any of the following conditions occurred: • BMU command requested a non-existent pool, • BTag write found more BTags written than configured, • Multi-use counter allocate found a counter already allocated for this pool/BTag, • Multi-use counter decrement to a non-existent counter.
BTag Unavailable	C	Rd	BMU:PID==30	BTag allocate command found insufficient BTags to complete the allocation.
QMU Read Error	C	Rd	QMU:PID==31	QMU detected a dequeue of an empty queue.
Payload ECC Error	D	Rd	Buffer memory: PID==0-29	Un-correctable ECC error occurred on a buffer memory read.
BTag ECC Error	D	Rd	BMU:PID==30	Uncorrected ECC error occurred then reading memory for a BTag allocate command.
Non-Existent memory	E	Rd	Buffer memory: PID==0-29	A payload read of buffer memory that was out of bounds due to an error in the way software configured the BMU.
Non-Existent memory	E	Rd	BMU:PID==30	A BTag read from memory was out of bounds due to an error in the way software configured the BMU.
No Match on Multi-use Counter Read	F	Rd	BMU:PID==30	A multi-use counter read command of a non-existent counter.

Table 103 RxCB1_Ctl Register (for Datascope1)

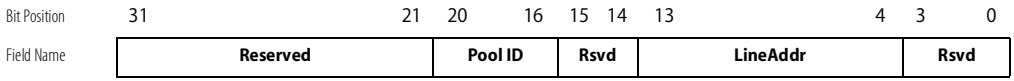
Register Name	Purpose	Address
RxCB1_Ctl	Same as *RxCB0_Ctl*, except for datascope1.	0xBC*n*04284

RxCB0_DMA_Addr Register (CP Rx Control Block0 Function)

Purpose Supplies the address of a 16Byte line in DMEM for DMA and the Pool ID of the buffer to write for datascope0. See Table 104 on page 388 for similar register.

Address 0xBC*n*04088

Access CPRC Read/Write



Field Name	Bit Position	Description
Reserved	31:21	Read as zero.
PoolID	20:16	**Pool ID** — Pool to write too. Legal range= 0 to 31.
Reserved	15:14	Read as zero.
LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA. It is auto-incremented during DMA; bits [6:4] cleared by the DMA engine when a transfer completes successfully.
Reserved	3:0	Read as zero.

Table 104 RxCB1_DMA_Addr Register (for Datascope1)

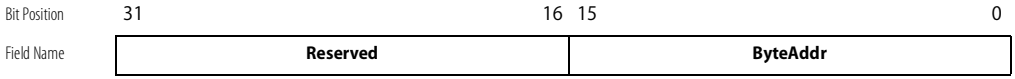
Register Name	Purpose	Address
RxCB1_DMA_Addr	Same as RxCB0_DMA_Addr, except for datascope1.	0xBC*n*04288

RxCB0_SDP_Addr Register (CP Rx Control Block0 Function)

Purpose Supplies the address of a byte in DMEM for DMA. It is auto-incremented during DMA for datascope0. See Table 105 on page 389 for similar register.

Address 0xBC*n*0408C

Access CPRC Read/Write



Field Name	Bit Position	Description
Reserved	31:16	Read as zero.
ByteAddr	15:0	**DMEM Byte Address** — DMEM byte address for DMA. It is auto-increment during DMA; bits [6:0] cleared by the DMA engine when a transfer completes successfully.

Table 105 RxCB1_SDP_Sys_Addr Register (for Datascope1)

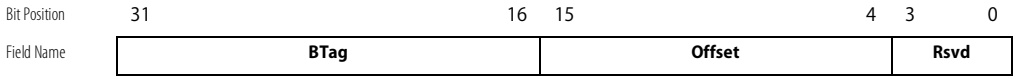
Register Name	Purpose	Address
RxCB1_SDP_Sys_Addr	Same as *RxCB0_SDP_Addr*, except for datascope1.	0xBC*n*0428C

WrCB0_Sys_ Addr Register (CP Wr Control Block0 Function)

Purpose Provides an address consisting of a Pool ID, BTag and offset. See Table 106 on page 390 for similar register.

Address 0xBC*n*04400

Access CPRC Read/Write



Field Name	Bit Position	Description
BTag	31:16	**Buffer Tag** — Legal range is a physical limit= 0 to 65,532Bytes or 0 to 0xFFFF.
Offset	15:4	**Offset** — Address or Command. Refer to "Using Multi-Use Control Blocks to Achieve Different Functions" on page 117. Legal range= 0 to 65,520Bytes, or 0 to 0xFFFF. Values must be 16Bytes aligned.
Reserved	3:0	Read as zero.

Table 106 WrCB1_Sys_Addr Register (for Control Block1)

Register Name	Purpose	Address
WrCB1_Sys_Addr	Same as *WrCB0_Sys_Addr*, except for control block1.	0xBC*n*04410

WrCB0_Ctl Register (CP Wr Control Block0 Function)

Purpose Controls DMA for payload write operation. See Table 107 on page 392 for similar register.

Address 0xBC*n*04404

Access CPRC Read/Write

Bit Position	31	30	29	28	27 24	23 22	21 20	19 18	17 16	15 14	13 0
Field Name	**Avail**	**NoRetry**	**Mod64**	**Rsvd**	**Error**	**Rsvd**	**Ctx**	**Rsvd**	**State**	**Rsvd**	**Length**
Reset Value	**1**	**x**	**x**	**raz**	**x**	**raz**	**x**	**raz**	**0**	**raz**	**x**

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — 1= WrCB is available to the CPRC, 0=Start the DRAM DMA engine.
NoRetry	30	**No Retry** — 1= Do not retry the transaction on bus NACK, 0=Retry 16 (Max.) times before reporting an error.
Mod64	29	**Modulo 64** — 1=Increment *WrCB0_Sys_Addr* bits [15:4] *Offset* field and *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field modulo 64Bytes during DMA to perform a wrap. 0=Increment *WrCB0_Sys_Addr* bits [15:4] *Offset* field and *WrCB0_DMA_Addr* bits [13:4] *LineAddr* field linearly during DMA that steps through memory.
Reserved	28	Read as zero.
Error	27:24	**Error** — When *Avail* bit [31]=1 and a non-zero value is returned after a DMA operation completes, the DMA operation encountered an error. See Table 102 on page 386 for error code definitions.
Reserved	23:22	Read as zero.
Ctx	21:20	**Context** — Two bit field that software can use to provide context or identifying information per requests. This filed has no impact on the operation.
Reserved	19:18	Read as zero.
State	17:16	**DMA State** — Shows the state of the DMA engine: 00 = Idle, 10 = Request, 01 = Grant. During a WrCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set and not changed if *Avail* bit [31] is clear.
Reserved	15:14	Read as zero.
Length	13:0	**Length** — Length of DMA transfer in Bytes. Legal range is a physical limit=12Kbytes.

Table 107 WrCB1_Ctl Register (for Control Block1)

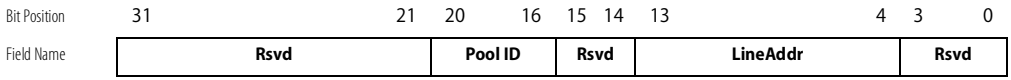
Register Name	Purpose	Address
WrCB1_Ctl	Same as *WrCB0_Ctl*, except for control block1.	0xBC*n*04414

WrCB0_DMA_Addr Register (CP Wr Control Block0 Function)

Purpose Supplies the address of a 16Byte line in DMEM for DMA and the Pool ID of the buffer to write. See Table 108 on page 392 for similar register.

Address 0xBC*n*04408

Access Read/Write



Field Name	Bit Position	Description
Reserved	31:21	Read as zero.
PoolID	20:16	**Pool ID** — Pool to write too. Legal range= 0 to 31.
Reserved	15:14	Read as zero.
LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA. It is auto-increment during DMA.
Reserved	3:0	Read as zero.

Table 108 WrCB1_DMA_Addr Register (for Control Block1)

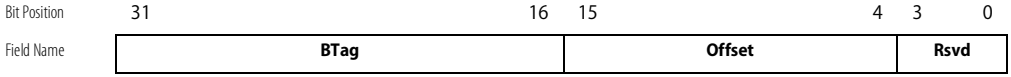
Register Name	Purpose	Address
WrCB1_DMA_Addr	Same as *WrCB0_DMA_Addr*, except for control block1.	0xBC*n*04418

RdCB0_Sys_Addr Register (CP Rd Control Block0 Function)

Purpose Provides an address consisting of a pool ID, BTag, and offset. See Table 109 on page 393 for similar register.

Address 0xBC*n*04420

Access CPRC Read/Write



Field Name	Bit Position	Description
BTag	31:16	**Buffer Tag** — Address Legal range is a physical limit= 0 to 65,532Bytes or 0 to 0xFFFF.
Offset	15:4	**Offset** — Address or Command. Refer to "Using Multi-Use Control Blocks to Achieve Different Functions" on page 117. Legal range= 0 to 65,520Bytes, or 0 to 0xFFFF. Values must be 16Bytes aligned.
Reserved	3:0	Read as zero.

Table 109 RdCB1_Sys_Addr register (for Control Block1)

Register Name	Purpose	Address
RdCB1_Sys_Addr	Same as RdCB0_Sys_Addr, except for control block1.	0xBC*n*04430

RdCB0_Ctl Register (CP Rd Control Block0 Function)

Purpose Controls DMA for payload read operation. See Table 110 on page 395 for similar register.

Address 0xBC*n*04424

Access CPRC Read/Write

Bit Position	31	30	29	28	27 24	23 22	21 20	19 18	17 16	15 14	13 4	3 0
Field Name	Avail	NoRetry	Mod64	Rsvd	Error	Rsvd	Ctx	Rsvd	State	Rsvd	Length	Rsvd
Reset Value	1	x	x	raz	x	raz	x	raz	0	raz	x	

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — 1=RdCB is available to the CPRC, 0=Start the DRAM DMA engine.
NoRetry	30	**No Retry** — 1=Do not retry the transaction on bus NACK, 0=retry, up to 16 (Max.) times.
Mod64	29	**Modulo 64** — 1=Increment *RdCB0_Sys_Addr* bits [15:4] *Offset* field and *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field modulo 64Bytes during DMA to perform a wrap. 0=Increment *RdCB0_Sys_Addr* bits [15:4] *Offset* field and *RdCB0_DMA_Addr* bits [13:4] *LineAddr* field linearly during DMA that steps through memory.
Reserved	28	Read as zero.
Error	27:24	**Error** — When *Avail* bit [31]=1 and a non-zero value is returned after a DMA operation completes, the DMA operation encountered an error. Refer to Table 102 on page 386 for error code definitions.
Reserved	23:22	Read as zero.
Ctx	21:20	**Context** — Two bit field that software can use to provide context or identifying information per requests. This filed has no impact on the operation.
Reserved	19:18	Read as zero.
State	17:16	**DMA State** — Shows the state of the DMA engine (read only): 00 =Idle, 10 = Request, 01 = Grant. During a RdCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set and not changed if *Avail* bit [31] is clear.
Reserved	15:14	Read as zero.
Length	13:0	**Length** — Length of DMA transfer in bytes. Legal range is a physical limit=12KBytes.

Table 110 RdCB1_Ctl Register (for Control Block1)

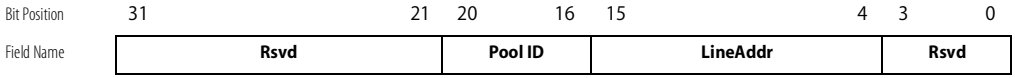
Register Name	Purpose	Address
RdCB1_Ctl	Same as *RdCB0_Ctl*, except for control block1.	0xBC*n*04434

RdCB0_DMA_Addr Register (CP Rd Control Block0 Function)

Purpose Supplies the address of a 16Byte line in DMEM for DMA and the Pool ID for buffer to read. See Table 111 on page 395 for similar register.

Address 0xBC*n*04428

Access CPRC Read/Write



Field Name	Bit Position	Description
Reserved	31:21	Read as zero.
PoolID	20:16	**Pool ID** — Pool to read too. Legal range= 0 to 31.
LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA operation. It is auto-incremented during DMA.
Reserved	3:0	Read as zero.

Table 111 RdCB1_DMA_Addr Register (for Control Block1)

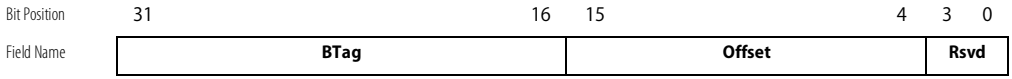
Register Name	Purpose	Address
RdCB1_DMA_Addr	Same as RdCB0_DMA_Addr, except for control block1.	0xBC*n*04438

TxCB0_Sys _Addr Register (CP Tx Control Block0 Function)

Purpose Provides an address consisting of a Pool ID, BTag and offset for datascope0. See Table 112 on page 396 for similar register.

Address 0xBC*n*04180

Access CPRC Read/Write



Field Name	Bit Position	Description
BTag	31:16	**Buffer Tag** — Address Legal range is a physical limit= 0 to 65,532Bytes or 0 to 0xFFFF.
Offset	15:4	**Offset** — Address or Command. Refer to "Using Multi-Use Control Blocks to Achieve Different Functions" on page 117. Legal range= 0 to 65,520Bytes, or 0 to 0xFFF0. Values must be 16Byte aligned.
Reserved	3:0	Read as zero.

Table 112 TxCB1_Sys_Addr Register (for Datascope1)

Register Name	Purpose	Address
TxCB1_Sys_Addr	Same as *TxCB0_Sys_Addr*, except for datascope1.	0xBC*n*04380

TxCB0_Ctl Register (CP Tx Control Block0 Function)

Purpose Controls DMA for payload transmit operation for datascope0. See Table 113 on page 398 for similar register.

Address 0xBC*n*04184

Access CPRC Read/Write

Bit Position	31	30	29	28	27 24	23	22	21 20	19 18	17 16	15 0
Field Name	Avail	NoRetry	EOP	OOB	Error	Own1	Own0	Ctx	SDP state	DMA state	TxLength
Reset Value	1	x	0	x	x	0	0	x	0	00	x

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — 1=TxCB is available to the CPRC, 0=Start the DRAM DMA engine.
NoRetry	30	**No Retry** — 1=Do not retry the transaction on bus NACK, 0=Retry, up to 16 (Max.) times before reporting an error.
EOP	29	**End of Packet** — Typically this is set by the SDP when scope is switched and cleared by the DMA engine when a transfer completes successfully. During a TxCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set, and not changed if *Avail* bit [31] is clear.
OOB	28	**Out of Band** — 1=Use out-of-band (OOB) bits, 0=Use the length field to determine end-of-frame.
Error	27:24	**Error** —When a DMA operation completes and the *Avail* bit [31] =1, then a non-zero value indicates that the DMA operation completed with an error. Refer to Table 102 on page 386 for error code definitions.
Own1	23	**Block Ownership Bit** — 1=SDP owns, 0=DMA owns.
Own0	22	**Block Ownership Bit** — 1=SDP owns, 0=DMA owns.
Ctx	21:20	**Context** — Two bit field that software can use to provide context or identifying information per requests. This filed has no impact on the operation.
SDP state	19:18	**SDP State** — Shows the state of the SDP engine. 0=Waiting for data, Non-zero value= Ready for transmit. During a TxCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set, and not changed if *Avail* bit [31] is clear.
State	17:16	**DMA State** — Shows the state of the DMA engine: 00 = Idle,10 = Request, 01 = Grant. During a TxCB*n*_Ctl write, this field is updated if *Avail* bit [31] is set, and not changed if *Avail* bit [31] is clear.

Field Name	Bit Position	Description
TxLength	15:0	**Transmit Length** — Counts down the bytes of transmit payload.

Table 113 TxCB1_Ctl Register (for Datascope1)

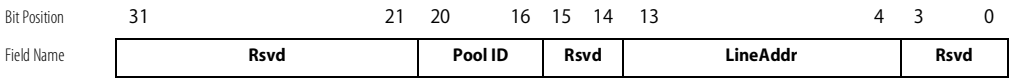
Register Name	Purpose	Address
TxCB1_Ctl	Same as *TxCB0_Ctl*, except for datascope1.	0xBC*n*04384

TxCB0_DMA_Addr Register (CP Tx Control Block0 Function)

Purpose Supplies the address of a 16Byte line in DMEM for DMA and the Pool ID of buffer to read for datascope0. See Table 114 on page 398 for similar register.

Address 0xBC*n*04188

Access CPRC Read/Write



Field Name	Bit Position	Description
Reserved	31:21	Read as zero.
Pool ID	20:16	**Pool ID** — Pool to read too. Legal range= 0 to 31.
Reserved	15:14	Read as zero.
LineAddr	13:4	**DMEM Line Address** — DMEM 16Byte line address for DMA. It is auto-incremented during DMA; bits [6:4] cleared by DMA engine when a transfer completes successfully.
Reserved	3:0	Read as zero.

Table 114 TxCB1_DMA_Addr Register (for Datascope1)

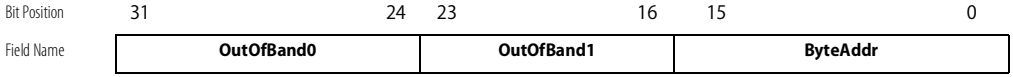
Register Name	Purpose	Address
TxCB1_DMA_Addr	Same as *TxCB0_DMA_Addr*, except for datascope1.	0xBC*n*04388

TxCB0_SDP_Addr Register (CP Tx Control Block0 Function)

Purpose Supplies the address of a byte in DMEM for DMA for transmit datascope0. See Table 115 on page 399 for similar register.

Address 0xBC*n*0418C

Access CPRC Read/Write



Field Name	Bit Position	Description
OutOfBand0	31:24	**Out of Band0** — The eight (OOB) bits accompanying DMEM buffer 0.
OutOfBand1	23:16	**Out of Band1** — The eight (OOB) bits accompanying DMEM buffer 1.
ByteAddr	15:0	**DMEM Byte Address** — DMEM byte address for DMA. It is auto-incremented during DMA; bits [6:0] cleared by the DMA engine when a transfer completes successfully.

Table 115 TxCB1_SDP_Addr Register (for Datascope1)

Register Name	Purpose	Address
TxCB1_SDP_Addr	Same as *TxCB0_SDP_Addr*, except for datascope1.	0xBC*n*0438C

TxCtl0_Status Register (CP Tx Control Block0 Function)

Purpose Semaphores governing SDP transmit operation for data cope0. See Table 116 on page 400 for similar register.

Address 0xBC*n*04190

Access Read/Write

Bit Position	31	30	29	28	27	26	25	24	23 ... 0
Field Name	**Avail**	**Error**	**L5**	**L4**	**L3**	**L2**	**L1**	**L0**	**Rsvd**
Reset Value	**1**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**raz**

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, CPRC owns. When the bit is 0, SDP owns the scope.
Error	30	**TxSDP Error** — SDP sets this bit to indicate and error during transmit processing.
L5 - L0	29:24	**SDP Level Bits** — SDP sets the corresponding bit to indicate level of processing, software defined.
Reserved	23:0	Read as zero.

Table 116 TxCtl1_Status Register (for Datascope1)

Register Name	Purpose	Address
TxCtl1_Status	Same as *TxCtl0_Status*, except for datascope1.	0xBC*n*04390

TxMsg0_Ctl Register (CP Ring Bus Tx Message Control Function)

Purpose — Provides the control portion of an outgoing Ring Bus message. See Table 118 on page 402 for similar registers.

Address — 0xBC*n*04440

Access — CPRC Read/Write = byte addressable, SDP Receive Byte Sequencer Write = field addressable

Bit Position	31	30 24	23	22 20	19 18	17 15	14 10	9 5	4 0
Field Name	Avail	Rsvd	Error	Rsvd	Type	Len	Seq	Dst	Src
Reset Value	1	raz	x	raz	x	x	x	x	x

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, slot is available to the CPRC. When the bit is 0, start the Ring Bus transmit engine.
Reserved	30:24	Read as zero.
ErrorFlag	23	**Error Flag** — Error bit of transmit message definable by the transmitter.
Reserved	22:20	Read as zero.
Type	19:18	**Transmit Message Type**: **Encoded Value** / **Type** 0 — Indication 1 — Confirmation 2 — Request 3 — Response
Len	17:15	**Transmit Message Length** — Length field of transmit message in 8-byte slots. Valid values are 1, 2, 4.
Seq	14:10	**Transaction Sequence Number** — Transaction sequence number of transmit message. Note: That the low-order three bits of this field specify the Receive Response Slot on which the message to be transmitted will be received.
Dst	9:5	**Transaction Message Destination** — Message destination, (Processor ID) typically the TLU.

Field Name	Bit Position	Description
Src	4:0	**Transmit Message Source** — Source of transmit message, typically the processor's Ring Bus node ID. See Table 117 on page 402.

Table 117 Ring Bus Processor IDs

Processor	Ring Bus Node ID	Processor	Ring Bus Node ID
CP0	0	CP10	10
CP1	1	CP11	11
CP2	2	CP12	12
CP3	3	CP13	13
CP4	4	CP14	14
CP5	5	CP15	15
CP6	6	XP	24
CP7	7	FP*	30
CP8	8	TLU	31
CP9	9		

* can only send messages on the Ring Bus. There is a direct connection from the TLU to the FP for responses. If some other node tries to send to the FP, the messages will circulate forever on the Ring Bus.

Table 118 TxMsg*n*_Ctl Registers (for Messages 1, 2 and 3)

Register Name	Purpose	Address
TxMsg1_Ctl	Same as *TxMsg0_Ctl*.	0xBC*n*04448
TxMsg2_Ctl	Same as *TxMsg0_Ctl*, but not writable from RxByte Processor.	0xBC*n*04450
TxMsg3_Ctl	Same as *TxMsg0_Ctl*, but not writable from RxByte Processor.	0xBC*n*04458

TxMsg0_Data_H Register (CP Ring Bus Tx Message Control Function)

Purpose Bits [63:32] of the transmit message data slot (big endian bytes 0-3). See Table 119 on page 403 for similar registers.

Address 0xBC*n*04460

Access CPRC Read/Write, RxByte Processor Read/Write, and is byte addressable

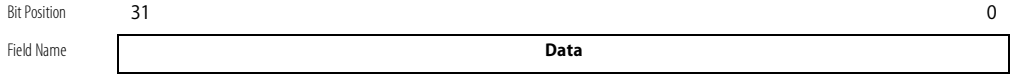


Table 119 TxMsg*n*_Data_H Registers (for Messages 1, 2 and 3)

Register Name	Purpose	Address
TxMsg1_Data_H	Same as *TxMsg0_Data_H*.	0xBC*n*04468
TxMsg2_Data_H	Same as *TxMsg0_Data_H*, but not writable form RxByte Processor.	0xBC*n*04470
TxMsg3_Data_H	Same as *TxMsg0_Data_H*, but not writable form RxByte Processor.	0xBC*n*04478

TxMsg0_Data_L Register (CP Ring Bus Tx Message Control Function)

Purpose Bits [31:0] of the transmit message data slot (big endian bytes 4-7). See Table 120 on page 403 for similar registers.

Address 0xBC*n*04464

Access CPRC Read/Write, RxByte Processor Read/Write, and is byte addressable

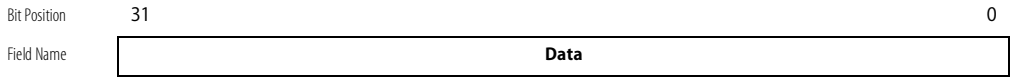


Table 120 TxMsg*n*_Data_L Registers (for Messages 1, 2 and 3)

Register Name	Purpose	Address
TxMsg1_Data_L	Same as TxMsg0_Data_l.	0xBCn0446C
TxMsg2_Data_L	Same as TxMsg0_Data_l, but not writable from RxByte Processor.	0xBCn04474
TxMsg3_Data_L	Same as TxMsg0_Data_l, but not writable from RxByte Processor.	0xBCn0447C

RxResp0_Ctl Register (CP Ring Bus Rx Response Control Function)

Purpose The control portion of an incoming Ring Bus response. See Table 121 on page 404 for similar registers.

Address 0xBC*n*04480

Access CPRC Read/Write

Bit Position	31	30 24	23	22 13	12 8	7 5	4 0
Field Name	Avail	Rsvd	Error	Rsvd	Seq	Rsvd	Src
Reset Value	0	raz	x	raz	x	raz	x

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, slot is valid for the CPRC. When the bit is 0, allow Ring Bus to fill.
Reserved	30:24	Read as zero.
ErrorFlag	23	**Error Flag** — Error bit of response definable by user programming.
Reserved	22:13	Read as zero.
Seq	12:8	**Response Sequence Number** — Sequence number of the response, bits [10:8] match *RxRespCtl* register number.
Reserved	7:5	Read as zero.
Src	4:0	**Response Source** — Source field of the response, typically the processor's Ring Bus node ID. See Table 117 on page 402.

Table 121 RxResp*n*_Ctl Registers (for Ring Bus Responses 1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
RxResp1_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*04484
RxResp2_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*04488
RxResp3_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*0448C
RxResp4_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*04490
RxResp5_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*04494
RxResp6_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*04498
RxResp7_Ctl	Same as *RxResp0_Ctl*.	0xBC*n*0449C

RxResp0_Data_H Register (CP Ring Bus Rx Response Control Function)

Purpose Bits [63:32] of the data portion of an incoming Ring Bus response (big endian bytes 0-3). See Table 122 on page 405 for similar registers.

Address 0xBC*n*044A0

Access CPRC Read/Write

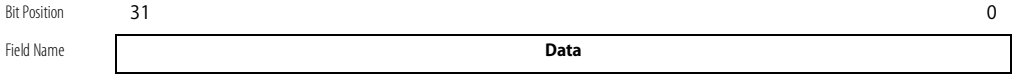


Table 122 RxResp*n*_Data_H Registers (for Ring Bus Responses 1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
RxResp1_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044A8
RxResp2_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044B0
RxResp3_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044B8
RxResp4_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044C0
RxResp5_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044C8
RxResp6_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044D0
RxResp7_Data_H	Same as *RxResp0_Data_H*.	0xBC*n*044D8

RxResp0_Data_L Register (CP Ring Bus Rx Response Control Function)

Purpose Bits [31:0] of the data portion of an incoming Ring Bus response (big endian bytes 4-7). See Table 123 on page 405 for similar registers.

Address 0xBC*n*044A4

Access CPRC Read/Write

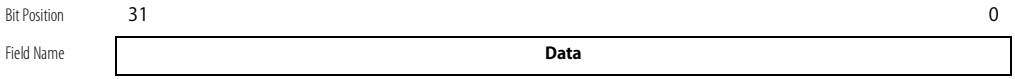


Table 123 RxResp*n*_Data_L Registers (for Ring Bus Responses 1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
RxResp1_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044AC
RxResp2_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044B4

Table 123 RxResp*n*_Data_L Registers (for Ring Bus Responses 1, 2, 3, 4, 5, 6 and 7) (continued)

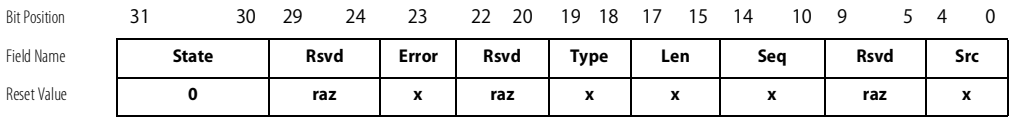
Register Name	Purpose	Address
RxResp3_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044BC
RxResp4_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044C4
RxResp5_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044CC
RxResp6_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044D4
RxResp7_Data_L	Same as *RxResp0_Data_L*.	0xBC*n*044DC

RxMsg_Ctl Register (CP Ring Bus Rx Message Control Function)

Purpose The top of the control portion of the Ring Bus receive message FIFO.

Address 0xBC*n*044E0

Access CPRC Read/Write



Bit Position	31 30	29 24	23	22 20	19 18	17 15	14 10	9 5	4 0
Field Name	State	Rsvd	Error	Rsvd	Type	Len	Seq	Rsvd	Src
Reset Value	0	raz	x	raz	x	x	x	raz	x

Field Name	Bit Position	Description
State	31:30	**Receive Message State**: **Encoded Value** / **State** 00 – Empty 10 – High word 11 – Low word If *State* equals 1, there is at least one valid message is in the receive FIFO available to the CPRC process. *State* equals 1 for as long as any portion of a valid message remains in the FIFO.
Reserved	29:24	Read as zero.
ErrorFlag	23	**Error Flag** — Error bit of receive message definable by user programming.
Reserved	22:20	Read as zero.

Field Name	Bit Position	Description
Type	19:18	**Receive Message Type**: **Encoded Value** / **Type** 0 – Indication 1 – Confirmation 2 – Request
Len	17:15	**Length** — Length of receive message in 8Byte slots. Valid values are 1, 2, or 4.
Seq	14:10	**Transaction Sequence Number** — Transaction sequence number of receive message.
Reserved	9:5	Read as zero.
Src	4:0	**Source** — Source field of receive message, typically the processor's Ring Bus node ID. See Table 117 on page 402.

RxMsg_FIFO Register (CP Ring Bus Rx Message Control Function)

Purpose The next four bytes of data from the Ring Bus receive message FIFO.

Address 0xBC*n*044E4

Access CPRC Read – Reading any portion of this register advances the receive FIFO. CPRC Write for test purposes only, writes the register but does not effect the receive FIFO.

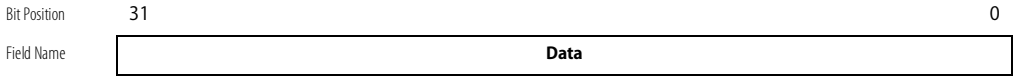
Bit Position	31	0
Field Name	**Data**	

Rx_SONETOH0 to Rx_SONETOH31 Registers (CP SONET Rx Control Function)

Purpose — Used for passing SONET Overhead fields extracted from the receive data stream by the framer to the CPRC.

Address — 0xBC*n*04500 to 0xBC*n*0457C

Access — CPRC Read, CPRC Write (during test), SDP Receive SONET Framer Write, and is byte addressable

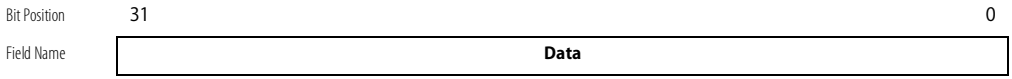


Tx_SONETOH0 to Tx_SONETOH31 Registers (CP SONET Tx Control Function)

Purpose — Used for merging SONET Overhead fields from the CPRC into the transmit data stream.

Address — 0xBC*n*04580 to 0xBC*n*045FC

Access — CPRC Read, CPRC Write, SDP Transmit SONET Framer Read, and is byte addressable



RxCtl_ByteSeq0 Register (CP SDP Rx Control Function)

Purpose — Provides an area for passing information between the CPRC and RxByte Processor. See Table 124 on page 408 for similar register.

Address — 0xBC*n*04600

Access — CPRC Read/Write, SDP Receive Byte Sequencer Read/Write, and is byte addressable

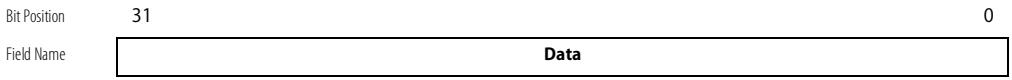


Table 124 RxCtl_ByteSeq1 Register (for Byte Sequence1)

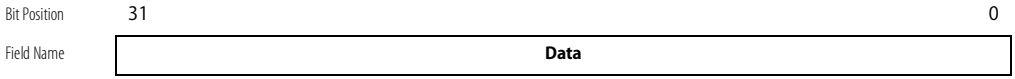
Register Name	Purpose	Address
RxCtl_ByteSeq1	Same as RxCtl_ByteSeq0.	0xBCn04604

RxCtl_SyncSeq Register (CP SDP Rx Control Function)

Purpose — Provides an area for passing information between the CPRC and RxSync Processor.

Address — 0xBC*n*04608

Access — CPRC Read/Write, RxSync Sequencer Read/Write, and is byte addressable



RxCtl_BitSeq0 Register (CP SDP Rx Control Function)

Purpose — Provides an area for passing information between the CPRC and RxBit Processor. See Table 125 on page 409 for similar register.

Address — 0xBC*n*0460C

Access — CPRC Read/Write, RxBit Sequencer Read/Write, and is byte addressable

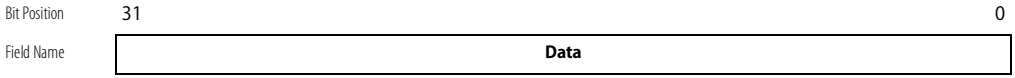


Table 125 RxCtl_BitSeq1 Register (for Bit Sequence1)

Register Name	Purpose	Address
RxCtl_BitSeq1	Same as *RxCtl_BitSeq0*.	0xBC*n*04610

TxCtl_ByteSeq0 Register (CP SDP Tx Control Function)

Purpose Provides an area for passing information between the CPRC and TxByte Processor. See Table 126 on page 410 for similar register.

Address 0xBC*n*04620

Access CPRC Read/Write, TxByte Processor Read/Write, and is byte addressable

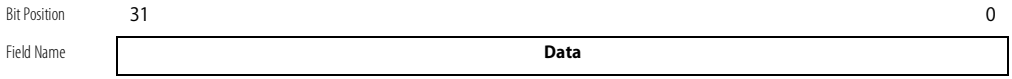


Table 126 TxCtl_ByteSeq1 Register (for Byte Sequence1)

Register Name	Purpose	Address
TxCtl_ByteSeq1	Same as *TxCtl_ByteSeq0*.	0xBC*n*04624

TxCtl_BitSeq0 Register (CP SDP Tx Control Function)

Purpose Provides an area for passing information between the CPRC and TxBit Processor. See Table 127 on page 410 for similar register.

Address 0xBC*n*0462C

Access CPRC Read/Write, TxBit Processor Read/Write, and is byte addressable

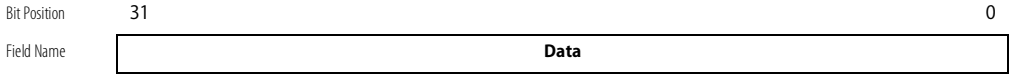


Table 127 TxCtl_BitSeq1 Register (for Bit Sequence1)

Register Name	Purpose	Address
TxCtl_BitSeq1	Same as *TxCtl_BitSeq0*.	0xBC*n*04630

CP_Mode0 Register (CP Mode Configuration Function)

Purpose Collects mode and control bits relevant to general CPRC and CP configuration.

Address 0xBC*n*04640

Bit Position	31	30	29	28	27 24	23 22	21 20	19	18 17	16	15 8	7	6	5	4	3	2	1	0
Field Name	RC Resetx	Wind Down	CP to XP IRQ	CP to all IRQ	Rsvd	QMU rdmbx	QMU wrmbx	Rsvd	Imode	Retry Global	WCS Write Byte	Scan Data Out	RxWCS Write	TxWCS Write	Rsvd	Scan Capture	Scan Update	Scan DataIn1	Scan DataIn0
Reset Value	0	raz	raz	raz	raz	0	0	raz	0	0	raz	x	raz	raz	raz	raz	raz	raz	raz

Field Name	Bit Position	Description
RC_Resetx	31	**CPRC Resetx** — The active low *CPRC Resetx* bit powers up asserted, that is, = 0, so that the CPRC is in the reset state. It must be set by the XP or another CPRC to release the reset and enable the CPRC to begin the boot sequence. A CP can not enable itself.
WindDown	30	**Wind Down** — When the bit is written to 1, it asserts a global signal informing all chip functions to wind down as soon as possible, and to leave as much predictable error recovery state around as possible.
CPtoXPIRQ	29	**CP to XP IRQ** — When the bit is written to 1, it asserts a global signal causing an XP interrupt from this CP.
CPtoAllIRQ	28	**CP to All IRQ** - When the bit is written to 1, it asserts a global signal that causes an interrupt to every CP.
Reserved	27:24	Read as zero.
QMU rdmbx	23:22	**QMU rdmbx Status**: **Encoded Value** / **Status** 00 — QMU idle or operation finished successfully 01 — operation finished with error (probably resource error, see above) 10 — busy, waiting to begin execution 11 — busy, executing in QMU engine

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>QMU wrmbx</td><td>21:20</td><td>QMU wrmbx Status:

<table><tr><th>Encoded Value</th><th>Status</th></tr><tr><td>00</td><td>QMU idle or operation finished successfully</td></tr><tr><td>01</td><td>operation finished with error (probably resource error, see above)</td></tr><tr><td>10</td><td>busy, waiting to begin execution</td></tr><tr><td>11</td><td>busy, executing in QMU engine</td></tr></table></td></tr>
<tr><td>Reserved</td><td>19</td><td>Read as zero.</td></tr>
<tr><td>Imode</td><td>18:17</td><td>IMEM Configuration Mode:

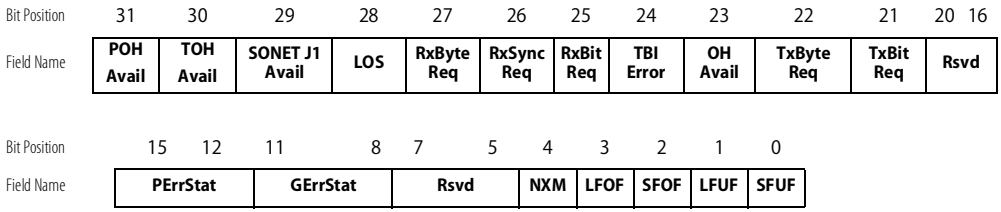
<table><tr><th>Encoded Value</th><th>Mode</th></tr><tr><td>00</td><td>Unshared memory</td></tr><tr><td>11</td><td>4-way shared memory</td></tr><tr><td>01</td><td>Unsupported</td></tr><tr><td>10</td><td>Unsupported</td></tr></table>
When using cluster memory, only CP0,4,8 & 12 use these 2 bits, therefore, CP1-3, 5-7, 9-11 & 13-15 do not use these 2 bits.</td></tr>
<tr><td>RetryGlobal</td><td>16</td><td>Global Bus Transaction Retry — This bit causes global load and store operations through the Global bus controller to be retried up to 256 times when NACK'd. When 256 tries have been NACK'd, the bus controller terminates the operation and asserts a bus error.</td></tr>
<tr><td>WCS Write Byte</td><td>15:8</td><td>Writable Control Store Write Byte — Writing this byte coincident with writing bit 6 and/or bit 5 causes the byte data to be written to SDP control store.</td></tr>
<tr><td>ScanDataOut</td><td>7</td><td>Scan Chain Data Out — the value of this last bit in the SDP scan chain.</td></tr>
</table>

Field Name	Bit Position	Description
RxWCSWrite	6	**RxWritable Control Store Write** — When the bit is 1, it causes the data in the WCS Write Byte field to be loaded into RxSDP control store at a value pointed to by the internal SDP WCS load address register, and the address to increment.
TxWCSWrite	5	**TxWritable Control Store Write** — When the bit is 1, it causes the data in the WCS Write Byte field to be loaded into Transmit SDP control store at a value pointed to by the internal SDP WCS load address register, and the address to increment.
Reserved	4	Read as zero.
ScanCapture	3	**Scan Capture** — Parallel load of the SDP scan chain with chip state.
ScanUpdate	2	**Scan Update** — Parallel drive SDP scan chain data into chip state.
ScanDataIn1	1	**Scan Data In 1** — Shift a 1 serially into the SDP scan chain.
ScanDataIn0	0	**Scan Data In 0** — Shift a 0 serially into the SDP scan chain.

CP_Mode1 Register (CP Mode Configuration Function)

Purpose Collects mode and status bits relevant to general CP configuration.

Address 0xBC*n*04644



Field Name	Bit Position	Description
POH Avail	31	**SONET Payload Overhead Address Avail** — When set, indicates that the final byte of path overhead (Z5) has just been written to the receive SONET overhead register area. All desired path overhead must be read out before the next time this signal is asserted or it will be overwritten
TOH Avail	30	**SONET Transport Overhead Address Avail** — When set, indicates that the final byte of transport overhead (E2) has just been written to the receive SONET overhead register area. All desired transport overhead must be read out before the next time this signal is set to a one or it will be overwritten. This bit is also available in SONET event register.
SONET J1 Avail	29	**SONET J1 Avail** — Indicates that the new j1 index written by the CPRC now has the corresponding J1 in the J1 overhead register area. This is cleared by writing a new j1 index. This is set by writing a J1 byte to the overhead register. This bit is also available in SONET event register.
LOS	28	**Loss of Signal** — This signal is either loss of synchronization or loss of frame as a function of other mode bits.

RxSONET Enable	LOS Mode	Function
0	0	Gigabit ethernet loss of sync
0	1	Fibre channel loss of sync
1	x	All zeros received for > 2.3 µseconds

Field Name	Bit Position	Description
RxByteReq	27	**SDP RxByte Service Request** — When set, RxByte processor microcode needs attention of the CPRC. This is also connected to the CP event register. This bit is written by the RxByte processor.
RxSyncReq	26	**SDP RxSync Service Request** — When set, RxSync processor microcode needs attention of the CPRC. This is also connected to the CP event register. This bit is written by the RxSync processor.
RxBitReq	25	**SDP RxBit Service Request** — When set, RxBit processor microcode needs the attention of the CPRC. This is also connected to the CP event register. This bit is written by the RxBit processor.
TBI Error	24	**Ten Bit Interface Symbol Error** — When set to a one, indicates that there has been a ten bit symbol error since this bit was last cleared. This bit is set by the ten bit decoder and cleared by the CPRC by writing a one to this bit. It is readable by the CPRC. This cannot be used for error counting, but it does give some indication as to the health of the link.
OH Avail	23	**SONET Overhead Avail** — show the current value of the SDP Transmit SONET Framer's address bit 6 into the SONET overhead registers.
TxByteReq	22	**SDP TxByte Service Request** — When set, TxByte processor microcode needs attention of the CPRC. This is also connected to the CP event register. This bit is written by the TxByte processor.
TxBitReq	21	**SDP TxBit Service Request** — When set, TxBit processor microcode needs attention of the CPRC. This is also connected to the CP event register. This bit is written by the TxBit processor.
Reserved	20:16	Read as zero.
PErrStat	15:12	**Payload Error Status** — loaded when a Payload Error occurs and is locked until the CPRC Process clears the PErr bit. The individual control blocks can be interrogated to determine the specific offender. Write 1 to clear. Refer to Table 102 on page 386 for error code definitions.
GErrStat	11:8	**Global Error Status** — loaded when a Global Error occurs and locked until the RC process clears the GErr bit. Codes are shown in Table 128. Write 1 to clear.
Reserved	7:5	Read as zero.

Field Name	Bit Position	Description
NXM	4	**Cluster Non-Existent Memory** — CPRC reference to cluster space addressed non-existent Data Memory (DMEM) or non-local configuration register space. Write 1 to clear.
LFOF	3	**Receive Large FIFO Overflow** — Receive Large FIFO overflow condition. Write 1 to clear.
SFOF	2	**Receive Small FIFO Overflow** —Receive Small FIFO overflow condition. Write 1 to clear.
LFUF	1	**Transmit Large FIFO Underflow** — Transmit Large FIFO underflow condition. Write 1 to clear.
SFUF	0	**Transmit Small FIFO Underflow** — Transmit Small FIFO underflow condition. Write 1 to clear.

Table 128 Global Bus Error Status Encoding

Error Type	Encoding	Read/Write	Description
Success	0	R, W	The transaction completed successfully. If an error is reported, it is due to a non-cluster local DMEM reference.
NACK Retry Limit	9	R, W	The target was unable to accept a global memory request. The global bus transaction was attempted until the NACK retry limit was reached.
Non-Existent Memory	A	R, W	Reference made to an un-subscribed 1MByte CP block.

SDP_Mode2 Register (CP Mode Configuration Function)

Purpose Collects SONET alarm and status information.

Address 0xBC*n*04648

Access CPRC Read/Write

Bit Position	31	30	29	28	27	26	25	24	23	22	21 16	15 8	7 6	5 0
Field Name	LOS	LOF	ASI-L	REI-L	RDI-L	LOP-P	ASI-P	REI-P	RDI-P	LCD-P	Rsvd	SONET_C2_Exp	Rsvd	SONET_Rx J1_Idx

Field Name	Bit Position	Description
LOS	31	**Loss Of Signal** — Bit is read-only and level-sensitive. This bit is meaningful for SONET, Gigabit Ethernet, and FibreChannel applications. When set to 1, LOS is on.
LOF	30	**Loss Of Framing** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, LOF is on.
AIS-L	29	**Alarm Indication Signal - Line** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, AIS-L is on.
REI-L	28	**Remote Error Indicator - Line** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, REI-L is on.
RDI-L	27	**Remote Defect Indicator - Line** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, RDI-L is on.
LOP-P	26	**Loss Of Pointer - Path** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, LOP-P is on.
AIS-P	25	**Alarm Indication Signal - Path** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, AIS-P is on.
REI-P	24	**Remote Error Indicator - Path** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, REI-P is on.
RDI-P	23	**Remote Defect Indicator - Path** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, RDI-P is on.
LCD-P	22	**Loss of Cell/Packet Delineation - Path** — Bit is read-only and level-sensitive. SONET only bit. When set to 1, LCD is on.
Reserved	21:16	Read as zero.
Sonet_C2_Exp	15:8	**SONET C2 Expected** — This is the value for C2 signal label that is expected to be received from the link partner. Used to generate ERDI-P (extended remote defect indication - path) on transmit. Also, causes the setting of *C2_ERROR* bit in the *SONET_EVENT* register when received signal label doesn't match this value.
Reserved	7:6	Read as zero.

Field Name	Bit Position	Description
Sonet_J1_Idx	5:0	**RxSONET J1 Index** — Indicate which of the 64Byte path trace (J1) message should be written to the receive overhead location for J1. Changing this field clears *J1_AVAIL* in the *SONET_EVENT* register. *J1_AVAIL*becomes set when the J1 is actually written to the overhead register.

SDP_Mode3 Register (CP Mode Configuration Function)

Purpose Collects configuration mode bits relevant for programming the RxSDP machines.

Address 0xBC*n*0464C

Access CPRC Read/Write

Bit Position	31	30	29	28	27	26	25	24	23	22	21
Field Name	RxResetx	RxEnable	RxByteEna	RxBitEna	RxSonetEna	RxSyncEna	RxByte Loopback	RxBit Loop back	RxAgg Mode		RxSync CRC16/32
Reset Value	0	0	0	0	0	0	x	x	x		x

Bit Position	20	19	18	17	16	15	14	13	12	11	10
Field Name	RxSyncCRC Init	RxSyncFOL	RxSonet Concat	RxSync CRCInv	LOSenable	LOSmode	Rsvd	Manual_febe	RxBitInWidth		SonetOC
Reset Value	x	x	x	x	x	x	x	x	x		x

Bit Position	9	8	7	6	5	4	3	2	1	0
Field Name	SonetDscr	RxByteCRCInit	RxByteCRC 16/32	RxByteCRC Inv	RxByteFirst OnLeft	RxBitFirst OnLeft	Payload Dscr	RxFifo ExDis	Rx10Bit	Bit2Bit Source
Reset Value	x	x	x	x	x	x	x	x	x	x

Field Name	Bit Position	Description
RxResetx	31	**RxSDP Master Reset** — When 0, puts RxSDP in reset state. When 1, RxSDP is in run state. This must be low (asserted) in order to load the microcode.
RxEnable	30	**RxSDP Master Enable** — When 1, enables all Rx Sequencers. Allows all Rx processors and the SONET logic to be enabled at once.

Field Name	Bit Position	Description
RxByteEnable	29	**RxByte Processor Enable** — Enables the RxByte processor, when set to 1. Freezes the micropc at the current microaddress when disabled. Receive master enable must be set in order for this bit to have an effect. The RxByte processor should be enabled for all applications.
RxBitEnable	28	**RxBit Processor Enable** — Enables the RxBit processor, when set to 1. Freezes the processor at the current microaddress when disabled. Receive master enable must be set in order for this bit to have an effect. This processor should be enabled for all applications.
RxSonetEnable	27	**RxSONET Enable** — When a one, the SONET pointer interpreter, payload demultiplexer and overhead termination logic is enabled. When a zero, this SONET logic is disabled and bypassed. Effects operation of the loss of sync signal bit.
RxSyncEnable	26	**RxSync Enable** — Enables the rxsync processor, when set to a one. Freezes the micropc at the current microaddress when disabled. Receive master enable must be set in order for this bit to have an effect.
RxSDP Configuration	25:0	See below:
RxByteLoopback	25	**RxByte Loopback Enable** — Connects network side of transmit large FIFO to network side of receive large FIFO, when set to a one. Only works for aggregation mode = 0 (groups of one).
RxBitLoopback	24	**RxBit Loopback Enable** — Connects network side of transmit small FIFO to network side of receive small FIFO, when set to a one. Only works for aggregation mode = 0 (groups of one). The pin logic is not included in this loopback.
RxAggMode	23:22	**SDP Receive Aggregation Mode** — Controls the number of receive SDPs in a cluster that work as a unit 00 = each SDP works independently 01 = SDPs work in two groups of two 10 = SDPs work as one group of four 11 = unused
RxSyncCRC16/32	21	**RxSync CRC 16/32** — When a one, selects CRC-16 operation. When a zero, selects CRC-32 operation.
RxSyncCRCInit	20	**RxSync CRC Initialize to Ones** — When a one, resetting the CRC register sets the CRC register to all ones. When a zero, resetting the CRC register sets the CRC register to all zeroes.

Field Name	Bit Position	Description
RxSyncFOL	19	**RxSync CRC First on Left** — Calculate CRC on the data assuming that the first bit received is the left-most bit of the byte. Must be set the same as *RxBitFirstOnLeft*.
RxSonetConcat	18	**RxSONET Concatenation Mode** — When in OC-12 mode and set to a one, the receive logic is configured to SONET OC-12c / SDH STM-4 VC-4-4c (one pipe). When in OC-12 mode and set to a zero, the receive logic is configured to SONET OC-12 (four OC-3c streams) / SDH STM-4 (four VC-4-1c streams). When RXSONET OC-12/OC-3 is set to a zero (OC-3 mode), this bit must be set.
RxSyncCRCInv	17	**RxSync CRC Output Invert** — Read the ones complement of the CRC register, when a one. When a zero, read the CRC register directly.
LOSenable	16	**Loss of Synchronization Enable** — When set, the 8 bit 10 bit (TBI) decoder runs the loss of synchronization state machine on the incoming ten bit data. The loss of sync signal bit is a one if this enable is set to zero.
LOSmode	15	**Loss of Synchronization Mode** — This signal is either loss of synchronization or loss of frame as a function of other mode bits **RxSONET Enable** / **LOS Mode** / **Function** 0 / 0 / Gigabit ethernet loss of sync 0 / 1 / Fibre channel loss of sync 1 / x / All zeros received for > 2.3 µseconds
Reserved	14	Read as zero.
Manual_febe	13	**Manual FEBE** — When a one, the transmitted values of K2, M1, and G1 are completely determined by the contents of the TxSONET overhead registers. When a zero, all of M1, G1, and the far end block error portion of K2 are automatically generated by the hardware as a function of the receive data.

Field Name	Bit Position	Description
RxBitInWidth	12:11	**RxBit Input Width** — Determines the input data width received by RxBit. RxBit Input Width / Input Width in Bits 0 / 1 1 / 2 2 / 4 3 / 8 If the ten bit decoder is enabled, this field is ignored.
SonetOC	10	**RxSONET OC12/OC3 Select** — When a one, the SONET/SDH receive logic is configured to receive SONET OC-12 / SDH STM-4. When a zero, this receive logic is configured to receive SONET OC-3c / SDH STM-1 VC-4-1c.
SonetDscr	9	**SONET Descramble Enable** — Enables the SONET / SDH descramble operation, when set to a one. When a zero, no descrambling occurs. Normally enabled if SONET is enabled.
RxByteCRCInit	8	**RxByte CRC Initialize to Ones** — When a one, resetting the CRC register sets the CRC register to all ones. When a zero, resetting the CRC register sets the CRC register to all zeroes.
RxByteCRC16/32	7	**RxByte CRC 16/32** — When a one, selects CRC-16 operation. When a zero, selects CRC-32 operation.
RxByteCRCInv	6	**RxByte CRC Output Invert** — Read the ones complement of the CRC register, when a one. When a zero, read the CRC register directly.
RxByteFirstOnLeft	5	**RxByte CRC First on Left** — Calculate CRC on the data assuming that the first bit received is the left-most bit of the byte. Must be set the same as *RxBitFirstOnLeft*.
RxBitFirstOnLeft	4	**RxBit First on Left** — When a zero, the first bit on received is on the right-most bit of the byte. When a one, the first bit received is on the left-most bit of the byte. Set to zero for Ethernet. Set to one for SONET / SDH.
PayloadDscr	3	**RxSync Payload Descramble Enable** — When a one, self-synchronous payload descrambling is applied to the receive data if the microcode also enables it. When a zero, data is not descrambled.

Field Name	Bit Position	Description
RxFifoExDis	2	**RxFIFO Exception Disable** — When set to a zero, allow the RxSDP overflow handler to empty the RxSDP pipeline in the event of FIFO overflow. When set to a one, inhibit the SDP overflow handler from emptying the RxSDP in the event of FIFO overflow. For diagnostics. This bit is normally set to zero.
Rx10Bit	1	**Fix 10 Bit Decoder Mode** — When set, the 8 bit to 10 bit decoder is enabled. The rxbit input width is set to 10 bits. When clear, the 8 bit to 10 bit decoder is disabled and rxbit input width is determined by the rxbit input width field as described above. This is used for gigabit ethernet 1000BaseX and Fibre Channel.
Bit2BitSource	0	**Receive Bit-tobit Source** — When set to one 1, selects the bit-tobit signal from RxBit processor 0 of its neighbor cluster. If we call the four clusters 0, 1, 2, and 3. 0 and 1 are neighbors. Clusters 2 and 3 are also neighbors. When set to 0, selects the bit-tobit signal from an RxBit processor in the local cluster. Which RxBit processor of the four is selected is a function of the aggregation mode. The bit-tobit signal is readable by the RxBit processor.

SDP_Mode4 Register (CP Mode Configuration Function)

Purpose Configures the TxLarge FIFO and Aggregation Multiplexer.

Address 0xBC*n*04650

Access CPRC Read/Write

Bit Position	31 26	25 24	23 11	10	9	8	7	6 0
Field Name	Reserved	AgMux State	Reserved	Idle Cell	Idle Insert	Auto Token	Rsvd	Large FIFO Watermark
Reset Value	raz	0	raz	x	x	x	raz	x

Field Name	Bit Position	Description
Reserved	31:26	Read as zero.
AgMuxState	25:24	**Aggregation Multiplexer State** — Allows the base CPRC of an aggregate group to determine which CPRC of the aggregate group is currently selected by the aggregation multiplexer.
Reserved	23:11	Read as zero.

Field Name	Bit Position	Description
Idle Cell	10	**Idle Cell Mode** — If the *Idle Insertion Enable* bit is set to 1, *Idle Cell Mode* determines what is inserted in to the data stream when no other data is available. 0 = insert a packet over SONET flag character 1 = insert an ATM idle cell
Idle Insert	9	**Idle Insertion Enable** — If set to 1, the transmit pipeline inserts either a packet over SONET flag (0x7E) or an ATM idle cell (if there is no other data available) depending on the state of the idle cell mode bit. This insertion occurs either before or after the aggregation multiplexer, depending on the state of the *Scramble/Insertion mode* bit in *SDP_MODE5* register. If set to a 0, no insertion takes place.
Auto Token	8	**AutoToken Enable** — When set to 1 and the aggregation group size is four, the token is passed without micro sequencer intervention whenever *data_nine* is read from the aggregation multiplexer output stage.
Reserved	7	Read as zero.
Large FIFO Watermark	6:0	**Large FIFO Watermark** — The TxLarge FIFO (128 words) has a watermark associated with it. Starting from the empty state, the FIFO remains empty until the entry count becomes greater than the watermark at which point it then becomes not-empty. The watermark depth is recalculated each time the last byte of a packet or cell has been transmitted. The last byte is defined as any byte with the 9th bit set. If the FIFO depth is less than the watermark number of bytes when the last byte is transmitted from the FIFO, the FIFO appears empty.

SDP_Mode5 Register (CP Mode Configuration Function)

Purpose — Collects configuration mode bits relevant for programming the TxSDP machines.

Address — 0xBC*n*04654

Access — CPRC Read/Write

Bit Position	31	30	29	28	27	26	25	24	23 22
Field Name	TxResetx	TxEnable	TxByteEna	TxBitEna	TxSonetEna	ForceLineAIS	TxSonetOh Comp	ForceSonet PErr	TxAggMode
Reset Value	0	0	0	0	0	x	x	x	x

Bit Position	21	20 19	18	17	16	15	14	13	12 11	10
Field Name	TxFifoExDisab	TxBitPHY	TxSonet ConcMode	ForcePath AIS3	ForcePath AIS2	ForcePath AIS1	ForcePath AIS0	Sonet j1Mis	TxBitOut Width	TxSonet OC
Reset Value	x	x	x	x	x	x	x	x	x	x

Bit Position	9	8	7	6	5	4	3	2	1	0
Field Name	TxSonetScr	TxByteByte CRCInit1	TxByteCRC 16	TxByte CRCInv	TxByteFirst OnLeft	TxBitFirst OnLeft	Payload ScrEna	Payload ScrMode	Tx10bit	ForceLOS
Reset Value	X	x	x	x	x	x	x	x	x	x

Field Name	Bit Position	Description
TxResetx	31	**TxSDP Master Reset** — When 0, puts TxSDP in reset state. When 1, puts TxSDP in run state. Must be low (asserted) in order to load the microcode.
TxEnable	30	**TxSDP Master Enable** — When 1, enables all Rx Sequencers. This allows all of the processors and the SONET logic to be started at the same time.
TxByteEna	29	**TxByte Enable** — Enables the TxByte processor, when set to 1. Freezes the micropc at the current microaddress when disabled. *TxEnable* must be set in order for this bit to have an effect. This processor should be enabled for all applications.
TxBitEna	28	**TxBit Enable** — Enables the TxBit processor, when set to a one. Freezes the micropc at the current microaddress when disabled. Transmit master enable must be set in order for this bit to have an effect. The TxBit processor should be enabled for all applications.

Field Name	Bit Position	Description
TxSonetEna	27	**TxSONET Enable** — When one, enables the TxSONET logic. When a zero, TxSONET logic is disabled and bypassed. When enabled this logic inserts SONET transport and path overhead into the data stream.
ForceLineAIS	26	**TxSONET Force Line Alarm Indicator Signal** — When a one, force the transmission of line alarm indication signal (AIS-L). However, when set to a one when *sonet_force_path_alarm_indication_signal* is set to a one, this bit does not force AIS-L; it enables transmission of new data flag in H1.
TxSonetOhComp	25	**TxSONET Overhead Complete** — When set to a one, the final element (E2) of transmit path and transport overhead has been read from the SONET transmit overhead buffer. Indicates that one more J1 has been updated in the J1 portion of that buffer. Operation of this indicator is complicated by the following: The SONET transmit pointer is fixed at decimal 40. Because of this, a complete set of path overhead is read in this order: Z3 Z4 Z5 J1 B3 C2 G1 F2 H4, not in the expected order of J1... Z5 This bit is also available in the SONET event register.
ForceSONETPErr	24	**TxSONET Force Parity Error** — When a one, transmits parity errors on all bit lanes of B1 B2 and B3. For diagnostics.
TxAggMode	23:22	**SDP Transmit Aggregation Mode** — Controls the number of TxSDPs in a cluster that work as a unit. 00 = each SDP works independently 01 = SDPs work in two groups of two 10 = SDPs work as one group of four 11 = unused
TxFifoExDisab	21	**TXFIFO Exception Disable** — When set to 0, allow the TxSDP underflow handler to transmit well-formed, but benign PDUs in the case of FIFO underflow. When set to 1, prevent this mechanism from operating, which will cause the last data in the Tx small FIFO to be repeated until new data is available. For diagnostics. This bit is normally set to zero.
TxBitPHY	20:19	**TxBit PHY Status Select** — Selects how the physical layer status bits from the PHY or transceiver chips are connected to the TxBit processor. The TxBit processor has two PHY status tests: *phy_status_0* and *phy_status_1*. See Table 129.

Field Name	Bit Position	Description
TxSonetConcMode	18	**TxSONET Concatenation Mode** — When in OC-12 mode and set to 1, the transmit logic is configured to SONET OC-12c / SDH STM-4 VC4-4c (one pipe). When in OC-12 mode and set to zero, the transmit logic is configured to SONET OC-12 (four OC-3c streams) / SDH STM-4 (four VC-4-1c streams). When *TxSonetOC* is set to a zero (OC-3 mode), this bit must be set.
ForcePathAIS3	17	**SONET Force Path Alarm Indicator Signal 3** — When a one, forces the path alarm indication signal (AIS-P) to be transmitted. The operation of this varies according to the SONET mode selected. In OC-3c mode, the *ForcePathAIS0* is the only bit that is used. In OC-12c mode, the *ForcePathAIS0* of CP0 is the only bit that is used. In OC-12 non-concatenated mode, each of the four bits in CP0 only controls one of the four OC-3c tributaries. However, when set to 1 when the *sonet_force_line_alarm_indication_signal* is set to 1, this bit does not force AIS-P; it enables transmission of new data flag in H1.
ForcePathAIS2	16	**SONET Force Path Alarm Indicator Signal 2** — See *ForcePathAIS3*.
ForcePathAIS1	15	**SONET Force Path Alarm Indicator Signal 1** — See *ForcePathAIS3*.
ForcePathAIS0	14	**SONET Force Path Alarm Indicator Signal 0** — See *ForcePathAIS3*.
SonetJ1Mis	13	**SONET J1 Mismatch** — When 1, the path extended remote defect indication field (ERDI-P), is set to reflect this condition. This bit is to be set when the received 64Byte J1 message does not match the message that is expected.
TxBitOutWidth	12:11	**TxBit Output Width** — Determines the output data width transmitter by TxBit. **RxBit Input Width** / **Input Width in Bits** 0 / 1 1 / 2 2 / 4 3 / 8 Overridden if the ten bit encoder is enabled

Field Name	Bit Position	Description
TxSonetOC	10	**TxSONET OC12/OC3 Select** — When a one, the SONET/SDH transmit logic is configured to transmit SONET OC-12 / SDH STM-4. When a zero, this transmit logic is configured to transmit SONET OC-3c / SDH STM-1 VC-4-1c.
TxSonetScr	9	**TxSONET Scramble Enable** — When 1, the data is frame-synchronously scrambled. When zero, the data is not scrambled.
TxByteCRCInit1	8	**TxByte CRC Initialize to Ones** — When a one, resetting the CRC register sets the CRC register to all ones. When a zero, resetting the CRC register sets the CRC register to all zeroes.
TxByteCRC16	7	**TxByte CRC 16/32** — When a one, selects CRC-16 operation. When a zero, selects CRC-32 operation.
TxByteCRCInv	6	**TxByte CRC Output Invert** — Read the ones complement of the CRC register, when a one. When a zero, read the CRC register directly.
TxByteFirstOnLeft	5	**TxByte First on Left** — Calculate CRC on the data assuming that the first bit received is the left-most bit of the byte. Must be set the same as *RxBitFirstOnLeft*.
TxBitFirstOnLeft	4	**TxBit First on Left** — When 0, the first bit transmitted is the right-most bit of the byte. When 1, the first bit transmitted is the left-most bit of the byte. Set to 0 for Ethernet. Set to 1 for SONET / SDH.
PayloadScrEna	3	**TxByte Payload Scramble Enable** — When set to a one, enables the self-synchronous payload scrambler if the microcode also enables it. Otherwise, data is transferred without scrambling. Usually set if SONET is enabled.
PayloadScrMode	2	**TxByte Payload Scramble Insertion Mode** — Selects whether optional insertion and/or payload scrambling of is done before or after aggregation multiplexing. When zero, insert/scramble before multiplexing. When one, insert/scramble after multiplexing. OC-12c applications *must* set this to a one. OC-3c and OC-12 non concatenated applications *must* set this to a zero.

Field Name	Bit Position	Description
Tx10bit	1	**Tx Ten Bit Enable** — When set, the 8 bit to 10 bit encoder is enabled. The TxBit output width is set to 10 bits. When clear, the 8 bit to 10 bit encoder is disabled and TxBit output width is determined by the TxBit output width field as described below. This is used for Gigabit Ethernet 1000BaseX and Fibre Channel.
ForceLOS	0	**TxSONET Force Loss Of Signal** — When a one, turns the entire SONET frame, post-scrambling to zeroes. For diagnostics. To the receiver this looks just like a fiber cut.

Table 129 PHY Status Bit - TxBit Processor Connections

Select Field	Operation		
0 or 3	*phy_status_0* is connected to receive data signal A *phy_status_1* is connected to receive data signal B Where signals A and B are the following:		
	Pin Mode	**Signal A**	**Signal B**
	DS1/3	frame sync	undefined
	OC-3	undefined	signal detect
	OC-12	frame pulse	PLL lock detect
1	*phy_status_0* is connected to carrier sense (RMII) *phy_status_1* is connected to collision (RMII)		
2	*phy_status_0* is connected to carrier sense (GMII) *phy_status_1* is connected to collision (GMII)		

Debug_Mode Register (CP Mode Configuration Function)

Purpose Configures the CP debug tap for the global debug counters.

Address 0xBC*n*04658

Access CPRC Read/Write

Bit Position	31	30 28	27 24	23	22 20	19 16	15	14 12	11 8	7	6 4	3 0
Field Name	Enb0	Rsvd	MUX0	Enb1	Rsvd	MUX1	Enb2	Rsvd	MUX2	Enb3	Rsvd	MUX3
Reset Value	0	raz	x	0	raz	x	0	raz	x	0	raz	x

Field Name	Bit Position	Description
Enb0	31	**Global Debug Wire 0 Enable** — Enable the driver onto global debug wire 0.
reserved	30:28	Read as zero.
MUX0	27:24	**Global Debug Wire 0 MUX** — Select 1 of 16 debug events onto global debug wire 0.
Enb1	23	**Global Debug Wire 1 Enable** — Enable the driver onto global debug wire 1.
reserved	22:20	Read as zero.
MUX1	19:16	**Global Debug Wire 1 MUX** — Select 1 of 16 debug events onto global debug wire 1.
Enb2	15	**Global Debug Wire 2 Enable** — Enable the driver onto global debug wire 2.
reserved	14:12	Read as zero.
MUX2	11:8	**Global Debug Wire 2 MUX** — Select 1 of 16 debug events onto global debug wire 2.
Enb3	7	**Global Debug Wire 3 Enable** — Enable the driver onto global debug wire 3.
reserved	6:4	Read as zero.
MUX3	3:0	**Global Debug Wire 3 MUX** — Select 1 of 16 debug events onto global debug wire 3.

There are four global debug wires that carry inputs to the global debug counter block. Each CP has a multiplexor that can select one of the 16 events enumerated in Table 130 to drive on each of the respective debug wires. Each multiplexor has a 4bit register to select what event to drive, and an enable bit to turn on the debug wire driver. Chip-wide, only one debug wire driver should be enabled at any time. Because of this restriction, the recommended procedure for software to manipulate debug taps is as follows:

1 Clear the master debug enable bit in the global debug configuration register space in the XP.

2 Clear the set driver enable bits. It may be safest to invoke a routine that clears all driver enable bits on every change regardless of the previous configuration.

3 Set one chip-wide driver enable bit and its corresponding multiplexor select value for each global debug wire.

4 Set up the global debug configuration bits and master debug enable.

Table 130 Debug Multiplexor Select Encodings

MUX input	Encoding	Description
1	15	Always selects 1 for multiplexor.
0	14:8	Always selects 0 for multiplexor (unused).
SDP service request	7	Logical OR of all SDP sequencers service requests.
SDP TXByte	6	SDP TxByte sequence moved a byte of payload from DMEM.
SDP RxByte	5	SDP RxByte sequence moved a byte of payload to DMEM.
Istall	4	CPRC instruction stall cycle.
Stall	3	CPRC data read or write stall cycle.
CPRC Read	2	CPRC data read.
CPRC Write	1	CPRC data write.
DeBugMatch	0	The CPRC data, data address, or instruction address matches the programmed match registers in the CPRC.

PIN_Mode Register (CP Mode Configuration Function)

Purpose Provides programmable pin configuration state.

Address 0xBC*n*0465C

Access CPRC Read/Write

Bit Position	31 23	22 21	20 18	17 14	13 7	6 0
Field Name	Reserved	DataCntg	RxClkMUX	TxClkMUX	Reserved	TxDataEna
Reset Value	raz	0	0	0	raz	0

Field Name	Bit Position	Description
Reserved	31:23	Read as zero.
DataCnfg	22:21	**Pin Data Configuration** — Sets CP configuration pins: 0,1 = CP pins in read-only mode 2 = CP pins in lvttl mode (all other configurations) 3 = CP pins in pecl mode (OC-3)
RxClkMUX	20:18	**Receive Clock MUX Control** — For CP0, CP4, CP8, and CP12 in each cluster, the receive clock MUX control cannot be disabled (set to a value of 0) for any cluster that is to be used for either RxSDP or TxSDP processing since this is how clock is driven throughout the SDPs for that cluster. For example, if the cluster consisting of CP0 - CP3 is being used to receive data, CP0 could be set to *RxClkMux* = *TxClkMux* (5) and the *TxClkMux* must select a driven input clock. RxClkMUX configurations are: 1 = local (CPn_1) 2 = x2 aggregate (CPn_1 of the aggregation pair) 3 = TBI - recovered from CP2_1 and CP3_1 in a four CP cluster 4 = PECL (for OC-3, from CPn_0 and CPn_1 differential input) 5 = transmit clock (for RMII) 6 = inverted transmit clock (for local loopback) 7 = x4 aggregate (CP2_1 is the clock in a four CP Cluster) 0 = disable receive clock (set internally to 0)

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>TxClkMUX</td><td>17:14</td><td>Transmit Clock MUX Control — Indicates the source of the TxClock used by the SDP:
<table>
<tr><th>Encoded Value</th><th>Source</th><th>Applicable Notes</th></tr>
<tr><td>1</td><td>T1</td><td rowspan="6">N/A</td></tr>
<tr><td>2</td><td>E1</td></tr>
<tr><td>3</td><td>E3</td></tr>
<tr><td>4</td><td>T3</td></tr>
<tr><td>5</td><td>RMII</td></tr>
<tr><td>6</td><td>Fibre Channel</td></tr>
<tr><td>7</td><td>MII</td><td>To be used by MII Mode for GMII autonegotiate down to 100/10BaseT.</td></tr>
<tr><td>9</td><td>GMII/Gigabit Ethernet</td><td rowspan="2">N/A</td></tr>
<tr><td>A</td><td>OC3</td></tr>
<tr><td>B</td><td>internal0</td><td rowspan="2">Internal0 and internal1 are internally buffered versions of the receive clocks on CP4 and CP8, respectively.</td></tr>
<tr><td>C</td><td>internal1</td></tr>
<tr><td>D</td><td>receive clock</td><td>Use the 'even' receive clock in an 'even-odd' pair, for example CP8 and CP9 will use receive clock from CP8.</td></tr>
<tr><td>E</td><td>receive clock</td><td>N/A</td></tr>
<tr><td colspan="3">0,8,F = transmit clock disabled (internally set to 0)</td></tr>
</table></td></tr>
<tr><td>Reserved</td><td>13:7</td><td>Read as zero.</td></tr>
<tr><td>TxDataEna</td><td>6:0</td><td>Transmit Data Enable — Selects transmit or receive mode for each of the CP's pins:
0 = Receive
1 = Transmit
Bits 0 - 6 map individually to each of the seven pins in each CP. Bit 1 maps CPn_1, bit 2 maps to CPn_2, and so on. Thus pins 0-4 could be in Rx mode while pins 5 and 6 could be in Tx mode.</td></tr>
</table>

Queue_Status0 Register (CP Queue Status Function)

Purpose Stores queue status broadcast by the queue controller. See Table 131 on page 433 for similar registers.

Address 0xBC*n*04660

Access CPRC Read, Write one to clear

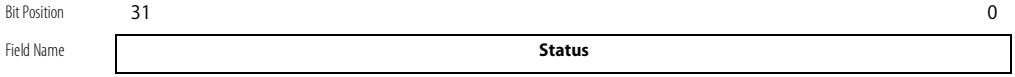


Table 131 Queue_Status*n* Registers (for Queue Status 1, 2 and 3)

Register Name	Purpose	Address
Queue_Status1	Same as *Queue_Status0*.	0xBC*n*04664
Queue_Status2	Same as *Queue_Status0*.	0xBC*n*04668
Queue_Status3	Same as *Queue_Status0*.	0xBC*n*0466C

Queue_Update0 Register (CP Queue Status Function)

Purpose Address through which bits are set in the queue status registers. See Table 132 on page 433 for similar registers.

Address 0xBC*n*04670

Access CPRC write one to set the corresponding bit of the queue status register.

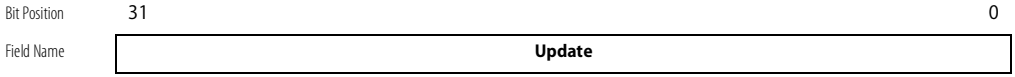


Table 132 Queue_Update*n* Registers (for Queue Updates 1, 2 and 3)

Register Name	Purpose	Address
Queue_Update1	Same as *Queue_Update0*.	0xBC*n*04674
Queue_Update2	Same as *Queue_Update0*.	0xBC*n*04678
Queue_Update3	Same as *Queue_Update0*.	0xBC*n*0467C

Event_Timer Register (CP Miscellaneous Control Function)

Purpose Cycle counter used to schedule timed events.

Address 0xBC*n*04684

Access CPRC Read/Write

Bit Position	31	0
Field Name	**Event Timer**	

Cycle_Count_H Register (CP Miscellaneous Control Function)

Purpose Most significant four bytes of the 64bit cycle counter, updated whenever Cycle_Count_L is read.

Address 0xBC*n*04688

Access CPRC Read

Bit Position	63	32
Field Name	**Frozen Count**	
Reset Value	**0**	

Cycle_Count_L Register (CP Miscellaneous Control Function)

Purpose Least significant four bytes of the 64bit cycle counter.

Address 0xBC*n*0468C

Access CPRC Read

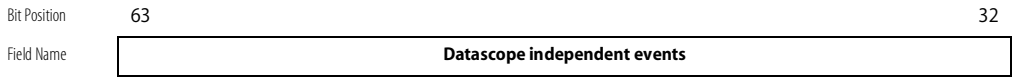
Bit Position	31	0
Field Name	**Count**	
Reset Value	**0**	

Event0 Register (CP Event and Interrupt Function)

Purpose Collects event bits relevant to datascope independent tasks.

Address 0xBC*n*046A0

Access CPRC Read, CPRC Write 1 bit to clear.



Field Name	Bit Position	Description
WindDown	63	**Wind Down** — When unmasked, this global input is a request to wind down all CP activity as soon as possible, and leave as much predictable error recovery state around as possible.
GlobalError	62	**CPRC Global Reference Error** - when asserted, this bit means a CPRC Write received an error on the Global Bus or a non-existent memory error within the cluster.
MCError	61	**Memory Controller Request Error** — Indicates an unrecoverable error occurred during a request sent to the BMU. An error status code is stored in the cp_mode_register, and also in the control block that initiated the request.
QMUError	60	**QMU Error** — Indicates an unrecoverable error occurred during a request sent to the QMU. An error status code is stored in the *cp_mode_register*.
XPInterrupt	59	**XP Interrupt Request** —The XP issued an interrupt request to this CP.
PayloadAlert	58	**Payload Request Alert** — A non-fatal bus error has occurred while trying to send a request to the BMU or QMU.
DebugMatch	57	**XPRC Debug Match** — The XPRC data, data address, and instruction address matched the programmed match registers in the CPRC.
TLUError	56	**TLU Error** — Indicates that an unrecoverable error occurred during a request sent to the TLU.

Field Name	Bit Position	Description
TxSDPError	55	**TxSDP Error** — Indicates that the SDP has encountered an error during processing. An error status code is stored in the *sdp_mode* register. This bit is the logical OR of the following conditions which are represented as bits in the *CP_MODE1* register: SDP TxBit service request [21] SDP TxByte service request [22] TxSmallFIFO underflow (SFUF) [0] TxLargeFIFO underflow (LFUF) [1] as well as these two bits: TxCtl0_Status [30] and TxCtl1_Status [30] error bit set (by TxByte)
RxSDPError	54	**RxSDP Error** — Indicates that the SDP has encountered an error during processing. An error status code is stored in the *sdp_mode* register.This bit is the logical OR of the following conditions which are represented as bits in the *CP_MODE1* register: SDP RxBit service request [25] SDP RxSync service request [26] SDP RxByte service request [27] RxSmallFIFO overflow (SFOF) [2] RxLargeFIFO overflow (LFOF) [3] as well as these two bits: RxCtl0_Status [30] and RxCtl1_Status [30] error bit set (by RxByte)
RxMsgFIFO	53	**Ring Bus Receive Message Available** — This bit indicates the availability of a Ring Bus message in the receive FIFO, and corresponds to *RxMsgCtl.State* [31].
TimerEvent	52	**Event Timer Time-out** — This bit indicates the event timer counted down to 0.
AllCpInt	51	**All CPs Interrupt Request** — One CP has interrupted all other CPs.
SonetOH	50	**SONET Overhead Event** — Masked OR of all bits in the SONET Event register. This bit is level sensitive.
Reserved	49:48	Software controlled.
RxResp7-0	47:40	**Ring Bus Receive Response Available** — These eight bits correspond to the available bit for the eight Ring Bus receive response available bits. Bit 47 represents *RxResp7Ctl.Avail*, and bit 40 represents *RxRespCtl0.Ctl*.
TxMsg3-0	39:36	**Ring Bus Transmit Message Available** — These four bits correspond to the available bit for the four Ring Bus transmit message control registers. Bit 39 represents *TxMsg3Ctl.Avail*, and bit 36 represents *TxMsg0Ctl.Avail*.

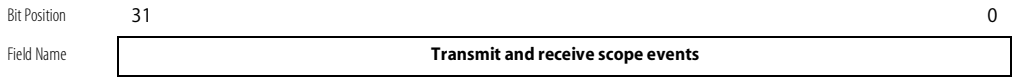
Field Name	Bit Position	Description
WrCB	35:34	**Write Control Blocks 0/1** — These two bits correspond to the available bit for the two payload bus write control blocks. Bit 35 corresponds to *WrCB1Ctl.Avail*, and bit 34 corresponds to *WrCB0Ctl.Avail*.
RdCB	33:32	**Read Control Blocks 0/1** — These two bits correspond to the available bit for the two payload bus read control blocks. Bit 33 corresponds to *RdCB1Ctl.Avail*, and bit 32 corresponds to *RdCB0Ctl.Avail*.

Event1 Register (CP Event and Interrupt Function)

Purpose Collects together event bits relevant to transmit and receive datascopes.

Address 0xBC*n*046A4

Access CPRC Read, CPRC Write 1 bits to clear



Field Name	Bit Position	Description
QRdMbxAvail	31	**Queue Read Mailbox Available** — This bit indicates that this CP's read mailbox in the QMU went from busy to available.
TxCB1_Avail	30	**TxCB1 Available** — Indicates that the available bit for datascope 1 Payload bus transmit control block *TxCB1Ctl.Avail* was set.
TxStatus1_Avail	29	**TxStatus1 Available** — Indicates that the TxSDP has set the *TxStatus1.Avail*.
TxStatus1_L1	28	**TxStatus1 Bit 1** — Indicates that the TxSDP has set the *TxStatus1* bit 1.
TxStatus1_L0	27	**TxStatus1 Bit 0** — Indicates that the TxSDP has set the *TxStatus1* bit 0.
QRdMbxBusy	26	**Queue Read Mailbox Busy** — This bit indicates that this CP's read mailbox in the QMU went from available to busy.
TxCB0_Avail	25	**TxCB0 Available** — Indicates that the available bit for datascope 0 payload bus transmit control block *TxCB0Ctl.Avail* was set.

Field Name	Bit Position	Description
TxStatus0_Avail	24	**TxStatus0 Available** — Indicates that the TxSDP has set the *TxStatus0.Avail*.
TxStatus0_L1	23	**TxStatus0 Bit 1** — Indicates that the TxSDP has set the *TxStatus0* bit 1.
TxStatus0_L0	22	**TxStatus0 Bit 0** — Indicates that the TxSDP has set the *TxStatus0* bit 0.
QueueStatus	21	**Queue Status** — This bit corresponds to the logical OR of all the bits in the four *Queue_Status* registers. The bit is level sensitive.
SoftEvent2 [4:0]	20:16	**Software Events 2 [4:0]** — These five bits correspond to events that are set explicitly by software.
QWrMbxAvail	15	**Queue Write Mailbox Available** — This bit indicates that this CP's write mailbox in the QMU went from busy to available.
RxCB1_Avail	14	**RxCB1 Available** — Indicates that the available bit for datascope 1 payload bus receive control block *RxCB1Ctl.Avail* was set.
RxStatus1_Avail	13	**RxStatus1 Available** — Indicates that the RxSDP has set the *RxStatus1.Avail*.
RxStatus1_L1	12	**RxStatus1 Bit 1** — Indicates that the RxSDP has set the *RxStatus1* bit 1.
RxStatus1_L0	11	**RxStatus1 Bit 0** — Indicates that the RxSDP has set the *RxStatus1* bit 0.
QWrMbxBusy	10	**Queue Write Mailbox Busy** — This bit indicates that this CP's write mailbox in the QMU went from available to busy.
RxCB0_Avail	9	**RxCB0 Available** — Indicates that the available bit for datascope 0 payload bus receive control block *RxCB0Ctl.Avail* was set.
RxStatus0_Avail	8	**RxStatus0 Available** — Indicates that the RxSDP has set the *RxStatus0.Avail*.
RxStatus0_L1	7	**RxStatus0 Bit 1** — Indicates that the RxSDP has set the *RxStatus0* bit 1.
RxStatus0_L0	6	**RxStatus0 Bit 0** — Indicates that the RxSDP has set the *RxStatus0* bit 0.
Reserved	5	Software controlled.
SoftEvent3 [5:0]	4:0	**Software Events 3 [5:0]** — These six bits correspond to events that are set explicitly by software. Bit 5 corresponds to software_event3_5.

Event_Mask0 Register (CP Event and Interrupt Function)

Purpose Provides mask that selects bits in the *Event0* register for event access. See Table 133 on page 439 for similar register.

Address 0xBC*n*046A8

Access CPRC Read, CPRC Write

Table 133 Event_Mask1 Register (for Mask1)

Register Name	Purpose	Address
Event_Mask1	Same as *Event_Mask0*, except it masks events [31:0]	0xBC*n*046AC

Event_Access Register (CP Event and Interrupt Function)

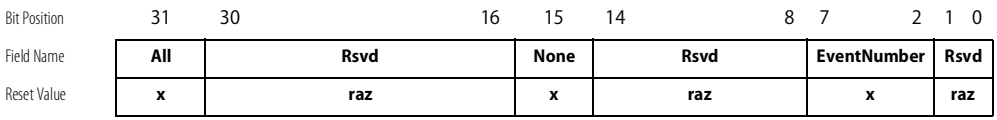
Purpose Provides access to next high bit (1) of *Event* register pair (*Event0* and *Event1*) that is set in *Event_Mask* register pair *(Event_Mask0* and *Event_Mask1*) and also provides event summaries.

Note: The fields for this register change when performing a Read as opposed to performing a Write.

Address 0xBC*n*046B0

Access CPRC Read, CPRC Write to set/clear *Event* register bit.

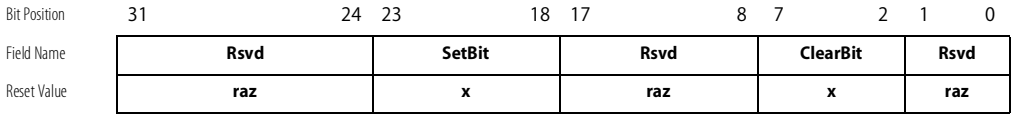
The Read Format is:



Field Name	Bit Position	Description
All	31	**All** — Provides logical-AND of *EVENT* register bits that are active in the mask.
Reserved	30:16	Read as zero.
None	15	**None** — Provides logical-NOR of *EVENT* register bits that are active in the mask.

Field Name	Bit Position	Description
Reserved	14:8	Read as zero.
EventNumber	7:2	**Event Number** — Denotes the highest number selected event.
Reserved	1:0	Read as zero.

The Write Format is:



Field Name	Bit Position	Description
Reserved	31:24	Read as zero.
SetBit	23:18	**Set Bit** — Sets an individual bit in the *EVENT_MASKn* register.
Reserved	17:8	Read as zero.
ClearBit	7:2	**Clear Bit** — Clears an individual bit in the *EVENT_MASKn* register.
Reserved	1:0	Read as zero.

Mask_Access Register (CP Event and Interrupt Function)

Purpose Provides decoder to access bits in *Event_Mask* register pair (*Event_Mask0* and *Event_Mask1*)one at a time.

Address 0xBC*n*046B4

Access CPRC Write

Bit Position	31 24	23 18	17 8	7 2	1 0
Field Name	Rsvd	SetBit	Rsvd	ClearBit	Rsvd
Reset Value	raz	x	raz	x	raz

Field Name	Bit Position	Description
Reserved	31:24	Read as zero.
SetBit	23:18	**Set Bit** — Sets an individual bit in the *EVENT_MASKn* register.
Reserved	17:8	Read as zero.
ClearBit	7:2	**Clear Bit** — Clears an individual bit in the *EVENT_MASKn* register.
Reserved	1:0	Read as zero.

Interrupt_Mask0 Register (CP Event and Interrupt Function)

Purpose Provides mask that select bits in the *Event* register pair (*Event0* and *Event1*) for interrupt reporting events [63:48] for IRQ0 and events [47:32] for IRQ1. See Table 134 on page 441 for similar register.

Address 0xBC*n*046B8

Access CPRC Read/ Write

Bit Position	31 16	15 0
Field Name	Interrupt Mask Bits [63:48] OR IRQ0	Interrupt Mask Bits [47:32] OR IRQ1

Table 134 Interrupt_Mask1 Register (for Mask Events [31:16] and [15:0])

Register Name	Purpose	Address
Interrupt_Mask1	Same as *Interrupt_Mask0*, except it masks events [31:16] for IRQ2 and events [15:0] for IRQ3.	0xBC*n*046BC

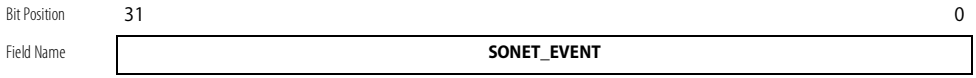
SONET_Event Register (CP Event and Interrupt Function)

Purpose — Collects together SONET event bits from the SDPs.

For all the _DELTA fields only, a 1 indicates a change in that bit's state since it was last cleared. For the bit's status, see *SDP_Mode2* register bits [31:22]. For the current values in the SONET Overhead data received, see registers starting at *Rx_SONETOH0* to *Rx_SONETOH31* in Appendix C.

Address — 0xBC*n*046C0

Access — CPRC Read, CRC Write 1 bit to clear.



Field Name	Bit Position	Description
LOS_DELTA	31	**Loss of Signal State Changed** — LOS occurs when all-zeros are detected: 2.3µs for OC-3c, or 4.6µs for OC-12. LOS clears when the FRAMELOSS bit in the control/status register (ireg14) in RxBit programmable processor is cleared.
LOF_DELTA*	30	**Loss of Framing State Changed** — Out of frame for 3ms. Turned off with framing achieved for 1-3 ms. (A1, A2). This bit is controlled through the rxBit processor within the SDP. rxBit microcode must set or clear the FRAMELOSS bit in the control/status register (ireg14) indicating whether it has achieved or lost SONET frame synchronization (SEF). The SONET block then times out this condition for 3ms (24frames) before setting the LOF bit. Requires 24 frames to clear LOS.
AIS_L_DELTA	29	**Line Alarm Indication Signal State Changed** — Five frames worth of Line AIS-L detected in bits 6, 7, and 8 of K2 byte. Bits 6, 7, and 8 must be all ones (111). Requires 5 frames to clear AIS_L.
REI_L	28	**Line Remote Error Indicator** — Far end B2 errors which were accumulated on the frame. This bit indicates that 1 or more B2 parity errors were detected by the far end on the frame this port transmitted.
RDI_L_DELTA	27	**Line Remote Defect Indicator State Changed** — Ten frames worth of Line RDI_L detected in bits 6, 7, and 8 of K2 byte. Bits 6, 7 must be ones and bit 8 must be zero (110). Requires 10 frames to clear RDI_L.

Field Name	Bit Position	Description
LOP_P_DELTA	26	**Loss Of Pointer at Path State Changed** — LOP is declared when ten frames of invalid pointers of NDF enabled indications are observed. LOP is removed when the same valid pointer with normal NDF is detected for three consecutive frames. Exit defect when 3 good pointers received.
AIS_P_DELTA	25	**Path Alarm Indication Signal State Changed** — The AIS-P signal is detected when the H1 and H2 bytes for the first STS-1 path contains an all-ones pattern in three consecutive frames. Exit defect when 3 good pointers received.
REI_P	24	**Path Remote Error Indicator** — Path is detected by extracting the 4bit *Remote Error Indication* field from the path status byte (G1). This bit indicates that 1 or more B3 parity errors were detected by the far end on the frame this port transmitted.
RDI_P_DELTA	23	**Path Remote Defect Indicator State Changed** — The RDI-P is detected by extracting bit 5 of the path status byte (G1). Path RDI is declared when the values 2, 5, or 6 are detected for ten consecutive frames.
LCD_P_DELTA	22	**Loss of Cell (or Packet) Delineation State Changed** — This bit is controlled through the rxSync processor within the SDP. rxSync microcode must set or clear the DELINLOSS bit in the control/status register (ireg14) indicating whether it has achieved or lost cell or frame delineation.
APS_ERROR	21	**APS Inconsistency Error** — In previous 12 frames, there were no three consecutive frames that had the same value for K1 or K2.
B1_ERROR	20	**B1 Section Parity Error** — Indicates that 1 or more B1 parity errors were detected on the incoming frame.
B2_ERROR	19	**B2 Line Parity Error** — Indicates that 1 or more B2 parity errors were detected on the incoming frame.
Z2_1_DELTA	18	**Z2-1 Value Changed** — Z2-1 growth byte value has changed.
Z2_0_DELTA	17	**Z2-0 Value Changed** — Z2-0 growth byte value has changed.
Z1_2_DELTA	16	**Z1-1 Value Changed** — Z1-1 growth byte value has changed.
Z1_1_DELTA	15	**Z1-0 Value Changed** — Z1-0 growth byte value has changed.
S1_DELTA	14	**S1 Value Changed** — S1 value has changed.

Field Name	Bit Position	Description
APS_DELTA	13	**APS Value Changed** — K1 or K2 value has changed and received 3 new values in a row.
J0_DELTA	12	**J0 Value Changed** — J0 value has changed.
B3_ERROR	11	**B3 Path Parity Error** — Indicates that 1 or more B3 parity errors were detected on the incoming frame.
PTR_JUST_EVENT	10	**Pointer Justification Event** — Pointer has incremented or decremented.
NDF	9	**New Data Flag in Pointer** — The most significant nibble of H1 determines this flag. If the nibble is set to 6, the SONET pointer is processed normally. If the nibble is set to 9, the pointer is set to the contents of the pointer field in H1 and the following H2. This bit indicates a NDF detection in the pointer.
C2_ERROR	8	**Path Payload Label Mismatch or Unequipped** — The PLM-P is provided when a change in the value of C2 has been detected for five consecutive frames. Software compares with previous payload label.
J1_AVAIL	7	**Receive J1 Available** — J1 is available to be read for the current frame. Cleared upon read.
Z5_DELTA	6	**Z5 Value Changed** — The value of Z5 has changed.
Z4_DELTA	5	**Z4 Value Changed** — The value of Z4 has changed.
Z3_DELTA	4	**Z3 Value Changed** — The value of Z3 has changed.
H4_DELTA	3	**H4 Value Changed** — The value of H4 has changed.
TX_OH_COMPLETE	2	**Transmit Overhead Complete** — The transmit overhead has been transmitted for the current frame. Software can use this as an indicator for when to write new transmit overhead data to the transmit SONET registers.
RX_POH_AVAIL	1	**Receive Path Overhead Available** — The path overhead for the current frame has been received and written to the receive SONET registers.
RX_TOH_AVAIL	0	**Receive Transport Overhead Available** — The section and line overhead for the current frame has been received and written to the receive SONET registers.

* The LOS indication requires the transmit logic enabled and the SONET clock set to the correct frequency.

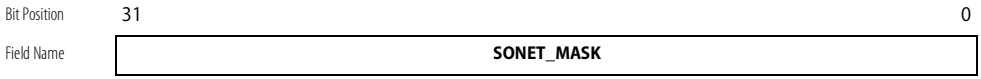
SONET_Mask Register (CP Event and Interrupt Function)

Purpose — Provides mask that selects bits in the *SONET_Event* register for event access. See bit field definitions in *SONET_Event* register.

When *SONET_Event* register logical AND *SONET_Mask* register results in any bit=1, then the CP *Event0* register *SONETOH Event* field bit [50]=1.

Address — 0xBC*n*046C4

Access — CPRC Read/ Write



Executive Processor (XP) Configuration Registers

Configuration Space in the XP is an area that contains a number of registers. The XPRC uses these registers to communicate with the SDP and the bus controllers (Payload Bus and Global Bus). The XP's registers can also be accessed by other components of the C-5 NP (all CPs).

XPSlot 24 Configuration Registers

The following is a list of each XP Slot 24 register along with its address, function, and reference to its detailed parameters. The detailed parameters provide, purpose, field name, bit position, and descriptions. Refer to Table 135 on page 447.

Table 135 XP Registers

Address	Register Name	Function	Detailed Parameters
0xBD808000	PCI Device ID	PCI Configuration	See page 456
0xBD808002	PCI Vendor ID		See page 456
0xBD808004	PCI Status		See page 456
0xBD808006	PCI Command		See page 458
0xBD808008	PCI Class Code		See page 459
0xBD80800B	PCI Revision ID		See page 460
0xBD80800D	PCI Header Type		See page 460
0xBD80800E	PCI Latency Timer		See page 461
0xBD808010	PCI Inbound Memory Base Address0		See page 461
0xBD808014	PCI Inbound Memory Base Address1		See page 462
0xBD80802C	PCI Subsystem ID (Read Only)		See page 463
0xBD80802E	PCI Subsystem Vendor ID (Read Only)		See page 463
0xBD80803E	PCI Interrupt Pin		See page 463
0xBD80803F	PCI Interrupt Line		See page 463
0xBD808040	PCI Inbound BAR0 Translation		See page 464
0xBD808044	PCI Inbound BAR1 Translation		See page 464
0xBD808048	PCI Auxiliary Control		See page 465
0xBD80804C	PCI Subsystem ID (Read/Write)		See page 465
0xBD80804E	PCI Subsystem Vendor ID (Read/Write)		See page 466
0xBD808050	PCI Inbound Byte Swap Control		See page 466
0xBD808100	Serial Bus Configuration	XP Miscellaneous Control	See page 467
0xBD808104	Serial Bus Data		See page 468
0xBD808108	XP to CP Interrupt Request		See page 469
0xBD80810C	Software Warm Reset Request		See page 470

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD808200	Outbound PCI Base Address0	XP Configuration	See page 471
0xBD808204	Outbound PCI Base Address1		
0xBD808208	Outbound PCI Base Address2		
0xBD80820C	Outbound PCI Base Address3		
0xBD808210	Outbound PCI Base Address4		
0xBD808214	Outbound PCI Base Address5		
0xBD808218	Outbound PCI Base Address6		
0xBD80821C	Outbound PCI Base Address7		
0xBD808220	Outbound BAR0 Translation		See page 472
0xBD808224	Outbound BAR1 Translation		
0xBD808228	Outbound BAR2 Translation		
0xBD80822C	Outbound BAR3 Translation		
0xBD808230	Outbound BAR4 Translation		
0xBD808234	Outbound BAR5 Translation		
0xBD808238	Outbound BAR6 Translation		
0xBD80823C	Outbound BAR7 Translation		
0xBD808240	DMA Transmit Channel0 PCI Target		See page 473

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD808244	DMA Transmit Channel1 PCI Target	XP Configuration (continued)	See Table 139 on page 474
0xBD808248	DMA Receive Channel0 PCI Target		See page 474.
0xBD80824C	DMA Receive Channel0 PCI Target Count		See page 474
0xBD808250	DMA Receive Channel1 PCI Target		See Table 140 on page 474
0xBD808254	DMA Receive Channel1 PCI Target Count		See Table 141 on page 475
0xBD808258	XP Miscellaneous Control		See page 476.
0xBD80825C	XP Auxiliary Event		See page 477
0xBD808260	Inbound PCI Mailbox0		See page 478
0xBD808264	Inbound PCI Mailbox1		
0xBD808268	Inbound PCI Mailbox2		
0xBD80826C	Inbound PCI Mailbox3		
0xBD808270	Inbound PCI Mailbox4		
0xBD808274	Inbound PCI Mailbox5		
0xBD808278	Inbound PCI Mailbox6		
0xBD80827C	Inbound PCI Mailbox7		
0xBD808280	IMEM Overlay Target Address		See page 479
0xBD808284	RxCB #25 Transfer Count		See page 479
0xBD808288	XP Diagnostic		See page 480
0xBD80828C	PCI Outbound Byte Swap Control		See page 480
0xBD808300	Debug Counter0 Start Value	XP Configuration (continued)	See page 481
0xBD808304	Debug Counter1 Start Value		See Table 144 on page 484
0xBD808308	Debug Counter2 Start Value		
0xBD80830C	Debug Counter3 Start Value		

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD808340	Debug Counter0 Control	XP Configuration (continued)	See page 483
0xBD808344	Debug Counter1 Control		See Table 144 on page 484
0xBD808348	Debug Counter2 Control		
0xBD80834C	Debug Counter3 Control		
0xBD808380	Debug Counter0 Current Value	XP Configuration (continued)	See page 485
0xBD808384	Debug Counter1 Current Value		See Table 145 on page 485
0xBD808388	Debug Counter2 Current Value		
0xBD80838C	Debug Counter3 Current Value		
0xBD804080	RxCB0_Sys_Addr	XP DMEM#24 Transfer RxControl Block0 These register are used to set up a DMA transaction from the PCI bus to SDRAM via DMEM#24. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804084	RxCB0_Ctl		
0xBD804088	RxCB0_DMA_Addr		
0xBD80408C	RxCB0_SDP_Addr		
0xBD804090	RxCtl0_Status	XP DMEM#24 Transfer RxControl Block0 (continued)	See page 486
0xBD804180	TxCB0_Sys_Addr	XP DMEM#24 Transfer TxControl Block0 (continued) These registers are used to set up a DMA transaction from the SDRAM to the PCI bus via DMEM#24. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378 and page 486
0xBD804184	TxCB0_Ctl		
0xBD804188	TxCB0_DMA_Addr		
0xBD80418C	TxCB0_SDP_Addr		

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804190	TxCtl0_Status	XP DMEM#24 Transfer TxConttrol Block0 (continued)	See page 487
0xBD804280	RxCB1_Sys_Addr	XP DMEM#24 Transfer RxControl Block1 These register are used to set up a DMA transaction from the PCI bus to SDRAM via DMEM#24. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804284	RxCB1_Ctl		
0xBD804288	RxCB1_DMA_Addr		
0xBD80428C	RxCB1_SDP_Addr		
0xBD804290	RxCtl1_Status	XP DMEM#24 Transfer RxControl Block1 (continued)	See Table 146 on page 486
0xBD804380	TxCB1_Sys_Addr	XP DMEM#24 Transfer TxControl Block1 (continued) These registers are used to set up a DMA transaction from the SDRAM to the PCI bus via DMEM#24. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378 and Table 147 on page 487
0xBD804384	TxCB1_Ctl		
0xBD804388	TxCB1_DMA_Addr		
0xBD80438C	TxCB1_SDP_Addr		
0xBD804390	TxCB1_Status	XP DMEM#24 Transfer TxControl Block1 (continued)	See Table 148 on page 487

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804400	WrCB0_Sys_Addr	XP DMEM#24 Transfer WrControl Block0 (continued) These register are identical to their counterparts in the CP except their addresses are different	See "CP Registers" on page 378
0xBD804404	WrCB0_Ctl		
0xBD804408	WrCB0_DMA_Addr		
0xBD804410	WrCB1_Sys_Addr	XP DMEM#24 Transfer WrControl Block1 (continued) These register are identical to their counterparts in the CP except their addresses are different	See "CP Registers" on page 378
0xBD804414	WrCB1_Ctl		
0xBD804418	WrCB1_DMA_Addr		
0xBD804420	RdCB0_Sys_Addr	XP DMEM#24 Transfer RdControl Block0 (continued)	See "CP Registers" on page 378
0xBD804424	RdCB0_Ctl		
0xBD804428	RdCB0_DMA_Addr		
0xBD804430	RdCB1_Sys_Addr	XP DMEM#24 Transfer RdControl Block1 (continued)	See "CP Registers" on page 378
0xBD804434	RdCB1_Ctl		
0xBD804438	RdCB1_DMA_Addr		
0xBD804440 to 0xBD8044E4	XP Ring Bus Control Configuration Registers	These registers are identical to their counterparts in the CP except their addresses are different.	See "CP Registers" on page 378
0xBD804640	XP_Mode	XP Mode Configuration	See page 488
0xBD804658	Debug_Mode		See page 490

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804660	Queue_Status0	Queue Status These register are identical to their counterparts in the CP except their addresses are different.	See "CP Registers" on page 378
0xBD804664	Queue_Status1		
0xBD804668	Queue_Status2		
0xBD80466C	Queue_Status3		
0xBD804670	Queue_Update0		
0xBD804674	Queue_Update1		
0xBD804678	Queue_Update2		
0xBD80467C	Queue_Update3		
0xBD804684	Event_Timer	Miscellaneous Control These register are identical to their counterparts in the CP except their addresses are different.	See "CP Registers" on page 378
0xBD804688	Cycle_Counter_H		
0xBD80468C	Cycle_Counter_L		
0xBD8046A0	Event0	Event and Interrupt Control	See page 492
0xBD8046A4	Event1		See page 493
0xBD8046A8	Event_Mask0	Event and Interrupt Control (continued) These register are identical to their counterparts in the CP except their addresses are different.	See "CP Registers" on page 378
0xBD8046AC	Event_Mask1		
0xBD8046B0	Event_Access		
0xBD8046B4	Mask_Access		
0xBD8046B8	Interrupt_Mask0		
0xBD8046BC	Interrupt_Mask1		
0xBD804880	RxCB0_Sys_Addr	XP DMEM#25 Transfer RxControl Block0 (continued) Theses registers are used to initialize SDRAM. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804884	RxCB0_Ctl		
0xBD804888	RxCB0_DMA_Addr		
0xBD80488C	RxCB0_SDP_Addr		

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804890	RxCtl0_Status	XP DMEM#25 Transfer RxControl Block0 (continued)	See page 496
0xBD804980	TxCB0_Sys_Addr	XP DMEM#25 Transfer TxControl Block0 (continued) These register are used to set up a DMA transaction from the SDRAM to the XP's IMEM. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804984	TxCB0_Ctl		
0xBD804988	TxCB0_DMA_Addr		
0xBD80498C	TxCB0_SDP_Addr		
0xBD804990	TxCtl0_Status	XP DMEM#25 Transfer Control Block0 (continued)	See page 497
0xBD804A80	RxCB1_Sys_Addr	XP DMEM#25 Transfer Control Block1 (continued) Theses registers are used to initialize SDRAM. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804284	RxCB1_Ctl		
0xBD804A88	RxCB1_DMA_Addr		
0xBD804A8C	RxCB1_SDP_Addr		
0xBD80A290	RxCB1_Status	XP DMEM#25 Transfer Control Block1 (continued)	See page 496

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804B80	TxCB1_Sys_Addr	XP DMEM#25 Transfer TxControl Block1 (continued) These register are used to set up a DMA transaction from the SDRAM to the XP's IMEM. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804B84	TxCB1_Ctl		
0xBD804B88	TxCB1_DMA_Addr	XP DMEM#25 Transfer TxControl Block1 (continued) These register are used to set up a DMA transaction from the SDRAM to the XP's IMEM. The fields of these registers are identical to their counterparts in the CP.	See "CP Registers" on page 378
0xBD804B8C	TxCB1_SDP_Addr		
0xBD804B90	TxCB1_Status	XP DMEM#25 Transfer Control Block1 (continued)	See page 497
0xBD804C00	WrCB0_Sys_Addr	XP DMEM#25 Transfer Control Block0 (continued) These register are identical to their counterparts in the CP except their addresses are different	See "CP Registers" on page 378
0xBD804C04	WrCB0_Ctl		
0xBD804C08	WrCB0_DMA_Addr		

Table 135 XP Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBD804C10	WrCB1_Sys_Addr	XP DMEM#25 Transfer Control Block1 (continued) These register are identical to their counterparts in the CP except their addresses are different	See "CP Registers" on page 378
0xBD804C14	WrCB1_Ctl		
0xBD804C18	WrCB1_DMA_Addr		
0xBD804C20	RdCB0_Sys_Addr	XP DMEM#25Transfer Control Block0 (continued)	See "CP Registers" on page 378
0xBD804C24	RdCB0_Ctl		
0xBD804C28	RdCB0_DMA_Addr		
0xBD804C30	RdCB1_Sys_Addr	XP DMEM#25 Transfer Control Block1 (continued)	See "CP Registers" on page 378
0xBD804C34	RdCB1_Ctl		
0xBD804C38	RdCB1_DMA_Addr		

XP Detailed Descriptions

The following is a detailed description of each of the XPSlot 24 registers and their individual parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

PCI Device ID Register (XP PCI Configuration Function)

Purpose Uniquely Identifies the Device.

Address 0xBD808000

Access Read only

PCI Vendor ID Register (XP PCI Configuration Function)

Purpose Uniquely Identifies the Device Vendor.

Address 0xBD808002

Access Read only

PCI Status Register (XP PCI Configuration Function)

Purpose Captures Status Information for PCI bus related events.

Address 0xBD808004

Access Read/Write, write 1 to clear

Bit Position	15	14	13	12	11	10	9	8	7	6	5	4	3	0
Field Name	DPE	SSE	RMA	RTA	STA	DEVSEL		DPE	FBC	UDF	66M	NCP	Rsvd	
Reset Value	0	0	0	0	0	01		0	1	0	1	0	0000	

Field Name	Bit Position	Description
DPE	15	**Detected Parity Error** — This bit is set by the device whenever it detects a parity error, even if parity error handling is disabled (as controlled by bit 6 in the Command Register).
SSE	14	**Signaled System Error** — This bit is set whenever the device asserts *SERR#*.
RMA	13	**Received Master Abort** — This bit is set by the master whenever its transaction is terminated with a Master Abort.
RTA	12	**Received Target Abort** — This bit is set by the master whenever its transaction is terminated with a Target Abort.
STA	11	**Signaled Target Abort** — This bit is set by the target whenever it terminates a transaction wit a Target Abort.
DEVSEL	10:9	**DEVSEL Timing** — These two read-only bits are hardwired to "01" indicating the medium DEVSEL response time of this device.
DPE	8	**Data Parity Error Detected** — This bit is set when three conditions are met: 1) this device asserted *PERR#* itself or observed *PERR#* asserted; 2) this devices was acting as the bus master for the operation in which the error occurred; 3) the Parity Error Response bit (Command Register) is set.
FBC	7	**Fast Back-to-Back Capable** — This read-only bit is hardwired to 1 indicating that this device is capable of performing fast back-to-back transactions.
UDF	6	**UDF Supported** — This read-only bit is hardwired to 0 indicating that User Definable Features is not supported.
66M	5	**66MHz Capable** — This read-only bit is hardwired to 1 indicating that this device is capable of 66MHz operation.
NCP	4	**New Capabilities Pointer Support** — This read-only bit is hardwired to 0 indicating that there are no new capabilities pointers supported in the configuration register space.
Reserved	3:0	Read as zero.

PCI Command Register (XP PCI Configuration Function)

Purpose Provides control over the device's ability to generate and respond to PCI transactions.

Address 0xBD808006

Access Read/Write

Bit Position	15 10	9	8	7	6	5	4	3	2	1	0
Field Name	Rsvd	FB2B	SERR	WAIT	PERR	Rsvd	MWI	SPC	MST	MEM	IO
Reset Value	000000	0	0	0	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Reserved	15:10	Read as zero.
FB2B	9	**Fast Back-to-Back** — When the bit is 0, the PCI Master will generate no fast back-to-back transactions. When the bit is 1, the PCI Master will generate fast back-to-back transactions whenever possible.
SERR	8	**System Error** — When the bit is 0, the System Error indication is masked off. When the bit is 1, all System Errors will be reported.
WAIT	7	**Wait** — This bit is used to control whether or not a device does address/data stepping. This device never does address/data stepping, therefore this bit is hardwired to 0 and is read-only.
PERR	6	**Parity Error** — This bit controls the device's response to parity errors. When the bit is set to 1, the device will take its normal action when a parity error is detected. When the bit is 0, all parity errors are ignored.
Reserved	5	Read as zero.
MWI	4	**Memory Write and Invalidate** — This is an enable bit for using the Memory Write and Invalidate command. This device will never generate a MWI command, therefore this bit is hardwired to 0 and is read-only.
SPC	3	**Special Cycles** — This device always ignores special cycles, therefore this bit is hardwired to 0 and is read-only.

Field Name	Bit Position	Description
MST	2	**PCI Master** — This bit controls the device's ability to act as a master on the PCI bus. When the bit is 0, it disables the device from generating PCI accesses. When the bit is 1, it allows the device to behave as a bus master.
MEM	1	**Memory Space** — This bit controls the device's response to Memory Space accesses. When the bit is 0, it disables the device's response. When the bit is 1, it allows the device to respond to Memory Space accesses
IO	0	**Input/Output Mapping** — This device requires no I/O spacing mappings, therefore this bit is hardwired to 0 and is read-only.

PCI Class Code Register (XP PCI Configuration Function)

Purpose Identifies the generic function of the device.

Address 0xBD808008

Reset Value 0xFF0000

Access Read only

Bit Position	31 — 24	23 — 16	15 — 8
Field Name	**BCC**	**SCC**	**RLPI**
Reset Value	**0xFF**	**0x00**	**0x00**

Field Name	Bit Position	Description
BCC	31:24	**Base Class Code** — Hardwired to FFh indicating that the device does not fit in any of the PCI pre-defined classes.
SCC	23:16	**Sub-Class Code** — Hardwired to 00h. This field has no real meaning since the device has a base class code of FFh.
RLPI	15:8	**Register-Level Programming Interface** — Hardwired to 00h. This field has no real meaning since the device has a base class code of FFh

PCI Revision ID Register (XP PCI Configuration Function)

Purpose Provides a device specific revision identifier.

Address 0xBD80800B

Reset Value See Table 136.

Access Read only

Bit Position	7	0
Field Name	Revision	
Reset Value	0x40	

Field Name	Bit Position	Description
Revision	7:0	**Revision** — Indicates the revision level of the device.

Table 136 PCI Revision ID Register Reset Values

Revision	Reset Value
A0 (1.0)	0x10
A1 (1.0.1)	0x11
A2 (1.0.2)	0x12
B0 (1.1)	0x20
C0 (1.2)	0x30
D0 (1.3)	0x40

PCI Header Type Register (XP PCI Configuration Function)

Purpose Identifies the PCI Configuration Register layout.

Address 0xBD80800D

Reset Value 0x00

Access Read only

PCI Latency Timer Register (XP PCI Configuration Function)

Purpose Limits the master's tenure on the bus in the presence of other bus access requests.

Address 0xBD80800E

Reset Value 0x00

Access Read/Write

Bit Position	7	3	2	0
Field Name	LAT		Rsvd	
Reset Value	00000		000	

Field Name	Bit Position	Description
LAT	7:4	**Latency Timer Value** — The high order 5bits of an 8bit latency timer counting the number of PCI clock cycles that the master has tenure on the PCI bus.
Reserved	2:0	Read as zero.

PCI Inbound Memory Base Address Register0 (XP PCI Configuration Function)

Purpose Provides the Base Address for a 1Mbyte window into C-5 NP Address Space.

Address 0xBD808010

Reset Value 0x0008

Access Read/Write

Bit Position	31	20	19	4	3	2	1	0
Field Name	BA		Rsvd		PREF	TYPE		IO/M
Reset Value	0x000		0x0_000		1	00		0

Field Name	Bit Position	Description
BA	31:20	**Base Address** — Provides the top 12bits of a 1MByte aligned address used for decoding and to identify PCI bus transactions for which this device is to act as a target.
Reserved	19:4	Read as zero.

Field Name	Bit Position	Description
PREF	3	**Prefetchable** — This read-only bit is hardwired to 1 indicating that there are no side effects on reads. The device returns all bytes on reads regardless of byte enables, and host bridges can merge processor writes into this range without causing errors.
TYPE	2:1	**Address Type** — These read-only bits are hardwired to 00 indicating that the base address can be set to locate this window anywhere in the 32bit PCI address space.
IO/M	0	**I/O or Memory Indicator** — This read-only bit is hardwired to 0 to indicate that this register is a memory space base address register.

PCI Inbound Memory Base Address Register1 (XP PCI Configuration Function)

Purpose Provides the Base Address for a 1Mbyte window into C-5 NP Address Space.

Address 0xBD808014

Reset Value 0x0008

Access Read/Write

Bit Position	31 20	19 4	3	2 1	0
Field Name	BA	Rsvd	PREF	TYPE	IO/M
Reset Value	0x000	0x0_000	1	00	0

Field Name	Bit Position	Description
BA	31:20	**Base Address** — Provides the top 12bits of a 1Mbyte aligned address used for decoding, and to identify PCI bus transactions for which this device is to act as a target.
Reserved	19:4	Read as zero.
PREF	3	**Prefetchable** — This read-only bit is hardwired to 1 indicating that there are no side effects on reads. The device returns all bytes on reads regardless of byte enables, and host bridges can merge processor writes into this range without causing errors.
TYPE	2:1	**Address Type** — These read-only bits are hardwired to 00 indicating that the base address can be set to locate this window anywhere in the 32bit PCI address space.

Field Name	Bit Position	Description
IO/M	0	**I/O or Memory Indicator** — This read-only bit is hardwired to 0 to indicate that this register is a memory space base address register.

PCI Subsystem ID Register (Read Only) (XP PCI Configuration Function)

Purpose Used to uniquely identify the subsystem where the PCI device resides.

Address 0xBD80802C

Reset Value 0x0000

Access Read only

PCI Subsystem Vendor ID Register (Read Only) (XP PCI Configuration Function)

Purpose Used to uniquely identify the vendor of the subsystem where the PCI device resides.

Address 0xBD80802E

Reset Value 0x0000

Access Read only

PCI Interrupt Pin Register (XP PCI Configuration Function)

Purpose Indicates that interrupt pin INTA# is the one being used.

Address 0xBD80803E

Reset Value 0x01

Access Read/Write

PCI Interrupt Line Register (XP PCI Configuration Function)

Purpose Used to communicate interrupt line routing information.

Address 0xBD80803F

Reset Value 0x00

Access Read/Write

PCI Inbound BAR0 Translation Register (XP PCI Configuration Function)

Purpose Provides Address Translation and Control for the PCI Inbound Memory Base Address Register 0 [Address: 10-13h].

Address 0xBD808040

Reset Value 0xA000

Access Read/Write

Bit Position	31 29	28 20	19 0
Field Name	101	TRANS	Rsvd
Reset Value	101	0_0000_0000	0000_0000_0000_0000_0000

Field Name	Bit Position	Description
101	31:29	**"101" Translation Address** — Bits 31 through 29 of the transaction address are always translated to 101. This limits the translated address to the range of 0xA0000000 to 0xBFFFFFFF.
TRANS	28:20	**Translation Address** — Provides the replacement values for bits 28 through 20 of the transaction address to translate it to a different 1MByte window in C-5 NP Address Space.
Reserved	19:0	Read as zero.

PCI Inbound BAR1 Translation Register (XP PCI Configuration Function)

Purpose Provides Address Translation and Control for the PCI Inbound Memory Base Address Register 1 [Address: 14-17h].

Address 0xBD808044

Reset Value 0xA000

Access Read/Write

Bit Position	31 29	28 20	19 0
Field Name	101	TRANS	Rsvd
Reset Value	101	0_0000_0000	0000_0000_0000_0000_0000

Field Name	Bit Position	Description
101	31:29	**"101" Translation Address** — Bits 31 through 29 of the transaction address are always translated to 101. This limits the translated address to the range of 0xA0000000 to 0xBFFFFFFF.

Field Name	Bit Position	Description
TRANS	28:20	**Translation Address** — Provides the replacement values for bits 28 through 20 of the transaction address to translate it to a different 1MByte window in C-5 NP Address Space.
Reserved	19:0	Read as zero.

PCI Auxiliary Control Register (XP PCI Configuration Function)

Purpose Provides Control for Miscellaneous Functions within the PCI interface.

Address 0xBD808048

Reset Value 0x00

Access Read/Write

Bit Position	31 … 3	2	1	0
Field Name	Rsvd	P2S	MA2S	TA2S
Reset Value	0000_0000_0000	0	0	0

Field Name	Bit Position	Description
Reserved	31:3	Read as zero.
P2S	2	**Map Parity Errors to SERR#** — When the bit is 1, any PCI parity error that is detected and reported will result in the device signaling SERR#.
MA2S	1	**Map Master Abort to SERR#** — When the bit is 1, whenever the master has a transaction terminated by a Master Abort, the device will signal SERR#.
TA2S	0	**Map Target Abort to SERR#** — When the bit is 1, whenever the master has a transaction terminated by a Target Abort the device will signal SERR#.

PCI Subsystem ID Register (XP PCI Configuration Function)

Purpose Writable image of the PCI Subsystem ID.

Address 0xBD80804C

Reset Value 0x0000

Access Read/Write

PCI Subsystem Vendor ID Register (XP PCI Configuration Function)

Purpose Writable image of the PCI Subsystem Vendor ID.

Address 0xBD80804E

Reset Value 0x0000

Access Read/Write

PCI Inbound Byte Swap Control Register (XP PCI Configuration Function)

Purpose Provides control of the byte swapping feature for inbound transactions.

Address 0xBD808050

Reset Value 0x0000

Access Read/Write

Bit Position	31	2	1	0
Field Name	**Rsvd**		**SWAP1**	**SWAP0**
Reset Value	**0000_0000_0000_0000_0000_0000_0000_00**		**0**	**0**

Field Name	Bit Position	Description
Reserved	31:2	Read as zero.
SWAP1	1	**BAR1 Byte Swap Enable** — When this bit is set to a 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Inbound Base Address 1* register.
SWAP0	0	**BAR0 Byte Swap Enable** — When this bit is set to a 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Inbound Base Address 1* register.

Serial Bus Configuration Register (XP Miscellaneous Control Function)

Purpose Sets the configuration of the Serial Bus.

Address 0xBD808100

Reset Value 0x000001F4

Access Read/Write

Bit Position	31 … 13	12	11	10	9	8 … 0
Field Name	Reserved	MDIO_Cycles	EN	PROTOCOL	PRE_SUPP	CLKDIV
Reset Value	0	0	0	0	0	1_1111_0100

Field Name	Bit Position	Description
Reserved	31:13	Read as zero.
MDIO_Cycles	12	**MDIO Turn Around Cycles** — When deasserted, which is the default, enables one turnaround cycle. When asserted, enables two turnaround cycles on reads.
EN	11	**Serial Bus Enable** — This is an enable control bit for the serial bus. When the bit is 1, the bus is enabled.
PROTOCOL	10	**Protocol Select** — Determines which of 2 possible protocols will be followed by the C-5 NP on the serial bus. A value of 0 selects the MDIO protocol and a value of 1 selects the low-speed serial bus protocol.
PRE_SUPP	9	**MDIO Preamble Suppression** — When this bit is 1, the 32bit preamble pattern will be skipped during MDIO transfers.
CLKDIV	8:0	**Core Clock Divider** — With the value *N* programmed into this 9bit field, the C-5 NP core clock will be divided by a factor of *4xN* to form the Serial bus clock. The default value of "*500*" causes the 166 MHz core clock to be divided by 4*500, generating a 83kHz Serial bus clock. Note: That when this *CLKDIV* field equals 0, the divider will be 512, resulting in a factor of 4*512=2048.

Serial Bus Data Register (XP Miscellaneous Control Function)

Purpose Specifies the address, data, and write/read of a serial bus transfer. A transfer will be initiated when the REQ bit gets set.

Address 0xBD808104

Reset Value 0x00000000

Access Read/Write

Bit Position	31	30	29	28	27	18	17	16	15	0
Field Name	REQ	Rsvd		WR	ADDR		AACK	DACK	DATA	
Reset Value	0	00		0	00_0000_0000		0	0	0000_0000_0000_0000	

Field Name	Bit Position	Description
REQ	31	**Request** — Indicates that a serial bus transfer is pending. Software should set this bit to a 1 to begin a serial bus transfer. When the *REQ* bit is set, the transfer will be performed using the values of *WR* and *ADDR* in this register, and using the configuration from the Serial Bus Configuration Register. The *REQ* bit will remain a 1 until the transfer is done (also indicated by the *SB_TRANSFER_DONE* bit in XP *EVENT0*), after which it is cleared by the hardware. Writes to the serial bus data register are delayed until the serial bus is idle (and the *REQ* bit is deasserted). It is acceptable to set the *REQ* bit with the same register access that sets up the proper values of *WR*, *ADDR*, and *DATA* fields.
Reserved	30:29	Read as zero.
WR	28	**Write** — Specifies whether the serial bus transfer is a write or a read: 1 = write, 0 = read.
ADDR	27:18	**Address** — Specifies the address of the serial bus transfer. In MDIO mode, bits [27:23] act as the PHY address, and bits [22:18] act as the register address. In low-speed serial bus mode, bits [24:18] make up the 7bit address (bits [27:25] are unused).
AACK	17	**Address Acknowledge** — This bit indicates the address acknowledge value captured on the serial bus. It is read-only (and is not affected by writes to this register) and applies only to the low-speed serial bus mode. When the bit is a 1, the previously attempted access failed due to a missing address acknowledgement.
DACK	16	**Data Acknowledge** — This bit indicates the data acknowledge value captured on the serial bus. It is read-only (and is not affected by writes to this register) and applies only to low-speed serial bus mode. When the bit is a 1, the previously attempted access failed due to a missing data acknowledgement.

Field Name	Bit Position	Description
DATA	15:0	**Data** — For writes to the serial bus, this field indicates the data to be written. For reads from the serial bus, this field contains the data that was read from the bus. The data is valid once the transfer is done, as indicated by the deassertion of the *REQ* bit and also by the *SB_TRANSFER_DONE* bit in XP *EVENT0*. Note: That bits [15:0] are valid for MDIO mode. For the low-speed serial bus mode, bits [15:8] are undefined and only bits [7:0] are valid.

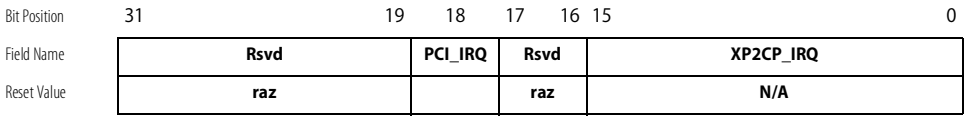
XP to CP Interrupt Request Registers (XP Miscellaneous Control Function)

Purpose Initiates Interrupt Requests from the XP to individual CPs

Address 0xBD808108

Reset Value 0x00

Access Write only



Field Name	Bit Position	Description
Reserved	31:19	Read as zero.
PCI_IRQ	18	**PCI Interrupt Request** — Provides a way for software to cause a PCI interrupt. When set to 1, the PCI Interrupt Line will be asserted.
Reserved	17:16	Read as zero.
XP2CP_IRQ	15:0	**XP to CP Interrupt Request Vector** — When the bit is 1, an interrupt request is sent to the CP corresponding to the bit number.

Software Warm Reset Request Register (XP Miscellaneous Control Function)

Purpose Trigger Reset Sequences for the C-5 NP and the XP.

Address 0xBD80810C

Reset Value 0x00

Access Write only

Bit Position	31	30	29	28 ... 0
Field Name	DCPRST	XPURST	WARM_XPUHOT	Rsvd
Reset Value	N/A	N/A	N/A	N/A

Field Name	Bit Position	Description
NPRST	31	**NP Warm Reset Request** — When the bit is 1, a state machine is triggered that waits for PCI inbound and outbound activity to idle, asserts reset to the entire C-5 NP, waits for 128 C-5 NP core clock cycles, and then deasserts reset.
XPURST	30	**XP Warm Reset Request** — When the bit is 1, a state machine is triggered that waits for XP activity to idle, asserts reset to the XPRC, waits for 128 C-5 NP core clock cycles, and then deasserts reset.
WARM_XPUHOT	29	**Warm XPUHOT** — When a C-5 NP warm reset or XP warm reset is requested (using one of the above bits), this bit specifies whether the XP will be turned on (XPUHOT=1) or off (XPUHOT=0) after the warm reset completes. This only applies to warm resets; on cold resets, the XP will be turned on based on how the XPUHOT external pin is sampled.
Reserved	28:0	Read as zero

Outbound PCI Base Address0 Register (XP Configuration Function)

Purpose Provides the Base Address for a variable size window for the XP into PCI Space. See Table 137 on page 471 for similar registers.

Address 0xBD808200

Reset Value 0x00

Access Read/Write

Bit Position	31	14	13	0
Field Name	BA		Rsvd	
Reset Value	0000_0000_0000_0000_00		00_0000_0000_0000	

Field Name	Bit Position	Description
BA	31:14	**Base Address** — Provides the high order address bytes used for decoding an access by the XP into a window into PCI Space. The number of bits used in the decode depends on the size specification for the window.
Reserved	13:0	Read as zero.

Table 137 Outbound PCI Base Address*n* Registers (for BAR 1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
Outbound PCI Base Address1	Same as *Outbound PCI Base Address0 Register*, except for BAR 1.	0xBD808204
Outbound PCI Base Address2	Same as *Outbound PCI Base Address0 Register*, except for BAR 2.	0xBD808208
Outbound PCI Base Address3	Same as *Outbound PCI Base Address0 Register*, except for BAR 3.	0xBD80820C
Outbound PCI Base Address4	Same as *Outbound PCI Base Address0 Register*, except for BAR 4.	0xBD808210
Outbound PCI Base Address5	Same as *Outbound PCI Base Address0 Register*, except for BAR 5.	0xBD808214
Outbound PCI Base Address6	Same as *Outbound PCI Base Address0 Register*, except for BAR 6.	0xBD808218
Outbound PCI Base Address7	Same as *Outbound PCI Base Address0 Register*, except for BAR 7.	0xBD80821C

Outbound BAR0 Translation Register (XP Configuration Function)

Purpose — Provides the Translation Address and control for a variable size window for the XP into the PCI Space. See Table 138 on page 473 for similar registers.

Address — 0xBD808220

Reset Value — 0x00

Access — Read/Write

Bit Position	31 — 14	13 — 4	3 — 1	0
Field Name	TRANS	Rsvd	SIZE	EN
Reset Value	0000_0000_0000_0000_00	raz	000	0

Field Name	Bit Position	Description
TRANS	31:14	**Translation Address** — Provides the high order address bits to replace the high order address bits from the original address to create an address in PCI space. The number of bits replaced depends on the size specification for the window.
Reserved	13:4	Read as zero
SIZE	3:1	**Window Size** — Specifies the size of the address region viewed by the window. Note: That the base address and the translation address will be interpreted as being size aligned. **Encoded Value** / **Window Size** 000 — 16 kB 001 — 32 kB 010 — 64 kB 011 — 128 kB 100 — 256 kB 101 — 512 kB 110 — 1 MB 111 — 2 MB

Field Name	Bit Position	Description
EN	0	**BAR Enable** — This bit controls whether or not the corresponding BAR is used to decode XP accesses to PCI. When the bit is 1, the corresponding BAR is used in transaction decode. When the bit is 0, the BAR will be ignored.

Table 138 Outbound BAR*n* Translation Registers (for BAR1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
Outbound Bar1 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR1.	0xBD808224
Outbound Bar2 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR2.	0xBD808228
Outbound Bar3 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR3.	0xBD80822C
Outbound Bar4 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR4.	0xBD808230
Outbound Bar5 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR5.	0xBD808234
Outbound Bar6 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR6.	0xBD808238
Outbound Bar7 Translation	Same as *Outbound BAR0 Translation Register*, except for BAR7.	0xBD80823C

DMA Transmit Channel0 PCI Target Register (XP Configuration Function)

Purpose Provides the PCI Target Address for the DMA Transmit Channel0. See Table 139 on page 474 for similar registers.

Address 0xBD808240

Reset Value 0x00

Access Read/Write

Bit Position	31 ... 4	3 ... 0
Field Name	ADDR	Rsvd
Reset Value	0000_000	raz

Field Name	Bit Position	Description
ADDR	31:4	**Target Address** — Provides a 16Byte aligned PCI address to start an outbound PCI write of data coming from Transmit Channel0.
Reserved	3:0	Read as zero.

Table 139 DMA Transmit Channel1 PCI Target Register (for Channel1)

Register Name	Purpose	Address
DMA Transmit Channel1 PCI Target	Same as *DMA Transmit Channel0 PCI Target*, except for channel1.	0xBD808244

DMA Receive Channel0 PCI Target Register (XP Configuration Function)

Purpose Provides the PCI Target Address for DMA Receive Channel0. See Table 140 on page 474 for similar registers.

Address 0xBD808248

Reset Value 0x00

Access Read/Write

Bit Position	31	4	3	0
Field Name	ADDR		Rsvd	
Reset Value	0000_000		raz	

Field Name	Bit Position	Description
ADDR	31:4	**Target Address** — Provides a 16Byte aligned PCI address to start an outbound PCI read of data going to Receive Channel 0.
Reserved	3:0	Read as zero.

Table 140 DMA Receive Channel1 PCI Target Register (for Channel1)

Register Name	Purpose	Address
DMA Receive Channel1 PCI Target	Same as *DMA Receive Channel0 PCI Target*, except for channel1.	0xBD808250

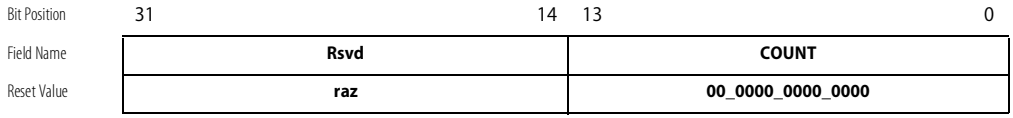
DMA Receive Channel0 Transfer Count Register (XP Configuration Function)

Purpose Provides the transfer count for DMA Receive Channel0. See Table 141 on page 475 for similar register.

Address 0xBD80824C

Reset Value 0x0000

Access Read/Write



Bit Position	31 … 14	13 … 0
Field Name	Rsvd	COUNT
Reset Value	raz	00_0000_0000_0000

Field Name	Bit Position	Description
Reserved	31:14	Read as zero.
COUNT	13:0	**Transfer Count** — Specifies the number of 4Byte transfers to be initiated on the PCI Bus when retrieving data for DMA Receive Channel 0. The transfer count legal range is 1 to 16k. The value of 16k is denoted by a programmed value of 0.

Table 141 DMA Receive Channel1 Transfer Count Register (for Channel1)

Register Name	Purpose	Address
DMA Receive Channel1 Transfer Count	Same as *DMA Receive Channel0 Transfer Count*, except for channel1.	0xBD808254

XP Miscellaneous Control Register (XP Configuration Function)

Purpose Provides control for the PROM and PCI interrupt line.

Address 0xBD808258

Reset Value 0x00

Access Read/Write

Bit Position	31 24	23	22	21	20	19 10	9	8 3	2 0
Field Name	Rsvd	PCI_IMSK3	PCI_IMSK2	PCI_IMSK1	PCI_IMSK0	Rsvd	ZBFP	Rsvd	PROMCLK
Reset Value	0	0	0	0	0	0	0	raz	000

Field Name	Bit Position	Description
Reserved	31:24	Read as zero.
PCI_IMSK3	23	**PCI Interrupt Mask 3** — When the bit is 1, any interrupt intended for the XP on IRQ3 is redirected to the PCI Interrupt Line. When the bit is 0, the interrupt is directed to the XP.
PCI_IMSK2	22	**PCI Interrupt Mask 2** — When the bit is 1, any interrupt intended for the XP on IRQ2 is redirected to the PCI Interrupt Line. When the bit is 0, the interrupt is directed to the XP.
PCI_IMSK1	21	**PCI Interrupt Mask 1** — When the bit is 1, any interrupt intended for the XP on IRQ1 is redirected to the PCI Interrupt Line. When the bit is 0, the interrupt is directed to the XP.
PCI_IMSK0	20	**PCI Interrupt Mask 0** — When the bit is 1, any interrupt intended for the XP on IRQ0 is redirected to the PCI Interrupt Line. When the bit is 0, the interrupt is directed to the XP.
Reserved	19:10	Read as zero.
ZBFP	9	**Payload Bus Arbiter FP More Slots** — When the bit is 1, the FP is given more slots on the Payload Bus.
Reserved	8:3	Read as zero.
PROMCLK	2:0	**PROM Clock Divider** — Specifies the clock divider applied to the core clock to generate the PROM Interface serial clock. The default is zero which, sets the clock divider to 16. All other values result in a clock divider that is 2 times the value.

XP Auxiliary Event Register (XP Configuration Function)

Purpose Contains flags for CP Interrupts and PCI Mailbox Interrupts.

Address 0xBD80825C

Reset Value 0x00

Access Read/Write, write 1 clear

Bit Position	31	30	29	28	27	26	25	24	23 19	18	17	16	15 0
Field Name	R7	R6	R5	R4	R3	R2	R1	R0	Rsvd	PCI_INT	FPTX_INT	FPRX_INT	CP_INT
Reset Value	0	0	0	0	0	0	0	0	raz	0	0	0	0000

Field Name	Bit Position	Description
R7	31	**Mailbox Register 7 Status Bit** — This bit is set by hardware when an inbound PCI transaction writes to *Inbound PCI Mailbox Register 7*, and is reset by hardware when the XP reads from *Inbound Mailbox Register 7*. This bit can also be reset by software by writing a 1 to the bit position.
R6	30	**Mailbox Register 6 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 6*.
R5	29	**Mailbox Register 5 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 5*.
R4	28	**Mailbox Register 4 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 4*.
R3	27	**Mailbox Register 3 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 3*.
R2	26	**Mailbox Register 2 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 2*.
R1	25	**Mailbox Register 1 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 1*.
R0	24	**Mailbox Register 0 Status Bit** — Same as R7 but pertains to *Inbound PCI Mailbox Register 0*.
Reserved	23:19	Read as zero.
PCI_INT	18	**PCI Interrupt** — Indicates that a PCI interrupt is active. (A PCI interrupt request can be made by setting a bit in the *XP To CP Interrupt Request* Register.)
FPtx_INT	17	**TxFP Interrupt Status Bits** — When read as 1, indicates that an interrupt request was made by the TxFP unit.

Field Name	Bit Position	Description
FPrx_INT	16	**RxFP Interrupt Status Bits** — When read as 1, indicates that an interrupt request was made by the RxFP unit.
CP_INT	15:0	**CP Interrupt Status Bits** — When read as 1, indicates that an interrupt request was made by the CP corresponding to the bit number.

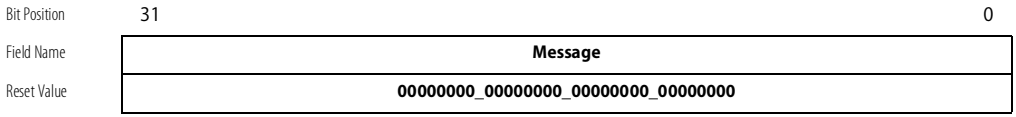
Inbound PCI Mailbox0 Register (XP Configuration Function)

Purpose Inbound PCI mailbox registers. See Table 142 on page 478 for similar registers.

Address 0xBD808260

Reset Value 0x0000

Access Read/Write



Field Name	Bit Position	Description
Message	31:0	**Mailbox Message**

Table 142 Inbound PCI Mailbox*n* Registers (for Mailbox 1, 2, 3, 4, 5, 6 and 7)

Register Name	Purpose	Address
Inbound PCI Mailbox1	Same as *Inbound PCI Mailbox0 Register*, except for mailbox 1.	0xBD808264
Inbound PCI Mailbox2	Same as *Inbound PCI Mailbox0 Register*, except for mailbox 2.	0xBD808268
Inbound PCI Mailbox3	Same as *Inbound PCI Mailbox0 Register*, except for mailbox 3.	0xBD80826C
Inbound PCI Mailbox4	Same as *Inbound PCI Mailbox0 Register*, except for mailbox 4.	0xBD808270
Inbound PCI Mailbox5	Same as *Inbound PCI Mailbox0 Register*, except for mailbox 5.	0xBD808274
Inbound PCI Mailbox6	Same as *Inbound PCI Mailbox0 Register*, except for mailbox6.	0xBD808278

Table 142 Inbound PCI Mailbox*n* Registers (for Mailbox 1, 2, 3, 4, 5, 6 and 7) (continued)

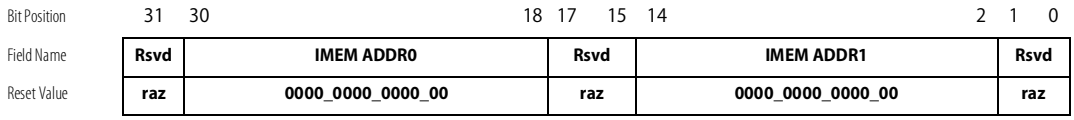
Register Name	Purpose	Address
Inbound PCI Mailbox7	Same as *Inbound PCI Mailbox0 Register*, except for mailbox7.	0xBD80827C

IMEM Overlay Target Address Register (XP Configuration Function)

Purpose IMEM Overlay Target Address Register.

Address 0xBD808280

Access Read/Write



Field Name	Bit Position	Description
Reserved	31	Read as zero.
IMEM ADDR0	30:18	**IMEM ADDR0** – This is the target address into the IMEM that is used during transfers driven by DataScope0 of the TxCB#25.
Reserved	17:15	Read as zero.
IMEM ADDR1	14:2	**IMEM ADDR1** – This is the target address into the IMEM that is used during transfers driven by DataScope1 of the TxCB#25.
Reserved	1:0	Read as zero.

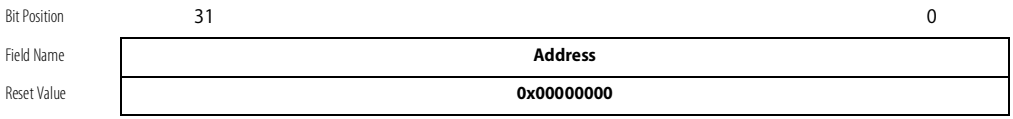
RxCB #25 Transfer Count Register (XP Configuration Function)

Purpose RxCB #25 Transfer Count Register.

Address 0xBD808284

Reset Value 0x0

Access Read/Write



XP Diagnostic Register (XP Configuration Function)

Purpose Retains data through warm reset. Used for diagnostic purposes.

Address 0xBD808288

Reset Value 0x0 (hard reset only; retains data on warm reset)

Access Read/Write

Bit Position	31	0
Field Name	**Address**	
Reset Value	**0x00000000**	

PCI Outbound Byte Swap Control Register (XP Configuration Function)

Purpose Provides control of the byte swapping feature for outbound transactions.

Address 0xBD80828C

Reset Value 0x0000

Access Read/Write

Bit Position	31 12	11	10	9	8	7	6	5	4	3	2	1	0
Field Name	**Rsvd**	**TX1**	**TX0**	**RX1**	**RX0**	**BAR7**	**BAR6**	**BAR4**	**BAR4**	**BAR3**	**BAR2**	**BAR1**	**BAR0**
Reset Value	**0x00000**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Field Name	Bit Position	Description
Reserved	31:12	Read as zero.
TX1	11	**TX1 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction associated with *DMA Transmit Channel 1* register.
TX0	10	**TX0 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction associated with *DMA Transmit Channel 0* register.
RX1	9	**RX1 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction associated with *DMA Receive Channel 1* register.
RX0	8	**RX0 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction associated with *DMA Receive Channel 0* register.

Field Name	Bit Position	Description
BAR7	7	**BAR7 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 7* register.
BAR6	6	**BAR6 Byte Swap Enable** — When set to1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 6* register.
BAR5	5	**BAR5 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 5* register.
BAR4	4	**BAR4 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 4* register.
BAR3	3	**BAR3 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 3* register.
BAR2	2	**BAR2 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 2* register.
BAR1	1	**BAR1 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 1* register.
BAR0	0	**BAR0 Byte Swap Enable** — When set to 1, byte swapping occurs as data passes through the PCI interface for any transaction decoded by *PCI Outbound Base Address 0* register.

Debug Counter0 Start Value Register (XP Configuration Function)

Purpose — Provides the start value for the associated 32bit debug event counter. See Table 143 on page 482 for similar registers.

Note: That following a RESET, this register contains the boot address read by the XP's IROM.

Address — 0xBD808300

Reset Value — 0xBFC00000

Access — Read/Write

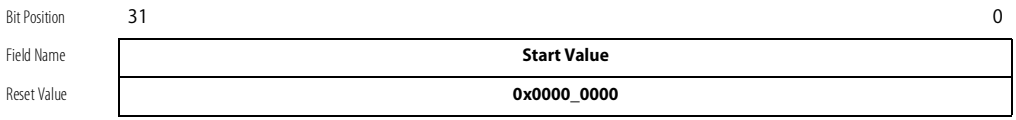


Table 143 Debug Counter*n* Start Value Registers (for Debug Counter 1, 2 and 3)

Register Name	Purpose	Address	Reset Value
Debug Counter1 Start Value	Same as *DebugCounter0 Start Value Register*, except for debug counter 1.	0xBD808304	0x00000000
Debug Counter2 Start Value	Same as *DebugCounter0 Start Value Register*, except for debug counter 2.	0xBD808308	
Debug Counter3 Start Value	Same as *DebugCounter0 Start Value Register*, except for debug counter 3.	0xBD80830C	

Debug Counter0 Control Register (XP Configuration Function)

Purpose Provides control for the associated debug counter register. See Table 144 on page 484 for similar registers.

Address 0xBD808340

Reset Value 0x00008888

Access Read/Write

Bit Position	31	30 20	19	18	17	16	15 12	11 8	7 4	3 0
Field Name	DEBUG_EN	Rsvd	INC_ED	LD_ED	STRT_ED	STP_ED	INCSEL	LDSEL	STRTSEL	STPSEL
Reset Value		raz	0	0	0	0	1000	1000	1000	1000

Field Name	Bit Position	Description
DEBUG_EN	31	**Debug Enable** — When the bit is 1, it enables the debug counters. When the bit is 0, it disables the debug counters.
Reserved	30:20	Read as zero.
INC_ED	19	**Increment Control Edge Detect Enable** — When the bit is 1, a rising edge detect is performed on the signal selected as the increment signal for the debug counter, providing a one core clock long pulse for each rising edge appearing on the input signal. When the bit is 0, the raw signal is used as the increment signal.
LD_ED	18	**Load Control Edge Detect Enable** — When the bit is 1, a rising edge detect is performed on the signal selected as the load signal for the debug counter, providing a one core clock long pulse for each rising edge appearing on the input signal. When the bit is 0, the raw signal is used as the load signal.
STRT_ED	17	**Start Control Edge Detect Enable** — When the bit is 1, a rising edge detect is performed on the signal selected as the start signal for the debug counter, providing a one core clock long pulse for each rising edge appearing on the input signal. When the bit is 0, the raw signal is used as the start signal.
STP_ED	16	**Stop Control Edge Detect Enable** — When the bit is 1, a rising edge detect is performed on the signal selected as the stop signal for the debug counter, providing a one core clock long pulse for each rising edge appearing on the input signal. When the bit is 0, the raw signal is used as the stop signal.

Field Name	Bit Position	Description
INCSEL	15:12	**Increment Control Select** — Selects the signal to use to control the increment line for the debug counter as listed below. **Encoded Value** / **Control Value** 0000 — Debug bus 0 0001 — Debug bus 1 0010 — Debug bus 2 0011 — Debug bus 3 0100 — Counter0 overflow 0101 — Counter1 overflow 0110 — Counter2 overflow 0111 — Counter3 overflow 1000 — 0 1001 — 1 1010 -1111 — Reserved
LDSEL	11:8	**Load Control Select** — Selects the signal to use to control the load line for the debug counter. The selection values are the same as shown above for INCSEL.
STRTSEL	7v4	**Start Control Select** — Selects the signal to use to enable the debug counter for counting. The selection values are the same as shown above for INCSEL.
STPSEL	3:0	**Stop Control Select** — Selects the signal to use to disable the debug counter for counting. The selection values are the same as shown above for INCSEL.

Table 144 Debug Counter*n* Control Registers (for Debug Counter 1, 2 and 3)

Register Name	Purpose	Address
Debug Counter1 Control	Same as *Debug Counter0 Control Register*, except for debug counter 1.	0xBD808344
Debug Counter2 Control	Same as *Debug Counter0 Control Register*, except for debug counter 2.	0xBD808348
Debug Counter3 Control	Same as *Debug Counter0 Control Register*, except for debug counter 3.	0xBD80834C

Debug Counter0 Current Value Register (XP Configuration Function)

Purpose Provides access to the current value of the associated 32bit debug event counter. See Table 145 on page 485 for similar registers.

Address 0xBD808380

Reset Value 0x00

Access Read only

Bit Position	31	0
Field Name	**Current Value**	
Reset Value	**0x0000_0000**	

Table 145 Debug Counter*n* Current Value Registers (for Debug Counter 1, 2 and 3)

Register Name	Purpose	Address
Debug Counter1 Current Value	Same as *Debug Counter0 Current Value Register*, except for counter 1.	0xBD808384
Debug Counter2 Current Value	Same as *Debug Counter0 Current Value Register*, except for counter 2.	0xBD808388
Debug Counter3 Current Value	Same as *Debug Counter0 Current Value Register*, except for counter 3.	0xBD80838C

All of the registers that pertain to XP DMEM #24 (0xBD804000 to 0xBD80443C) are identical to their counterparts in the Channel Processors (CP) except for those (6) registers documented here. These same (6) registers are also found in the (CP), however, the registers provide different functions for the XP versus the CP. By changing the use of the individual bits inside these registers they are capability of providing the different functions needed in the XP and CPs.

RxCtl0_Status Register (XP DMEM#24 Transfer Rx Control Block0 Function)

Purpose Semaphores governing PCI DMA receive operation for datascope0. See Table 146 on page 486 for similar register.

Address 0xBD804090

Reset Value 0x80000000

Access XP Read/Write

Bit Position	31	30 24	23 0
Field Name	**Avail**	**Reserved**	**Reserved**
Reset Value	**1**	**raz**	**raz**

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, the XP owns receive datascope 0. When the bit is 0, PCI DMA owns receive datascope0.
Reserved	30:24	Read as zero.
Reserved	23:0	Read as zero.

Table 146 RxCtl1_Status Register (for Datascope1)

Register Name	Purpose	Address
RxCtl1_Status	Same as RxCtl0_Status, but pertains to datascope1.	0xBD804290

TxCB0_Ctl Register (XP DMEM#24 Transfer Tx Control Block0 Function)

Purpose Controls DMA for payload transmit operation for datascope0. See Table 147 on page 487 for similar register.

Address 0xBD804184

Reset Value 1x0xxxxxxx000x00xxxxxxxxxxxxxxxxb

Access CPRC Read/Write. Usage is the same as for the CPs with the exception that when using *TxCB0_CTL* to perform DMAs on the PCI bus, the transfer length (*TxLength*) must be a multiple of four bytes.

Table 147 TxCB1_CTL Register

Register Name	Purpose	Address	Access
TxCB1_CTL	Same as TxCB0_CTL, but pertains to transmit datascope1.	0xBD804384	Same as TxCB0_CTL

TxCtl0_Status Register (XP DMEM#24 Transfer Tx Control Block0 Function)

Purpose Semaphores governing PCI DMA transmit operation for datascope0.
See Table 148 on page 487 for similar register.

Address 0xBD804190

Reset Value 0x80000000

Access XP Read/Write

Bit Position	31	30 24	23 0
Field Name	Avail	Reserved	Reserved
Reset Value	1	raz	raz

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, the XP owns transmit datascope 0. When the bit is 0, PCI DMA owns transmit datascope 0.
Reserved	30:24	Read as zero.
Reserved	23:0	Read as zero.

Table 148 TxCtl1_Status Register (for Datascope1)

Register Name	Purposes	Address
TxCtl1_ Status	Same as TxCtl0_Status, but pertains to transmit datascope1.	0xBD804390

All of the registers that pertain to XP Mode, Queue Status and Event (0xBD804500 to 0xBD8046C0) are identical to their counterparts in the Channel Processors (CP) except for those (4) registers documented here. These same (4) registers are also found in the (CP), however, the registers provide different functions for the XP versus the CP. By changing the use of the individual bits inside these registers they are capability of providing the different functions needed in the XP and CPs.

XP_Mode Register (XP Mode Configuration Function)

Purpose Collects mode and error status bits relevant to general XP configuration.

Address 0xBD804640

Access Upper 16 bits are XP Read/Write, Lower 16 bits are Write 1 to clear, hardware update, except for *QMU rdmbx* and *QMU wrmbx* which are read only.

Bit Position	31	30	29 17	23 22	21 20	19 17	16	15 13	12	11	10 8	7	6 4	3	2 0
Field Name	XP Reset	Wind Down	Rsvd	QMU rdmbx	QMU wrmbx	Rsvd	Retry Global	Rsvd	NXM	PErr #24	PErr Status #24	PErr #25	PErr Status #25	GErr	GErr Status
Reset Value	1	raz	raz	0	0	raz	0	raz		0	000	0	000	0	000

Field Name	Bit Position	Description
XP Reset	31	**XP Reset** — When the bit is 0, the XP is held in reset state. To bring the XP out of reset, this bit must be set to 1 with a PCI inbound transaction.
WindDown	30	**WindDown** — When the bit is 1, this bit asserts a global signal informing all chip functions to wind down as soon as possible, and leave as much predictable error recovery state around as possible. This bit is readable, allowing a process to determine that the global signal was caused by a process setting this bit, however, the write causes only a single global wind down request.
Reserved	29:24	Read as zero.
QMU rdmbx	23:22	**QMU Read Mailbox Status** (read only): **Encoded Value** / **Status** 00 — QMU idle or operation finished successfully 01 — operation finished with error (probably resource error, see above 10 — busy, waiting to begin execution 11 — busy, executing in QMU engine

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>QMU wrmbx</td><td>21:20</td><td>QMU Write Mailbox Status (read only):

<table><tr><th>Encoded Value</th><th>Status</th></tr><tr><td>00</td><td>QMU idle or operation finished successfully</td></tr><tr><td>01</td><td>operation finished with error (probably resource error, see above</td></tr><tr><td>10</td><td>busy, waiting to begin execution</td></tr><tr><td>11</td><td>busy, executing in QMU engine</td></tr></table></td></tr>
<tr><td>Reserved</td><td>19:17</td><td>Read as zero.</td></tr>
<tr><td>RetryGlobal</td><td>16</td><td>Global Bus Transaction Retry — This bit causes global load and store operations through the Global Bus controller to be retried up to 16 times when NACK'd. When 16 tries have been NACK'd, the bus controller terminates the operation and asserts a bus error.</td></tr>
<tr><td>Reserved</td><td>15:13</td><td>Read as zero.</td></tr>
<tr><td>NXM</td><td>12</td><td>Non-Existent Memory — Indicates that an access has occurred to a non-existent memory location (NXM). This bit is write 1 to clear.</td></tr>
<tr><td>PErr #24</td><td>11</td><td>Payload #24 Error — An error was detected on a payload bus read or write on Payload Bus Node #24.</td></tr>
<tr><td>PErr Status #24</td><td>10:8</td><td>Payload #24 Error Status — Loaded when a Payload Error occurs in Payload Bus Node #24, and is locked until the XPRC clears the Payload #24 Error bit. The individual control blocks can be interrogated to determine the specific offender.</td></tr>
<tr><td>PErr #25</td><td>7</td><td>Payload #25 Error — An error was detected on a payload bus read or write on Payload Bus Node #25.</td></tr>
<tr><td>PErr Status #25</td><td>6:4</td><td>Payload #25 Error Status — Loaded when a Payload Error occurs in Payload Bus Node #25, and locked until the XPRC clears the Payload #25 Error bit. The individual control blocks can be interrogated to determine the specific offender.</td></tr>
<tr><td>GErr</td><td>3</td><td>Global Bus Controller Error — An error was detected on a Global read or write attempted by the XPRC.</td></tr>
<tr><td>GErr Status</td><td>2:0</td><td>Global Bus Error Status — Loaded when a global error occurs, and is locked until the XPRC process clears the global error bit. Status codes are identical to those for the CPs.</td></tr>
</table>

XP Debug Mode Register (XP Mode Configuration Function)

Purpose Configures the XP debug tap for the global debug counters.

Address 0xBD804658

Access XP Read/Write

Bit Position	31	30 28	27 24	23	22 20	19 16	15	14 12	11 8	7 6	4 3 0
Field Name	Enb0	Rsvd	MUX0	Enb1	Rsvd	MUX1	Enb2	Rsvd	MUX2	Enb3	Rsvd	MUX 3
Reset Value	0	raz	x	0	raz	x	0	raz	x	0	raz	x

Field Name	Bit Position	Description
Enb0	31	**Enable Driver 0** — Enable the driver onto global debug wire 0.
Reserved	30:28	Read as zero.
MUX0	27:24	**Mux 0** — Select 1 of 16 debug events onto global debug wire 0.
Enb1	23	**Enable Driver 1** — Enable the driver onto global debug wire 1.
reserved	22:20	Read as zero.
MUX1	19:16	**Mux 1** — Select 1 of 16 debug events onto global debug wire 1.
Enb2	15	**Enable Driver 2** — Enable the driver onto global debug wire 2.
reserved	14:12	Read as zero.
MUX2	11:8	**Mux 2** — Select 1 of 16 debug events onto global debug wire 2.
Enb3	7	**Enable Driver 3** — Enable the driver onto global debug wire 3.
reserved	6:4	Read as zero.
MUX3	3:0	**Mux 3** — Select 1 of 16 debug events onto global debug wire 3.

There are four (4) global debug wires that carry inputs to the global debug counter block. Each CP and XP have a multiplexor that can select one of the 16 events. The selectable events in the XP to drive on each of the respective debug wires are enumerated in Table 149. Each multiplexor has a 4bit register to select what event to drive, and an enable bit to turn on the debug wire driver. Since chip-wide only one debug wire driver should be enabled at any time, the recommended procedure to use the debug taps is:

1 Clear the master debug enable bit in the global debug configuration register space in the XP.

2 Clear the set driver enable bits. It may be safest to invoke a routine that clears all driver enable bits on every change regardless of the previous configuration.

3 Set one chip-wide driver enable bit and its corresponding multiplexor select value for each global debug wire.

4 Set up the global debug configuration bits and master debug enable.

Table 149 XP Debug Multiplexor Select Encodings

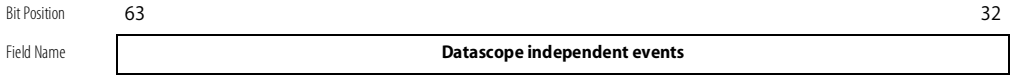
MUX Input	Encoding	Description
0	16:13	Select 0 for multiplexor
ISM_TRAN	12	PCI Master Transaction Initiated
ISM_WRXFER	11	PCI Master has completed a write data phase
ISM_RDXFER	10	PCI Master has completed a read data phase
ISM_DISC	9	PCI Master transaction has been disconnected by addressed target
TSM_TRAN	8	PCI Target has decoded a new inbound transaction
TSM_WRXFER	7	PCI Target has completed a write data phase
TSM_RDXFER	6	PCI Target has completed a read data phase
TSM_LATTO	5	PCI Target has disconnected due to a data latency time-out
Bubble	4	XPRC has inserted a bubble into its pipeline
Stall	3	XPRC data read or write stall cycle
RC Read	2	XPRC data Read
RC Write	1	XPRC data Write
Debug/Match	0	The XPRC data, data address, or instruction address matches the programmed match registers in the XPRC

Event0 Register (Event and Interrupt Control Function)

Purpose Collects together event bits relevant to datascope independent tasks.

Address 0xBD8046A0

Access XPRC Read/Write, write 1 to clear.



Field Name	Bit Position	Description
WindDown	63	**Wind Down** — When unmasked, this global input is a request to wind down all CP activity as soon as possible, and leave as much predictable error recovery state around as possible.
GlobalError	62	**CPRC Global Reference Error** - when asserted, this bit means a CPRC Write received an error on the Global Bus or a non-existent memory error within the cluster.
MCError	61	**Memory Controller Request Error** — Indicates an unrecoverable error occurred during a request sent to the BMU. An error status code is stored in the *xp_mode_register*, and also in the control block that initiated the request.
QMUError	60	**QMU Error** — Indicates an unrecoverable error occurred during a request sent to the QMU. An error status code is stored in the *xp_mode_register*.
CP_Interrupt	59	**CP Interrupt Request** — One of the CPs issued an interrupt request to the XP.
PayloadAlert	58	**Payload Request Alert** — A non-fatal bus error has occurred while trying to send a request to the BMU or QMU.
DebugMatch	57	**XPRC Debug Match** — The XPRC data, data address, and instruction address matched the programmed match registers in the CPRC.
TLUError	56	**TLU Error** — Indicates that an unrecoverable error occurred during a request sent to the TLU.
SB_TransDone	55	**Serial Bus Transfer Done** — Indicates that a read or write transfer completed on the serial bus.
Reserved	54	Read as zero.
RxMsgFIFO	53	**Ring Bus Receive Message Available** — This bit indicates the availability of a Ring Bus message in the receive FIFO, and corresponds to *RxMsgCtl.State* [31].
TimerEvent	52	**Event Timer Time-out** — This bit indicates the event timer counted down to 0.

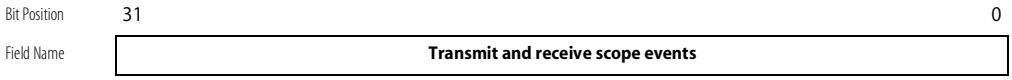
Field Name	Bit Position	Description
PCI Mailbox	51	**PCI Mailbox Interrupt** — One of the PCI mailbox registers has been written to by an inbound PCI transaction and contains a pending message.
Reserved	50	Read as zero.
Reserved	49:48	Software controlled.
RxResp7-0	47:40	**Ring Bus Receive Response Available** — These eight bits correspond to the available bit for the eight Ring Bus receive response available bits. Bit 47 represents RxResp7Ctl.Avail, and bit 40 represents RxRespCtl0.Ctl.
TxMsg3-0	39:36	**Ring Bus Transmit Message Available** — These four bits correspond to the available bit for the four Ring Bus transmit message control registers. Bit 39 represents TxMsg3Ctl.Avail, and bit 36 represents *TxMsg0Ctl.Avail*.
WrCB	35:34	**Write Control Blocks 0/1** — These two bits correspond to the available bit for the two payload bus write control blocks. Bit 35 corresponds to *WrCB1Ctl.Avail*, and bit 34 corresponds to *WrCB0Ctl.Avail*.
RdCB	33:32	**Read Control Blocks 0/1** — These two bits correspond to the available bit for the two payload bus read control blocks. Bit 33 corresponds to *RdCB1Ctl.Avail*, and bit 32 corresponds to *RdCB0Ctl.Avail*.

Event1 Register (Event and Interrupt Control Function)

Purpose Collects together event bits relevant to transmit and receive datascopes.

Address 0xBD8046A4

Access CPRC Read/Write, write 1 to clear



Field Name	Bit Position	Description
QRdMbxAvail	31	**Queue Read Mailbox Available** — This bit indicates that this CP's read mailbox in the QMU went from busy to available.
TxCB1_Avail (DMEM #24)	30	**Transmit Control Block Available** - Indicates that the available bit for datascope1 Payload Bus transmit control block #24 (TxCB1Ctl.Avail) was set.

Field Name	Bit Position	Description
TxStatus1_Avail (DMEM #24)	29	**PCI Transmit Datascope1 Available** - Indicates that the PCI transmit state machine has set TxStatus1.Avail.
TxCB1_Avail (DMEM #25)	28	**Transmit Control Block Available** - Indicates that the available bit for datascope1 Payload Bus transmit control block #25 (TxCB1Ctl.Avail) was set.
TxStatus1_Avail (DMEM #25)	27	**IMEM Loader Datascope1 Available** — Indicates that the IMEM Loader state machine has set TxStatus1_Avail.
QRdMbxBusy	26	**Queue Read Mailbox Busy** — This bit indicates that this CP's read mailbox in the QMU went from available to busy.
TxCB0_Avail (DMEM #24)	25	**Transmit Control Block Available** — Indicates that the available bit for datascope0 payload bus transmit control block #24 (TxCB0Ctl.Avail) was set.
TxStatus0_Avail (DMEM #24)	24	**PCI Transmit datascope0 Available** — Indicates that the PCI transmit state machine has set TxStatus0.Avail.
TxCB0_Avail (DMEM #25)	23	**Transmit Control Block Available** — Indicates that the available bit for datascope0 payload bus transmit control block #25 (TxCB0Ctl.Avail) was set.
TxStatus0_Avail (DMEM #25)	22	**IMEM Loader Datascope0 Available** — Indicates that the IMEM Loader state machine has set TxStatus0_Avail.
QueueStatus	21	**Queue Status** — This bit corresponds to the logical OR of the bits within each of the four *Queue_Status* registers. The bit is level sensitive.
Reserved	20	Read as zero.
WrCB1-0 (DMEM #25)	19:18	**Write Control Block Available** — These two bits correspond to the available bit for the two payload bus write control blocks associated with DMEM #25. Bit 19 corresponds to WrCB1Ctl, and bit 18 corresponds to WrCB0Ctl.Avail.
RdCB1-0 (DMEM #25)	17:16	**Read Control Block Available** — These two bits correspond to the available bit for the two payload bus read control blocks associated with DMEM #25. Bit 19 corresponds to RdCB1Ctl, and bit 18 corresponds to RdCB0Ctl.Avail.
QWrMbxAvail	15	**Queue Write Mailbox Available** — This bit indicates that this CP's write mailbox in the QMU went from busy to available.
RdCB1_Avail (DMEM #24)	14	**Receive Control Block Available** — Indicates that the available bit for datascope1 payload bus receive control block #24 (RxCB1_Ctl.Avail) was set.
RxStatus1_Avail (DMEM #24)	13	**PCI Receive Datascope1 Available** — Indicates that the PCI receive state machine has set RxStatus1.Avail.

Field Name	Bit Position	Description
RdCB1_Avail (DMEM #25)	12	**Receive Control Block Available** — Indicates that the available bit for datascope1 payload bus receive control block #25 (RxCB1_Ctl.Avail) was set.
RxStatus1_Avail (DMEM #25)	11	**#25 Receive Datascope1 Available** — Indicates that the #25 receive state machine has set RxStatus1.Avail.
QWrMbxBusy	10	**Queue Read Mailbox Busy** — This bit indicates that this CP's write mailbox in the QMU went from available to busy.
RxCB0_Avail (DMEM #24)	9	**Receive Control Block Available** — Indicates that the available bit for datascope0 payload bus receive control block #24 (RxCB0_Ctl.Avail) was set.
RxStatus0_Avail (DMEM #24	8	**PCI Receive Datascope0 Available** — Indicates that the PCI receive state machine has set RxStatus0.Avail.
RxCB0_Avail (DMEM #25)	7	**Receive Control Block Available** — Indicates that the available bit for datascope0 payload bus receive control block #25 (RxCB0_Ctl.Avail) was set.
RxStatus0_Avail (DMEM #25	6	**#25 Receive Datascope0 Available** — Indicates that the #25 receive state machine has set RxStatus0.Avail.
Ext_Interrupt	5	**External PHY Interrupt** — A PHY generated external interrupt that comes in through the XPU_HOT pin when not in reset, causes an event in this bit.
PCI Master Abort	4	**PCI Master Transaction Aborted** — The PCI Master has experienced either a Master Abort, a Target Abort, or an abort caused by the PCI Master being disabled.
Debug Events	3:0	**Debug Events** — Bits 3:0 correspond to debug counters 3:0, and are set whenever a debug counter triggers its event signal.

All of the registers that pertain to XP DMEM #25 (0xBD804800 to 0xBD804C3C) are identical to their counterparts in the Channel Processors (CP) except for those (4) registers documented here. These same (4) registers are also found in the (CP), however, the registers provide different functions for the XP versus the CP. By changing the use of the individual bits inside these registers they are capability of providing the different functions needed in the XP and CPs.

RxCtl0_Status Register (XP DMEM#25 Transfer Control Block0 Function)

Purpose Semaphores governing PCI DMA receive operation for datascope0. See Table 150 on page 496 for similar register.

Address 0xBD804890

Reset Value 0x80000000

Access XP Read/Write

Bit Position	31	30 24	23 0
Field Name	**Avail**	**Reserved**	**Reserved**
Reset Value	**1**	**raz**	**raz**

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, the XP owns receive datascope0. When the bit is 0, PCI DMA owns receive datascope0.
Reserved	30:24	Read as zero.
Reserved	23:0	Read as zero.

Table 150 RxCtl1_Status Register

Register Name	Purpose	Address
RxCtl1_Status	Same as *RxCtl0_Status*, but for datascope1.	0xBD80A290

TxCtl0_Status Register (XP DMEM#25 Transfer Control Block0 Function)

Purpose Semaphores governing PCI DMA transmit operation for datascope0. See Table 151 on page 497 for similar register.

Address 0xBD804990

Reset Value 0x80000000

Access XP Read/Write

Bit Position	31	30 24	23 0
Field Name	Avail	Reserved	Reserved
Reset Value	1	raz	raz

Field Name	Bit Position	Description
Avail	31	**Availability Bit** — When the bit is 1, the XP owns transmit datascope0. When the bit is 0, PCI DMA owns transmit datascope0.
Reserved	30:24	Read as zero.
Reserved	23:0	Read as zero.

Table 151 TxCtl1_Status Register

Register Name	Purpose	Address
TxCtl1_Status	Same as *TxCtl0_Status*, but pertains to transmit datascope1.	0xBD804B90

Queue Management Unit (QMU) Configuration Registers

Configuration Space in the QMU is an area that contains a number of registers. The QMU occupies 1MByte within the C-5 NP's Configuration Space starting at 0xBDA00000 to 0xBDAFFFFF. The QMU only supports 32bit aligned operations. The QMU uses these registers to: map queues to CPs, XP and FP, configure the QMU, debug the QMU, and collect QMU statistics. Processor registers (WrCB0, and RdCB0) are described in Chapter 2 to move data through the internal QMU and external QMU to/from SRAM from/to the DMEM of either the requesting CPs, or XP.

These registers (WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr and RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr) are physically located in the Configuration Space of their respective CPs, or XP and not in the BMU Configuration Space.

WARNING: *When the QMU is run-enabled, an attempt to read or write many of the internal registers and memories will interfere with the operation of the QMU.*

QMU Registers

The following is a list of each QMU register along with its address, function, and reference to its detailed parameters. The detailed parameters provide: purpose, field name, bit position, and descriptions.

Table 152 QMU Registers

Address	Register Name	Function	Detailed Parameters
0xBDA00000	QMU_Run_Enable	Enables QMU	See page 501
0xBDA00040	Base_Queue_CP0	CP's Queue Allocation	See page 501
0xBDA00044	Base_Queue_CP1		
0xBDA00048	Base_Queue_CP2		
0xBDA0004C	Base_Queue_CP3		
0xBDA00050	Base_Queue_CP4		
0xBDA00054	Base_Queue_CP5		
0xBDA00058	Base_Queue_CP6		
0xBDA0005C	Base_Queue_CP7		
0xBDA00060	Base_Queue_CP8		
0xBDA00064	Base_Queue_CP9		
0xBDA00068	Base_Queue_CP10		
0xBDA0006C	Base_Queue_CP11		
0xBDA00070	Base_Queue_CP12		
0XBDA00074	Base_Queue_CP13		
0xBDA00078	Base_Queue_CP14		
0xBDA0007C	Base_Queue_CP15		
0xBDA000C0	Base_Queue_FP	FP's Queue Allocation	See page 502
0xBDA000C8	Base_Queue_XP	XP's Queue Allocation	See page 502

Table 152 QMU Registers (continued) (continued)

Address	Register Name	Function	Detailed Parameters
0xBDA000D4	Num_Descriptors	QMU Configuration	See page 503
0xBDA000DC	Dyn_Des_Usage_Lim_Pool0		See page 503
0xBDA000E0	Dyn_Des_Usage_Lim_Pool1		
0xBDA000E4	Dyn_Des_Usage_Lim_Pool2		
0xBDA000E8	Dyn_Des_Usage_Lim_Pool3		
0xBDA000F0	Operation_Mode		See page 504
0xBDA000F4	Descriptor_Size		See page 504
0xBDA00180	Config_Q_Cnt	QMU Statistics	See page 505
0xBDA00184	Rd_Q_Status_Cnt		See page 505
0xBDA00188	CP_Uni_Enq_Cnt		See page 505
0xBDA0018C	CP_Multi_Enq_Cnt		See page 505
0xBDA00190	CP_Multi_Enq_Target_Cnt		See page 505
0xBDA00194	CP_Dequeue_Cnt		See page 505
0xBDA00198	FP_Uni_Enq_Cnt		See page 505
0xBDA0019C	FP_Multi_Enq_Cnt		See page 505
0xBDA001A0	FP_Multi_Enq_Target_Cnt		See page 506
0xBDA001A4	FP_Dequeue_Cnt		See page 506
0xBDA001A8	QMU_Idle_Cycles		See page 506
0xBDA001AC	Payload_NACK_Cnt		See page 506
0xBDA001B0	Global_NACK_Cnt		See page 506
0xBDA001B4	Payload_Rd_Failures_Cnt		See page 506
0xBDA001B8	Cmd_Processor_Err_Cnt		See page 506
0xBDA001BC	Q_Engine_Err_Cnt		See page 507
0xBDA00400 to 0xBDA0063C	Multicast_Destination0 to Multicast_Destination143	QMU Configuration	See page 507

Table 152 QMU Registers (continued) (continued)

Address	Register Name	Function	Detailed Parameters
0xBDA7E008	Free_Descriptor_Buffer_List	QMU Status	See page 510
0xBDA7E080	Dyn_Descriptor_Pool0_Usage		See page 511
0xBDA7E084	Dyn_Descriptor_Pool1_Usage		See Table 157 on page 511.
0xBDA7E088	Dyn_Descriptor_Pool2_Usage		
0xBDA7E08C	Dyn_Descriptor_Pool3_Usage		

QMU Detailed Descriptions

The following is a detailed description of each of the QMU registers and their individual parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

QMU_Run_Enable Register (QMU Enable Queue Function)

Purpose When this one bit wide register is 1, it enables the QMU's processing of queue operations. When this bit is 0, it disables the QMU's execution of queue operations. The QMU powers up when this bit is clear. This bit must be set to a "1" before the QMU can process any queue operations.

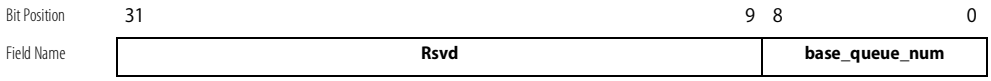
Address 0xBDA00000

Bit Position	31 ... 1	0
Field Name	Rsvd	Enable

Base_Queue_CP0 to Base_Queue_CP15 Registers (QMU CP's Queue Allocation Function)

Purpose These registers specify the base address for a CP's queues.

Addresses 0xBDA00040 (CP 0 base address), 0xBDA00044 (CP 1 base address)
0xBDA00048 (CP 2 base address), 0xBDA0004C (CP 3 base address)
0xBDA00050 (CP 4 base address), 0xBDA00054 (CP 5 base address)
0xBDA00058 (CP 6 base address), 0xBDA0005C (CP 7 base address)
0xBDA00060 (CP 8 base address), 0xBDA00064 (CP 9 base address)
0xBDA00068 (CP 10 base address), 0xBDA0006C (CP 11 base address)
0xBDA00070 (CP 12 base address), 0xBDA00074 (CP 13 base address)
0xBDA00078 (CP 14 base address), 0xBDA0007C (CP 15 base address)

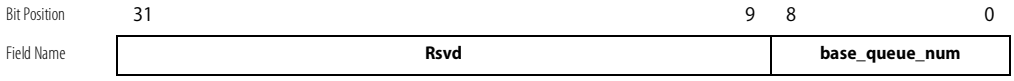


Field Name	Bit Position	Description
Reserved	31:9	Read as zero.
Base_Queue_Num	8:0	Establishes the base queue address for the CP (0 to 15).

Base_Queue_FP Register (QMU FP's Queue Allocation Function)

Purpose This register specifies the base address for the FP's queues.

Address 0xBDA000C0



Field Name	Bit Position	Description
Reserved	31:9	Read as zero.
Base_Queue_Num	8:0	Specifies the base address for the FP's queues

Base_Queue_XP Register (QMU XP's Queue Allocation Function)

Purpose This register specifies the base address for the XP queues. It has the same data format as *Base_Queue_FP* register.

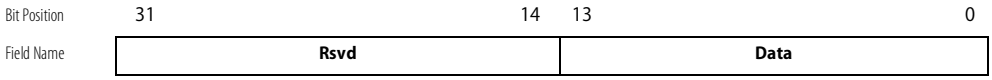
Address 0xBDA000C8

Num_Descriptors Register (QMU Configuration Function)

Purpose This 14 bit wide register specifies the number of descriptor buffers to be available in the QMU. Legal range= 0 to 16,384 as detailed here:

Programmed Value	Number of Descriptors
0 • • • 16,383	1 • • • 16,384

Address 0xBDA000D4



Dyn_Des_Usage_Lim_Pool0 Register (QMU Configuration Function)

Purpose Specify the maximum number of descriptors that can be enqueued dynamically to the queues associated with Pool0. Legal range= 0 to 16K.

The total number of Descriptors allocated among all four (4) pools of the *Dyn_Des_Usage_Lim_Pool0* to *Dyn_Des_Usage_Lim_Pool3* registers should be < the number of dynamically enqueued descriptors. See Table 153 on page 503 for similar registers.

Address 0xBDA000DC

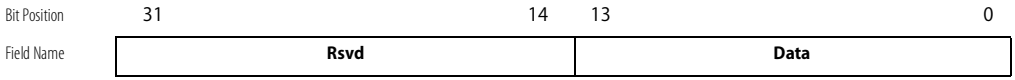


Table 153 Dyn_Des_Usage_Lim_Pool*n* Registers (for Descriptor Pools 1, 2 and 3)

Register Name	Purpose	Address
Dyn_Des_Usage_Lim_Pool1	Same as *Dyn_Des_Usage_Lim_Pool0*, except for descriptor pool1.	0xBDA000E0
Dyn_Des_Usage_Lim_Pool2	Same as *Dyn_Des_Usage_Lim_Pool0*, except for descriptor pool2.	0xBDA000E4

Table 153 Dyn_Des_Usage_Lim_Pool*n* Registers (for Descriptor Pools 1, 2 and 3)

Register Name	Purpose	Address
Dyn_Des_Usage_Lim_Pool3	Same as *Dyn_Des_Usage_Lim_Pool0*, except for descriptor pool2.	0xBDA000E8

Operation_Mode Register (QMU Configuration Function)

Purpose This four bit wide register specifies the operating mode of the QMU. The codes for the modes are listed in Table 154.

Address 0xBDA000F0

Table 154 Queue Operating Mode Codes

Mode	Code
Internal-Queues	0x1
External-Queues	Not implemented

Descriptor_Size Register (QMU Configuration Function)

Purpose This two bit wide register specifies the size of the data stored for each descriptor in an encoded form.

Address 0xBDA000F4

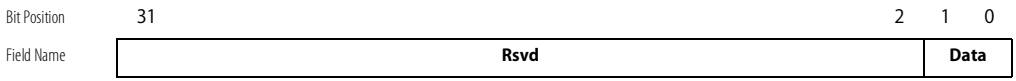


Table 155 Descriptor Size Codes

Descriptor Size	Code
12 Bytes	0
16 Bytes	1
24 Bytes	2
32 Bytes	3

Config_Q_Cnt Register (QMU Statistics Function)

Purpose Count of Queue Configuration operations.

Address 0xBDA00180

Rd_Q_Status_Cnt Register (QMU Statistics Function)

Purpose Count of Read Status operations.

Address 0xBDA00184

CP_Uni_Enq_Cnt Register (QMU Statistics Function)

Purpose Count of Unicast Enqueues from the CPs.

Address 0xBDA00188

CP_Multi_Enq_Cnt Register (QMU Statistics Function)

Purpose Count of Multicast Enqueues from the CPs.

Address 0xBDA0018C

CP_Multi_Enq_Target_Cnt Register (QMU Statistics Function)

Purpose Count of Total Multicast Enqueues Targets from the CPs.

Address 0xBDA00190

CP_Dequeue_Cnt Register (QMU Statistics Function)

Purpose Count of Dequeue operations from the CPs.

Address 0xBDA00194

FP_Uni_Enq_Cnt Register (QMU Statistics Function)

Purpose Count of Unicast Enqueues from the FP.

Address 0xBDA00198

FP_Multi_Enq_Cnt Register (QMU Statistics Function)

Purpose Count of Total Multicast Enqueues from the FP.

Address 0xBDA0019C

FP_Multi_Enq_Target_Cnt Register (QMU Statistics Function)

Purpose Count of Multicast Enqueue Targets from the FP.

Address 0xBDA001A0

FP_Dequeue_Cnt Register (QMU Statistics Function)

Purpose Count of Dequeue operations from the FP.

Address 0xBDA001A4

QMU_Idle_Cycles Register (QMU Statistics Function)

Purpose Count of QMU Idle Clock Cycles.

Address 0xBDA001A8

Payload_NACK_Cnt Register (QMU Statistics Function)

Purpose Count of payload NACKs.

Address 0xBDA001AC

Global_NACK_Cnt Register (QMU Statistics Function)

Purpose Count of Global NACKs.

Address 0xBDA001B0

Payload_Read_Failures_Cnt Register (QMU Statistics Function)

Purpose Count of payload read failures.

Address 0xBDA001B4

Cmd_Processor_Err_Cnt Register (QMU Statistics Function)

Purpose Count of command processor errors, illegal opcodes and out of range queue numbers.

Address 0xBDA001B8

Q_Engine_Err_Cnt Register (QMU Statistics Function)

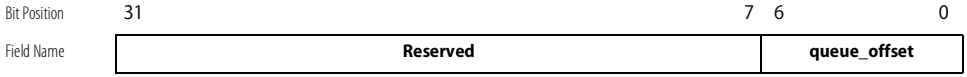
Purpose Count of queue engine errors.

Address 0xBDA001BC

Multicast_Destination0 to Multicast_Destination143 Registers (QMU Configuration Function)

Purpose Provide the mapping of the multicast destination port and queue level number to target a queue number for each leaf of a multicast elaboration.

Addresses 0xBDA00400 — 0xBDA0063C



Field Name	Bit Position	Description
Reserved	31:7	Read as zero.
queue_offset	6:0	**Destination Queue Offset** — The queue offset for the specified processor. The QMU adds this number to the processor's base number (*Base_Queue_CPnn, Base_Queue_XP, Base_Queue_FP)* to yield the target queue number. (Note: That only the least-significant seven bits of the address are used). These registers can only be accessed when the QMU is off-line. (*QMU_Run_Enable*=0). Refer to Table 156 on page 508 for detail *Queue_Offset* addresses.

Table 156 Multicast Mapping

Processor ID Number	Queue Level Number	Queue_Offset Address	Processor ID Number	Queue Level Number	Queue_Offset Address
0 for (CP0)	0	0xBDA00400	1 for (CP1)	0	0xBDA00420
	1	0xBDA00404		1	0xBDA00424
	2	0xBDA00408		2	0xBDA00428
	3	0xBDA0040C		3	0xBDA0042C
	4	0xBDA00410		4	0xBDA00430
	5	0xBDA00414		5	0xBDA00434
	6	0xBDA00418		6	0xBDA00438
	7	0xBDA0041C		7	0xBDA0043C
2 for (CP2)	0	0xBDA00440	3 for (CP3)	0	0xBDA00160
	1	0xBDA00444		1	0xBDA00164
	2	0xBDA00448		2	0xBDA00168
	3	0xBDA0044C		3	0xBDA0016C
	4	0xBDA00450		4	0xBDA00170
	5	0xBDA00454		5	0xBDA00174
	6	0xBDA00458		6	0xBDA00178
	7	0xBDA0045C		7	0xBDA0047C
4 for (CP4)	0	0xBDA00480	5 for (CP5)	0	0xBDA004A0
	1	0xBDA00484		1	0xBDA004A4
	2	0xBDA00488		2	0xBDA004A8
	3	0xBDA0048C		3	0xBDA004AC
	4	0xBDA00490		4	0xBDA004B0
	5	0xBDA00494		5	0xBDA004B4
	6	0xBDA00498		6	0xBDA004B8
	7	0xBDA0049C		7	0xBDA004BC

Table 156 Multicast Mapping (continued)

Processor ID Number	Queue Level Number	Queue_Offset Address	Processor ID Number	Queue Level Number	Queue_Offset Address
6 for (CP6)	0	0xBDA004C0	7 for (CP7)	0	0xBDA004E0
	1	0xBDA004C4		1	0xBDA004E4
	2	0xBDA004C8		2	0xBDA004E8
	3	0xBDA004CC		3	0xBDA004EC
	4	0xBDA004D0		4	0xBDA004F0
	5	0xBDA004D4		5	0xBDA004F4
	6	0xBDA004D8		6	0xBDA004F8
	7	0xBDA004DC		7	0xBDA004FC
8 for (CP8)	0	0xBDA00500	9 for (CP9)	0	0xBDA00520
	1	0xBDA00504		1	0xBDA00524
	2	0xBDA00508		2	0xBDA00528
	3	0xBDA0050C		3	0xBDA0052C
	4	0xBDA00510		4	0xBDA00530
	5	0xBDA00514		5	0xBDA00534
	6	0xBDA00518		6	0xBDA00538
	7	0xBDA0051C		7	0xBDA0053C
10 for (CP10)	0	0xBDA00540	11 for (CP11)	0	0xBDA00560
	1	0xBDA00544		1	0xBDA00564
	2	0xBDA00548		2	0xBDA00568
	3	0xBDA0054C		3	0xBDA0056C
	4	0xBDA00550		4	0xBDA00570
	5	0xBDA00554		5	0xBDA00574
	6	0xBDA00558		6	0xBDA00578
	7	0xBDA0055C		7	0xBDA0057C

Table 156 Multicast Mapping (continued)

Processor ID Number	Queue Level Number	Queue_Offset Address	Processor ID Number	Queue Level Number	Queue_Offset Address
12 for (CP12)	0	0xBDA00580	13 for (CP13)	0	0xBDA005A0
	1	0xBDA00584		1	0xBDA005A4
	2	0xBDA00588		2	0xBDA005A8
	3	0xBDA0058C		3	0xBDA005AC
	4	0xBDA00590		4	0xBDA005B0
	5	0xBDA00594		5	0xBDA005B4
	6	0xBDA00598		6	0xBDA005B8
	7	0xBDA0059C		7	0xBDA005BC
14 for (CP14)	0	0xBDA005C0	15 for (CP15)	0	0xBDA005E0
	1	0xBDA005C4		1	0xBDA005E4
	2	0xBDA005C8		2	0xBDA005E8
	3	0xBDA005CC		3	0xBDA005EC
	4	0xBDA005D0		4	0xBDA005F0
	5	0xBDA005D4		5	0xBDA005F4
	6	0xBDA005D8		6	0xBDA005F8
	7	0xBDA005DC		7	0xBDA005FC
16 for (XP)	0	0xBDA00600	17 for (FP)	0	0xBDA00620
	1	0xBDA00604		1	0xBDA00624
	2	0xBDA00608		2	0xBDA00628
	3	0xBDA0060C		3	0xBDA0062C
	4	0xBDA00610		4	0xBDA00630
	5	0xBDA00614		5	0xBDA00634
	6	0xBDA00618		6	0xBDA00638
	7	0xBDA0061C		7	0xBDA0063C

Free_Descriptor_Buffer_List Register (QMU Status Function)

Purpose Designates the total number of free descriptors.

Address 0xBDA7E008

Dyn_Descriptor_Pool0_Usage Register (QMU Status Function)

Purpose Designates how many descriptor buffers are in use in Pool0. Legal range= 0 to 16K. See Table 157 on page 511 for similar register.

Address 0xBDA7E080

Table 157 Dyn_Descriptor_Buffer_Usage_Pool*n* Register (for Pool1, 2 and 3)

Register Name	Purpose	Address
Dyn_Descriptor_Pool1_Usage	Same as Dyn_Descriptor_Pool0_Usage, but for pool1.	0xBDA7E084
Dyn_Descriptor_Pool2_Usage	Same as Dyn_Descriptor_Pool0_Usage, but for pool2.	0xBDA7E088
Dyn_Descriptor_Pool3_Usage	Same as Dyn_Descriptor_Pool0_Usage, but for pool3.	0xBDA7E08C

Buffer Management Unit (BMU) Configuration Registers

Configuration Space in the BMU contains a number of registers. The BMU uses these registers to configure and operate the BMU. The BMU uses others registers (WrCB0, RdCB0, RxCB0 and TxCB0) as described in Chapter 2 to move data through the BMU to/from SDRAM from/to the DMEM of either the requesting CPs, XP or FP.

These registers (WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB0_SDP_Addr; RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB0_SDP_Addr; RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr; and TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr) *are physically located in the Configuration Space of their respective CPs and not in the BMU Configuration Space.*

The BMU registers are located in the KSEG1 region, (0xA0000000 to 0xBFFFFFFF), which is uncached, starting at address 0xBDB00000. Refer to "C-5 NP Address Mapping" on page 50.

Warning: *Attempting to access a buffer pool before it is setup results in unpredictable behavior.*

BMU Registers

The following is a list of each register along with its address, function and reference to its detailed parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

Table 158 BMU Registers

Address	Register Name	Function	Detailed Parameters
0xBDB00000	Pool0 Base	Buffer Pool Base Address	See page 518
0xBDB00004	Pool1 Base		
0xBDB00008	Pool2 Base		
0xBDB0000C	Pool3Base		
0xBDB00010	Pool4 Base		
0xBDB00014	Pool5 Base		
0xBDB00018	Pool6 Base		
0xBDB0001C	Pool7 Base		
0xBDB00020	Pool8 Base		
0xBDB00024	Pool Base		
0xBDB00028	Pool10 Base		
0xBDB0002C	Pool11 Base		
0xBDB00030	Pool12 Base		
0xBDB00034	Pool13 Base		
0xBDB00038	Pool14 Base		
0xBDB0003C	Pool15 Base		
0xBDB00040	Pool16 Base		
0xBDB00044	Pool17 Base		
0xBDB00048	Pool18 Base		
0xBDB0004C	Pool19 Base		

Table 158 BMU Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDB00050	Pool20 Base	Buffer Pool Base Address (continued)	See page 518
0xBDB00054	Pool21 Base		
0xBDB00058	Pool22 Base		
0xBDB0005C	Pool23 Base		
0xBDB00060	Pool24 Base		
0xBDB00064	Pool25 Base		
0xBDB00068	Pool26 Base		
0xBDB0006C	Pool27 Base		
0xBDB00070	Pool28 Base		
0xBDB00074	Pool29 Base		
0xBDB10000	Pool0 BTag Shift	Encoded Buffer Size	See page 519
0xBDB10004	Pool1 BTag Shift		
0xBDB10008	Pool2 BTag Shift		

Table 158 BMU Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDB1000C	Pool3 BTag Shift	Encoded Buffer Size I (continued)	See page 519
0xBDB10010	Pool4 BTag Shift		
0xBDB10014	Pool5 BTag Shift		
0xBDB10018	Pool6 BTag Shift		
0xBDB1001C	Pool7 BTag Shift		
0xBDB10020	Pool 8 BTag Shift		
0xBDB10024	Pool9 BTag Shift		
0xBDB10028	Pool10 BTag Shift		
0xBDB1002C	Pool11 BTag Shift		
0xBDB10030	Pool12 BTag Shift		
0xBDB10034	Pool13 BTag Shift		
0xBDB10038	Pool14 BTag Shift		
0xBDB1003C	Pool15 BTag Shift		
0xBDB10040	Pool16 BTag Shift		
0xBDB10044	Pool17 BTag Shift		
0xBDB10048	Pool18 BTag Shift		
0xBDB1004C	Pool19 BTag Shift		
0xBDB10050	Pool20 BTag Shift		
0xBDB10054	Pool21 BTag Shift		
0xBDB10058	Pool22 BTag Shift		
0xBDB1005C	Pool23 BTag Shift		
0xBDB10060	Pool24 BTag Shift		
0xBDB10064	Pool25 BTag Shift		
0xBDB10068	Pool26 BTag Shift		
0xBDB1006C	Pool27 BTag Shift		
0xBDB10070	Pool28 BTag Shift		
0xBDB10074	Pool29 BTag Shift		
0xBDB20000	BTag FIFO Base0	BTag FIFO Base Address	See page 520

Table 158 BMU Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDB20004	BTag FIFO Base1	BTag FIFO Base Address (continued)	See page 520
0xBDB20008	BTag FIFO Base2		
0xBDB2000C	BTag FIFO Base3		
0xBDB20010	BTag FIFO Base4		
0xBDB20014	BTag FIFO Base5		
0xBDB20018	BTag FIFO Base6		
0xBDB2001C	BTag FIFO Base7		
0xBDB20020	BTag FIFO Base8		
0xBDB20024	BTag FIFO Base9		
0xBDB20028	BTag FIFO Base10		
0xBDB2002C	BTag FIFO Base11		
0xBDB20030	BTag FIFO Base12		
0xBDB20034	BTag FIFO Base13		
0xBDB20038	BTag FIFO Base14		
0xBDB2003C	BTag FIFO Base15		
0xBDB20040	BTag FIFO Base16		
0xBDB20044	BTag FIFO Base17		
0xBDB20048	BTag FIFO Base18		
0xBDB2004C	BTag FIFO Base19		
0xBDB20050	BTag FIFO Base20		
0xBDB20054	BTag FIFO Base21		
0xBDB20058	BTag FIFO Base22		
0xBDB2005C	BTag FIFO Base23		
0xBDB20060	BTag FIFO Base24		
0xBDB20064	BTag FIFO Base25		
0xBDB20068	BTag FIFO Base26		
0xBDB2006C	BTag FIFO Base27		
0xBDB20070	BTag FIFO Base28		
0xBDB20074	BTag FIFO Base29		

Table 158 BMU Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDB30000	Num BTag0	Number of BTags in a Pool	See page 520
0xBDB30004	Num BTag1		
0xBDB30008	Num BTag2		
0xBDB3000C	Num BTag3		
0xBDB30010	Num BTag4		
0xBDB30014	Num BTag5		
0xBDB30018	Num BTag6		
0xBDB3001C	Num BTag7		
0xBDB30020	Num BTag8		
0xBDB30024	Num BTag9		
0xBDB30028	Num BTag10		
0xBDB3002C	Num BTag11		
0xBDB30030	Num BTag12		
0xBDB30034	Num BTag13		
0xBDB30038	Num BTag14		
0xBDB3003C	Num BTag15		
0xBDB30040	Num BTag16		
0xBDB30044	Num BTag17		
0xBDB30048	Num BTag18		
0xBDB3004C	Num BTag19		
0xBDB30050	Num BTag20		
0xBDB30054	Num BTag21		
0xBDB30058	Num BTag22		
0xBDB3005C	Num BTag23		
0xBDB30060	Num BTag24		
0xBDB30064	Num BTag25		
0xBDB30068	Num BTag26		
0xBDB3006C	Num BTag27		

Table 158 BMU Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDB30070	Num BTag28	Number of BTags in a Pool (continued)	See page 520
0xBDB30074	Num BTag29		
0xBDB40000	Memory Size	Physical Memory Configuration, Test and Debug	See page 521
0xBDB40008	SDRAM Config		See page 522
0xBDB4000C	Single ECC Errors		See page 523
0xBDB40010	ECC Enable and Test Enable		See page 523
0xBDB40014	Debug Config		See page 524
0xBDB40018	Wr_Mem_Violation_Hi		See page 525
0xBDB4001C	Wr_Mem_Violation_Lo		See page 525

BMU Detailed Descriptions

The following is a detailed description of each of the BMU registers and their individual parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

Pool0 Base to Pool29 Base Registers (Buffer Pool Base Address Function)

Buffer pools must be configured during system initialization. Unpredictable behavior results when a pool is accessed prior to its initialization. The following registers are used to initialize buffer pools.

Purpose — Buffer pool base address. Width depends upon physical memory size: minimum value is 0, maximum value is the physical memory limit.

Software is responsible for ensuring that there is enough space to hold all of the pool's buffers.

Note: That the buffer pool "ends" at the next allocated piece of memory.

Address — 0xBDB00000 - 0xBDB00074

Access — Read/Write

Bit Position	31	24	23	0
Field Name	Rsvd		Pool Base	

Pool0 BTag Shift to Pool29 BTag Shift Registers (Buffer Size for a Pool Function)

Purpose BTag shift amount for address calculation. This value encodes the buffer size for a pool. Table 159 lists legal buffer sizes and their encodings.

Minimum value is 0. Maximum value is 10.

Address 0xBDB10000 - 0xBDB10074

Access Read/Write

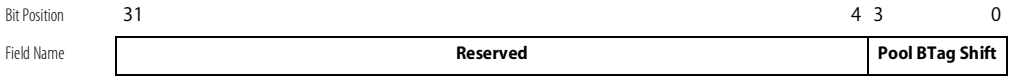


Table 159 BTag Shift Values and Corresponding Buffer Sizes

BTag Shift	Buffer Size	BTag Shift	Buffer Size
0	64kB	6	1kB
1	32kB	7	512B
2	16kB	8	256B
3	8kB	9	Not Supported
4	4kB	10	64B
5	2kB		

BTag FIFO Base0 to BTag FIFO Base29 Registers (BTag FIFO Base Address Function)

Purpose — Buffer pool BTag FIFO base address. Used for BTag FIFO management.

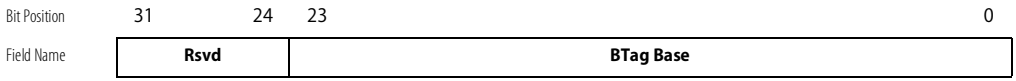
Software is responsible for ensuring that there is enough space to hold all of the pool's BTags.

Note: That the FIFO "ends" at the next allocated piece of memory. Each BTag consumes two bytes of memory.

Minimum value is 0. Maximum value is the physical memory limit.

Address — 0xBDB20000 - 0xBDB20074

Access — Read/Write

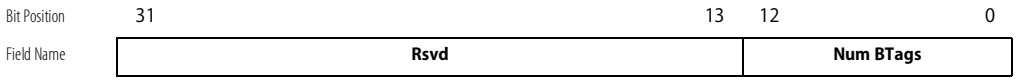


Num BTags0 to Num BTags29 Registers (Number of BTags in a Pool Function)

Purpose — Number of BTags in pool, in multiples of eight.

Address — 0xBDB30000 - 0xBDB30074

Access — Read/Write



Memory Size Register (Miscellaneous Function)

Purpose: Physical memory size in bytes. Software determines the amount of physical memory by writing and reading bit patterns to SDRAM. This configuration register is written with a value representing the amount of physical memory that software had determined that was present in the system.

Address: 0xBDB40000

Reset Value: 10

Access: Read/Write

Field Name	Bit Position	Description
Reserved	31:2	Read as zero.
phyMemSize	1:0	**Physical Memory Size** — Size of physical memory in bytes. Supported values are: **Encoded Value** / **Size** 00 / 64MB 01 / 128MB 10 / 256MB 11 / Reserved

SDRAM Config Register (Miscellaneous Function)

Purpose SDRAM controller configuration register.

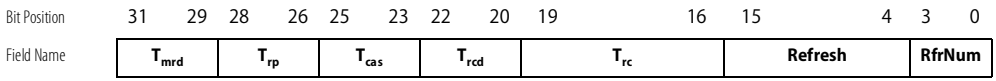
A write to this register tells the SDRAM controller the timing properties of the SDRAM and also initiates the SDRAM configuration process.

Note: That SDRAMs require a manufacturer specific initial settling time before the configuration process can begin. It is software's responsibility to ensure that this time has elapsed before the SDRAM configuration register is written.

Address 0xBDB40008

Reset Value 0

Access Read/Write



Field Name	Bit Position	Description
T_{mrd}	31:29	**Timing Mode Register Delay**
T_{rp}	28:26	**Precharge Command Period Timing**
T_{cas}	25:23	**CAS Timing** — CAS timing.
T_{rcd}	22:20	**Timing From Active to Command** — Active to command timing.
T_{rc}	19:16	**Timing RAS Cycle Time** — RAS cycle time.
Refresh	15:4	**Refresh Period** — Refresh rate of SDRAM. For Micron SDRAM memory parts, this value is computed as 15.625 µsec times the clock rate for the memory. For example: for 100MHz Micron SDRAM parts, this value must be less than or equal to 1562.
RfrNum	3:0	**Refresh Number** — Number of initial refreshes.

Single ECC Errors Register (Miscellaneous Function)

Purpose This read only register counts the number of single Error Correction Code (ECC) errors that have occurred.

Address 0xBDB4000C

Reset Value 0

Access Read

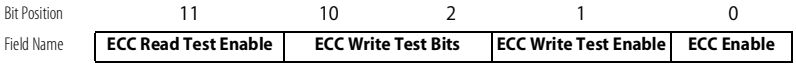
ECC Enable and Test Enable Register (Miscellaneous Function)

Purpose During normal operation, the Single Error Correction/Double Error Detecting (SECDED) error code if bit [0] is set to 1. ECC is disabled if bit [0] is set to 0. ECC test modes are controlled by bits [11:1].

Address 0xBDB40010

Reset Value 0

Access Read/Write



Field Name	Bit Position	Description
ECC Read Test Enable	11	**ECC Read Test Enable** – This enables the ECC read test function, placing ECC bits directly on the Payload Bus.
ECC Write Test Bits	10:2	**ECC Write Test Bits** – Provides ECC bits for ECC write test.
ECC Write Test Enable	1	**ECC Write Test Enable** –This bit enables the ECC write test function, writing ECC write test bits directly to SDRAM.
ECC Enable	0	**ECC Enable** – This bit enables ECC checking during normal operation.

Debug Config Register (Miscellaneous Function)

Purpose BMU C-5 NP debug register in canonical format.

Address 0xBDB40014

Access Read/Write

Bit Position	31	30 28	27 24	23	22 20	19 16	15	14 12	11 8	7 6	4 3 0
Field Name	Enb0	Rsvd	MUX0	Enb1	Rsvd	MUX1	Enb2	Rsvd	MUX2	Enb3	Rsvd	MUX3
Reset Value	0	raz	0	0	raz	0	0	raz	0	0	raz	0

Table 160 BMU Debug Inputs

Mux*n* Value	Event Chosen
0	Payload read
1	Payload write
2	Global read
3	Global write
4	BTag read
5	BTag write
6	BTag deallocation
7	Counter allocation
8	Counter decrement
9	Global read to SDRAM
10	Global write to SDRAM
11	BTag write to SDRAM
12	BTag deallocation to SDRAM
13	Counter decrement deallocation to SDRAM
14	BTag read to SDRAM
15	Write causing a memory violation

Wr_Mem_Violation_Hi Register (Miscellaneous Function)

Purpose Captures the global or payload address of transactions that led to the write error. Used in conjunction with Wr_Mem_Violation_Lo register.

Address 0xBDB40018

Access Read

Bit Position	7	6	5 ... 0
Field Name	Error	Bus	Payload_Addr_Ctl [37:32]
Reset Value	0	0	0

Field Name	Bit Position	Description
Error	7	**Error** — 1=Error, 0=No error.
Bus	6	**Bus** — 1=Global bus, 0=Payload bus.
Bus Addr	0:5	**Bus Addr** — Records the high order of the Payload bus error bits that caused the error.

Wr_Mem_Violation_Lo Register (Miscellaneous Function)

Purpose Captures the global or payload address of transactions that led to the write error. Used in conjunction with Wr_Mem_Viloation_Hi.

Address 0xBDB4001C

Access Read

Bit Position	31 ... 0
Field Name	Global_Addr [31:0] or Payload_Addr [31:0]
Reset Value	0

Field Name	Bit Position	Description
Bus Addr	31:0	**Bus Addr** — Records the Global or Payload bus that caused the error. If Payload this register capture only the low order bits. Where as, the high order are recorded in the *Wr_Mem_Violation_Hi* register.

Fabric Processor (FP) Configuration Registers

Configuration Space in the FP is an area that contains a number of registers. The FP uses these registers to communicate with the SDP and the bus controllers (Payload Bus and Global Bus). The FP performs flow mapping and management to and from the switching fabric.

FP Registers

The following is a list of each FP register and its address, function, and reference to its detailed parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions. You should also refer to "TxByte Processor Registers" on page 156.

Table 161 Fabric Processor Registers

Address	Register Name	Function	Detailed Parameters
0xBDD04000	TxFP_Enable	FP Tx Enable	See page 530
0xBDD04004	TxFI_Configuration	FP Tx Configuration	See page 530
0xBDD04008	TxDescInfo		See page 532
0xBDD0400C	TxDM_Header/Payload Delimiter		See page 532
0xBDD04010	TxQueueWeight_Configuration		See page 532
0xBDD04014	TxSysConfig		See page 534
0xBDD04018	TxFI_CRC		See page 534
0xBDD0401C	TxFCE_Configuration		See page 535
0xBDD04020	TxDebugMux_Control	FP Tx DeBug	See page 537
0xBDD04024	TxWCS_CAM	TxWCS_CAM Function	See page 539
0xBDD0402C	TxFlowTbl	FP Tx DeBug	See page 540
0xBDD04030	TxFlowTblDl		See page 540
0xBDD04034	TxFlowTblDH		See page 541
0xBDD04038	TxFlowCam	FP Tx Configuration	See page 541
0xBDD0403C	TxMergeAddr	FP Tx DeBug	See page 542
0xBDD04040	TxMergeData	FP Tx DeBug	See page 543
0xBDD04044	TxIdleData	FP Tx Configuration	See page 544
0xBDD04040	TxFDP_Mrg0 to TxFDP_Mrg63	FP Tx DeBug	See page 543

Table 161 Fabric Processor Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDD0404C	TxByte	FDP_CTL1 General Purpose	See page 544
0xBDD04048	TxByte	FDP_CTL0 General Purpose	See page 545
0xBDD04050	TxDebug Internal State	FP Tx DeBug	See page 545
0xBDE04090	RxStatus0	FP RxByte processor	See page 546
0xBDE04094	RxFlowSeg0		See page 546
0xBDE04098	RxFlowSz0		See page 547
0xBDE0409A	RxTxCgs0		See page 548
0xBDE04290	RxStatus1		See page 548
0xBDE04294	RxFlowSeg1		See page 549
0xBDE04298	RxFlowSz1		See page 550
0xBDE0429A	RxTxCgs1		See page 550
0xBDE04600	RxEnable_Configuration0	FP Rx Enable	See page 551
0xBDE04604	RxFI_Configuration	FP Rx Configuration	See page 551
0xBDE04608	RxDS_Header_Change1		See page 553
0xBDE0460C	RxDS_Header_Change2		See page 554
0xBDE04610	RxDS_Header/Payload_Delimiter0		See page 554
0xBDE04614	RxDS_Header/Payload_Delimiter1		See page 554
0xBDE04618	RxDS_Header/Payload_Delimiter2		See page 555
0xBDE0461C	RxDS_Configuration		See page 555
0xBDE04620	RxFI_CRC		See page 556
0xBDE04624	RxWCS_CAM	RxWCS_CAM Function	See page 557
0xBDE04628	RxByte0	FP Rx Configuration	See page 559
0xBDE0462C	RxByte1		See page 559
0xBDE04630	FCE_Configuration0		See page 559
0xBDE04634	FCE_Configuration1		See page 561
0xBDE04638	FCE_Configuration2		See page 562

Table 161 Fabric Processor Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDE04640	Pool0_CFG0	FP Rx Pool Configuration	See page 563
0xBDE04648	Pool1_CFG0		See page 563
0xBDE04650	Pool2_CFG0		See page 563
0xBDE04658	Pool3_CFG0		See page 563
0xBDE04644	Pool0_CFG1		See page 564
0xBDE0464C	Pool1_CFG1		See page 564
0xBDE04654	Pool2_CFG1		See page 564
0xBDE0465C	Pool3_CFG1		See page 564
0xBDE04660	RxByte_Shared0	FP RxByte Shared Function	See page 565
0xBDE04664	RxByte_Shared1		See page 565
0xBDE04680	RxFP_Interrupt_Event	FP Rx Interrupt	See page 566
0xBDE04684	RxFP_Interrupt_Enable		See page 567
0xBDE04688	RxFP_Debug_Event_Mux_Control	FP Rx DeBug	See page 567
0xBDE04690	RxMemory_Address		See page 570
0xBDE04694	RxMemory_Data		See page 570
0xBDE04698	RxPDU_ID_CAM		See page 570
0xBDE046A0	SEGS_RCVD	FP Rx Statistics	See Table 166 on page 572
0xBDE046A4	PDUS_RCVD		
0xBDE046A8	SEGS_LOST		
0xBDE046AC	PDUS_LOST		

Table 161 Fabric Processor Registers (continued)

Address	Register Name	Function	Detailed Parameters
0xBDE046C0	CPARITY_ERR	FP Rx Statistics (continued)	See Table 166 on page 572
0xBDE046C4	ERR_HDR		
0xBDE046C8	PARITY_ERR		
0xBDE046CC	LENGTH_ERR		
0xBDE046D0	Reserved		
0xBDE046D4	CRC_ERR		
0xBDE046D8	ODD_PDU		
0xBDE046DC	SEQ_ERR		
0xBDE046E0	SEQ_DIS		
0xBDE046E4	LOST_PDU		
0xBDE046E8	NO_FLOW_TBL		
0xBDE046EC	NO_BTAG		
0xBDE046F0	BTAG_ERR		
0xBDE046F4	ALLOC_ERR		
0xBDE046F8	ENQUE_ERR		
0xBDE04700	RxDebug_Internal_State		See page 573

FP Detailed Descriptions

The following is a detailed description of each of the FP registers and their individual parameters. The detailed parameters provide: purpose, field name, bit positions and descriptions.

TxFP_Enable Register (FP Tx Enable Function)

The transmit path is enabled via the *TxEnable* register.

The FPTx cannot be re-enabled after it has been disabled.

Purpose	Provides TxFP enable/disable.
Address	0xBDD04000
Access	Global Read/Write

Bit Position	31	30 ... 0
Field Name	Enable	Reserved
Reset Value	0	0

Field Name	Bit Position	Description
Enable	31	**Tx Fabric Port Enable** — 1 enables the internal FP Tx logic; 0 disables the FP Tx and holds it in reset.
Reserved	30:0	Read/write

TxFI_Configuration Register (FP Tx Configuration Function)

Purpose	Allows physical configuration of the TxFP Interconnect.
Address	0xBDD04004
Access	Global Read/Write

Bit Position	31	30 26	25	24	23	22	21	20	19	18	17	16	15 8	7	6 5	4	3	2	1	0
Field Name	Enable	Rsvd	U2 Tri Enable	U2 Mode	Variable Cells	Rsvd	PRIZMA	Rsvd	PowerX	Idle	Rsvd	Rsvd	SEG Size	Rsvd	Bus Width	BigEnd	ATM	OddP	Rsvd	RegIn
Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Enable	31	**CFI Enable** — 1 enables the external C-Port Fabric Interface; 0 disables (external pins are tri-stated).
Reserved	30:26	Read/write
U2 Tri Enable	25	**Utopia 2 Tri-state Enable** — Must be set to 1. 1 enables tri-state controls in Utopia 2 mode.

Field Name	Bit Position	Description
U2 Mode	24	**Utopia 2 Mode Enable** — 1 enables Utopia 2 mode; 0 disables Utopia 2 mode.
Variable Cells	23	**Variable Cell Size Enable** — 1 enables variable length cells; 0 disables.
Reserved	22	Read/write
PRIZMA	21	**PRIZMA Mode Enable** — 1 enables PRIZMA fabric mode; 0 disables the PRIZMA fabric mode. Cannot be set with PowerX bit 19.
Reserved	20	Read/write
PowerX	19	**PowerX Mode Enable** — 1 enables PowerX mode; 0 deselects PowerX mode. Cannot be set with PRIZMA bit 21.
Idle	18	**Idle Cell Enable** — 1 generates idle cell when the transmit FIFO is empty; 0 inhibits generation of idle cell. Idle cells are only supported for PRIZMA mode. Must be set in PRIZMA mode.
Reserved	17	Read/write
Reserved	16	Must be set to a 1.
SEG Size	15:8	**Segment Size**— Configures segment size of fabric (legal values are from 40 to 204Bytes). NOTE: The segment size must be a multiple of four Bytes (1 word). Unpredictable results may occur for values outside of this range or non word aligned.
Reserved	7	Must be set to zero.
Bus Width	6:5	**Bus Width** — Specifies Fabric bus width: **Encoded Value** / **Width (Bits)** 00 — 8 01 — 16 10 — undefined 11 — 32
BigEnd	4	**Select Big Endian** — 1 select big endian; 0 selects little endian.
ATM	3	**ATM** — Selects ATM or PHY mode for Utopia. 1 selects ATM; 0 selects PHY.
OddP	2	**Odd Parity** — 1 selects odd parity; 0 selects even parity.

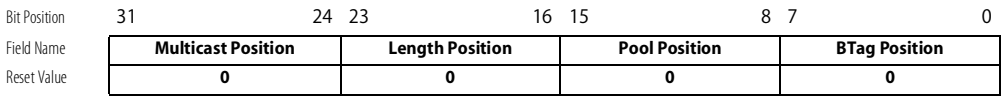
Field Name	Bit Position	Description
Reserved	1	Read/write
RegIn	0	**Select Registered Inputs** — 1 selects registered inputs; 0 selects non-registered inputs. Must be configured to 0 for Utopia2 mode. Must be a 1 for Utopia3 and PRIZMA. This bit is a “don’t care” for PowerX mode since there are no inputs to the FPTx; it is recommended to set the bit to a 1.

TxDescInfo Register (FP Tx Configuration Function)

Purpose Describes descriptor layout allowing TxFCE to extract key information.

Address 0xBDD04008

Access Global Read/Write



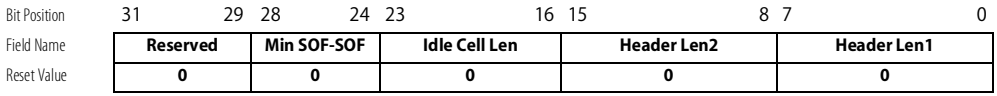
Field Name	Bit Position	Description
Multicast Position	31:24	**Multicast Position** — Offset bit position in descriptor of multicast bit.
Length Position	23:16	**Length Position** — Offset bit position in descriptor of PDU length.
Pool Position	15:8	**Pool Position** — Offset bit position in descriptor of Pool ID.
BTag Position	7:0	**BTag Position** — Offset bit position in descriptor of BTag.

TxDM_Header/Payload Delimiter Register (FP Tx Configuration Function)

Purpose Used by the Data Merge hardware to prepend the header to the payload.

Address 0xBDD0400C

Access Global Read/Write



Field Name	Bit Position	Description
Reserved	31:29	Read/write

Field Name	Bit Position	Description
Min SOF-SOF Spacing	28:24	**Minimum Cell Size** — Specifies minimum Start of Frame (SOF)-to-SOF timing, in terms of fabric clocks for short cells in PowerX mode.
Idle Cell Len	23:16	**Idle Cell Length** — Length of Idle Cell in bus cycles. (Used in conjunction with *TxFI Configuration* register's *Idle Cell Enable* field and the *Idle Cell Header* register.) Only in PRIZMA mode.
Header Len2	15:8	**Header Length 2** — Length of middle and last segment headers in Bytes.
Header Len1	7:0	**Header Length 1** — Length of first and only segment headers in Bytes.

TxQueueWeight_Configuration Register (FP Tx Configuration Function)

Purpose A globally accessible register that allows the configuration of relative FPTx queue priorities.

Address 0xBDD04010

Access Global Read/Write

Bit Position	31 28	27 24	23 17	16	15 11	10 7	6 0
Field Name	Reserved	Queue Wgt (Rd)	Reserved	Write	Reserved	Queue Wgt(Wr)	Queue Number
Reset Value	raz	n/a	0	0	0	0	0

Field Name	Bit Position	Description
Reserved	31:28	Read as zero.
Queue Wgt (Read only)	27:24	**Queue Weight Read Data** — When read with the *Write* field (bit 16) set to zero (0), it provides the weight of specified queue number.
Reserved	23:17	Read/write
Write	16	**Write** — Writes queue weight value to queue number specified.
Reserved	15:11	Read/write
Queue Wgt (Write only)	10:7	**Queue Weight Write Data** — Weight value to be written to queue number specified (default weight for each queue is 1).
Queue Number	6:0	**Queue Number** — Number of the queue to be accessed.

TxSysConfig Register (FP Tx Configuration Function)

Purpose Sets the base queue number for the FP.

Address 0xBDD04014

Access Global Read/Write

Bit Position	31 … 25	24 … 16	15 … 0
Field Name	Reserved	Queue Offset	Fabric ID
Reset Value	0	0	0

Field Name	Bit Position	Description
Reserved	31:25	Read/write
Queue Offset	24:16	**Queue Offset** — Base queue number for FP block of 128 queues in QMU.
Fabric ID	15:0	Reserved

TxFI_CRC Register (FP Tx Configuration)

Purpose The *TxFI CRC* register configures the CRC function. The result can be configured in two ways:

- Initial value of the CRC accumulator (0 or all 1s)
- Inverted (one's complement)

The *TxFI CRC* provides index fields that allow the CRC to be calculated over any sequential portion of the segment, and then appended or inserted anywhere afterward.

Address 0xBDD04018

Access Global Read/Write

Bit Position	31	30 … 28	27	26	25	24	23 … 16	15 … 8	7 … 0
Field Name	Enable	Reserved	Rsvd	CRC Ini1	Rsvd	Invert	First Index	Last Index	Append Index
Reset Value	0	0	1	0 or 1	0	0 or 1	0	0	0

Field Name	Bit Position	Description
Enable	31	**CRC Enable** — 1 enables CRC mechanism; 0 disables and leaves the CRC mechanism in reset
Reserved	30:28	Read/write
Reserved	27	Must be set to 1

Field Name	Bit Position	Description
Init1	26	**CRC Initialization** — 1 initializes the CRC register to all 1s; 0 initializes it to a 0.
Reserved	25	Must be set to 0
Invert	24	**Invert CRC** — 1 selects CRC to be inverted prior to being appended to Segment; 0 selects not inverted.
First Index	23:16	**First Index** — Offset from Byte 0 of segment to start of CRC accumulation Byte. Index must be a multiple of 4.
Last Index	15:8	**Last Index** — Must be equal to (cell size - 4). This represents the offset (plus 1 byte) from byte 0 of segment to the last byte to be part of the CRC accumulation.
Append Index	7:0	**Append Index** — Byte offset from byte 0 of segment to appended CRC (currently not supported).

TxFCE_Configuration Register (FP Tx Configuration Function)

Purpose Configures FCE descriptor size, queue configuration, and flow mask.

Address 0xBDD0401C

Access Global Read/Write. Bits [31:28] are read only from a global perspective; they are written by FP hardware.

Bit Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	0
Field Name	QMU Parity Error	Rd Error	Wr Error	QMU Error	INT Ack	INT Enable	Desc Size		Rsvd		QMU Parity Error Enable	PDU Pause	FC Enab.	Queue Depth	Rsvd	Rsvd	Flow Mask	
Reset Value	0	0	0	0	0	0	0		0		0	0	0	0	1	0	0	

Field Name	Bit Position	Description
QMU Parity Error	31	**QMU Parity Error** — Indicates QMU dequeue parity error status.
Rd Error	30	**Read Error** — Indicates the TxFCE Read Control Block transfer failed.
Wr Error	29	**Write Error** — Indicates the TxFCE Write Control Block transfer failed.
QMU Error	28	**QMU Error** — Indicates the TxFCE dequeue operation failed.
INT Ack	27	**Interrupt Acknowledge** — Set to a 1 to acknowledge and clear an interrupt, then set to 0.

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>INT Enable</td><td>26</td><td>Interrupt Enable — 1 enables interrupts to be generated to the XP for the following errors (see bits 31:28):
QMU parity error
Read error
Write error
QMU error</td></tr>
<tr><td>Desc Size</td><td>25:24</td><td>Descriptor Size:<table><tr><th>Encoded Value</th><th>Size (Bytes)</th></tr><tr><td>00</td><td>12</td></tr><tr><td>01</td><td>16</td></tr><tr><td>10</td><td>24</td></tr><tr><td>11</td><td>32</td></tr></table></td></tr>
<tr><td>Reserved</td><td>23:22</td><td>Read/write</td></tr>
<tr><td>QMU Parity Error Enable</td><td>21</td><td>QMU Parity Error Enable— 1 enables QMU dequeue parity error checking, 0 disables parity checking.</td></tr>
<tr><td>PDU Pause</td><td>20</td><td>Reserved. Must be set to zero</td></tr>
<tr><td>FC Enable</td><td>19</td><td>Flow Control Enable— 1 enables per-queue flow control; 0 disables per-queue flow control.</td></tr>
<tr><td>Queue Depth</td><td>18</td><td>Queue Depth — 1 selects 16 queues x eight priorities; 0 selects 32 queues x four priorities.</td></tr>
<tr><td>Reserved</td><td>17</td><td>Must be set to 1.</td></tr>
<tr><td>Reserved</td><td>16</td><td>Read as zero.</td></tr>
<tr><td>Flow Mask</td><td>15:0</td><td>Flow Mask — Used to mask ‘x’ bits of 16bit Flow ID during CAM operation on congestion messages.
When matching entries in the Tx Flow ID CAM using the TxFlowCam register (0xBDD0438), you must be sure to also set this register’s Flow Mask field. The value of the Flow Mask is AND’ed with the value of the TxFlowCam register's Match field to obtain the result submitted to the CAM. The default value for the Flow Mask is zero. Hence failing to set the Flow Mask means you will never match any entry in the CAM.</td></tr>
</table>

TxDebugMux_Control Register (FP Tx DeBug Function)

For the purposes of debug, most of the internal registers are made visible via Global Address Space, and a limited number of events (usually used to count) can be viewed via the *TxDebugMux* register.

Purpose — Allows you to monitor events for FPTx debug purposes. These events are selectable. Refer to Table 162 on page 538.

For the purposes of debug, specific monitoring points with in the FP Tx are wired to the event register as selected by the *TxDebugMux Control* register.

Address — 0xBDD04020

Access — FDP Read/Write, Global Read

The selectable events are shown in Table 162. Any event can be viewed in association with any of the four selection fields, including simultaneously being selected in more than one field (that is, viewed multiple times).

Bit Position	31	30 28	27 24	23	22 20	19 16	15	14 12	11 8	7	6 4	3 0
Field Name	EN0	RSVD	SEL0	EN1	RSVD	SEL1	EN2	RSVD	SEL2	EN3	RSVD	SEL3
Reset Value	0	0	0	0	0	0	0	0	0	0	0	0

Field Name	Bit Position	Description
EN0	31	**TxDebug Event Mux Control Enable 0** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	30:28	Read/write
SEL0	27:24	**TxDebug Event Mux Control Select 0** — Selects one of the eight FP Tx events to be viewed for the corresponding field.
EN1	23	**TxDebug Event Mux Control Enable 1** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	22:20	Read/write
SEL1	19:16	**TxDebug Event Mux Control Select 1** — Selects one of the eight FP Tx events to be viewed for the corresponding field.
EN2	15	**TxDebug Event Mux Control Enable 2** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	14:12	Read/write

Field Name	Bit Position	Description
SEL2	11:8	**TxDebug Event Mux Control Select 2** — Selects one of the eight FP Tx events to be viewed for the corresponding field.
EN3	7	**TxDebug Event Mux Control Enable 3** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	6:4	Read/write
SEL3	3:0	**TxDebug Event Mux Control Select 3** — Selects one of the eight FP Tx events to be viewed for the corresponding field.

Table 162 FPTx_Debug Monitored Events

Select Value	Monitored Event	Description
0	Congestion control, stops receive	Pulses once per flow stop request from the RxByte processor.
1	Cell complete	Pulses once per cell.
2	PDU complete	Pulses once per complete PDU transmitted.
3	DMA request	Pulses once per payload DMA.
4	PDU resume	Pulses once per PDU resumed after a per flow stop.
5	Fabric FIFO empty	Active for every clock that the transmit FIFO is empty.
6	Send pause	Pulses once per RxCFI request to send a link-level pause.
7	Pause	Pulses once per FPRx request to link-level pause.

TxWCS_CAM (WCS_CAM Function)

Purpose Interface in global address space to initialize FP Tx Byte processor WCSs and CAMs.

Address 0xBDD04024

Access Global Read / Write (bits [7:6] are read only)

Bit Position	31 16	15 8	7 6	5	4	3	2	1	0
Field Name	Reserved	WCS Write Data	WCS Scan Out	WCS Write Cmd	CAM Reset	CAM Update	WCS/CAM Capture	WCS DATA IN1	WCS DATA IN0
Reset Value	0	0	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Reserved	31:16	Read/write
WCS Write Data	15:8	**TxWCS_CAM WCS Write** — Data to be written to both WCSes via the Byte write interface.
WCS Scan Out	7:6	**TxWCS_CAM WCS Scan** — Read only. WCS scan shift out for Byte processors 1 & 0 (in that order).
WCS Write Cmd	5	**TxWCS_CAM WCS Write Cmd** — Setting this to a 1 launches a Byte write to both WCSes. Cleared by hardware.
CAM Reset	4	**TxWCS_CAM Reset** — Setting this to a 1 resets the TxByte CAM.
CAM Update	3	**TxWCS_CAM Update** — Setting this to a 1 updates the CAM array from the CAM's shift registers. Cleared by hardware.
WCS/CAM Capture	2	**TxWCS_CAM WCS/CAM Capture** — Setting this to a 1 launches a WCS and CAM scan capture for diagnostic purposes (loads data into the WCS and CAM shift registers). Cleared by hardware.
WCS DATAIN1	1	**TxWCS_CAM WCS DATAIN1** — Setting this to a 1 shifts a 1 into the WCSes. Cleared by hardware.
WCS DATAIN0	0	**TxWCS_CAM WCS DATAIN0** — Setting this to a 1 shifts a 0 into the WCSes. Cleared by hardware.

TxFlowTbl Register (FP Tx DeBug Function)

Purpose Allows access to global address space read flow table inside the TxFCE.

Address 0xBDD0402C

Access Global Read/Write, 128 60bit entries

Bit Position	31 … 17	16	15 … 8	7 … 0
Field Name	Reserved	WT	Reserved	ADDR
Reset Value	0	0	0	0

Field Name	Bit Position	Description
Reserved	31:17, 15:8	Read/write.
WT	16	**Write Flow Table** — Writing a 1 initiates a flow table write. This bit is automatically cleared by hardware.
ADDR	7:0	**Flow Table Address** — Address of flow table to write or read NOTE: Only the first128 entries are valid.

TxFlowTblDL Register (FP Tx DeBug Function)

Purpose Least significant word of flow table data.

Address 0xBDD04030

Access Global Read/Write

Bit Position	31 … 0
Field Name	DATA_LOW
Reset Value	Undefined

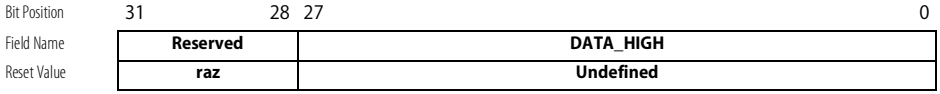
Field Name	Bit Position	Description
DATA_LOW	31:0	This is the low order data which was read from a flow table read, or the data to be written on a flow table write.

TxFlowTblDH Register (FP Tx DeBug Function)

Purpose Most significant word of flow table data.

Address 0xBDD04034

Access Global Read/Write



Bit Position	31 28	27 0
Field Name	Reserved	DATA_HIGH
Reset Value	raz	Undefined

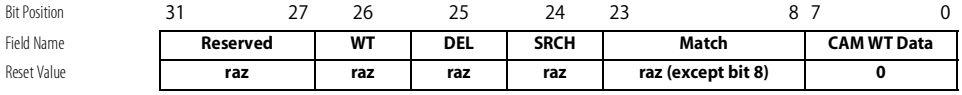
Field Name	Bit Position	Description
Reserved	31:28	Read As Zero (raz)
DATA_HIGH	27:0	This is the high order data which was read from a flow table read, or the data to be written on a flow table write.

TxFlowCam Register (FP Tx DeBug Function)

Purpose Interface to global address space to initialize the FDP TxFlow Cam store.

Address 0xBDD04038

Access Global Read/Write



Bit Position	31 27	26	25	24	23 8	7 0
Field Name	Reserved	WT	DEL	SRCH	Match	CAM WT Data
Reset Value	raz	raz	raz	raz	raz (except bit 8)	0

Field Name	Bit Position	Description
Reserved	31:27	Read as zero.
WT	26	**Write CAM Location** — This bIt is always read as zero.Writes the location matched, or the next free location if nothing matches (for diagnostic purposes only). Setting this bit to a 1 launches a CAM write of the write data. After the CAM write, the bit is cleared by the hardware. 1 = write Cam entry 0 = do not write Cam entry

Field Name	Bit Position	Description
DEL	25	**Delete CAM Entry** — This bit is always read as zero.Deletes CAM entry matched (for diagnostic purposes only). Setting this bit to a 1 launches a CAM delete of the entry corresponding to the previous match value. After the CAM write, the bit is cleared by the hardware. **Encoded Value** / **CAM Action** 1 — Delete Cam entry 0 — Do not delete Cam entry
SRCH	24	**CAM Search** — 1 selects CAM search. This bit is always read as zero.
Match	23:8	**16Bit CAM Match Value** — Value to search on. On reads, bits 23:9 return zeros, while bit 8 returns the CAM free indication. When matching entries in the Tx Flow ID CAM using this register, you must be sure to also set the *TxFCE Configuration* (0xBDD0438) register's *Flow Mask* field. The value of the *Flow Mask* is *AND'ed* with the value of the *TxFlowCam* register's *Match* field to obtain the result submitted to the CAM. The default value for the *Flow Mask* is zero. Hence, failing to set the *Flow Mask* means you will never match any entry in the CAM.
CAM WT Data	7:0	**CAM Write Data** — CAM data read from the CAM, or to be written to the CAM. Field is 6:0, but bit 7 is always set to 0.

TxMergeAddr (FPTx Debug Function)

Purpose — Used to select the Merge Block address and access (read/write), bits [5] Selects Merge block 1/ 0, and bits [4:0] select the word offset (0-31).

Global Address — 0xBDD0403C

Access — Global Read, Global Write

Bit Position	31 … 17	16	15 … 8	5 … 0
Field Name	Reserved	WMB	Reserved	ADDR
Reset Value	raz	0	raz	0

Field Name	Bit Position	Description
Reserved	31:17	Read as zero.

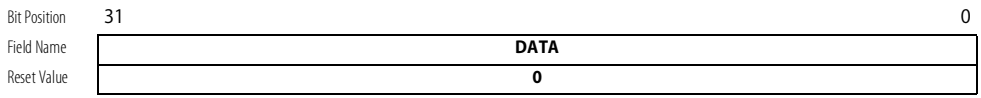
Field Name	Bit Position	Description
WMB	16	**Write Merge Block** — 1 selects write; 0 selects read.
Reserved	15:8	Read as zero.
ADDR	5:0	**Merge Space Address** — Address of Merge Space to write/read.

TxMergeData (FPTx Debug Function)

Purpose Used to Write / Read Data from each Merge block

Global Address 0xBDD04040

Access Global Read, Global Write

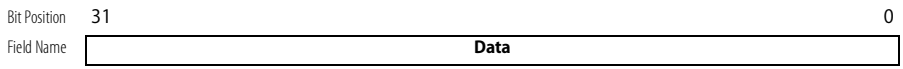


TxFDP_Mrg0 - TxFDP_Mrg63

Purpose Used for passing Merge data to the transmit data stream from the Descriptor

Address Globally accessed via Mailbox registers shown above (TxMergeAddr (FPTx Debug Function) and TxMergeData (FPTx Debug Function).

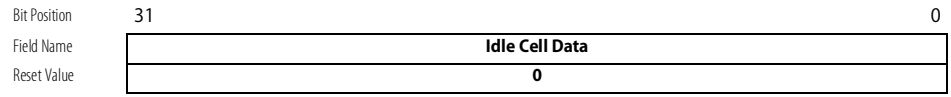
Access Global Read, Global Write (during test), FDP Receive Byte processor Write - Byte addressable



TxIdleData Register (FP Tx Configuration Function)

Purpose	Provides data for idle cells.
Address	0xBDD04044
Access	Global Read/Write

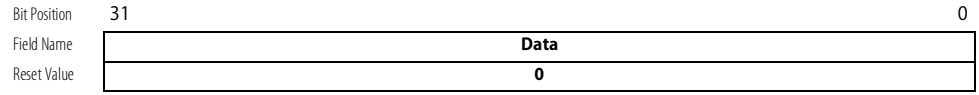
This register is meaningful only in PRIZMA mode.



Field Name	Bit Position	Description
Idle Cell Data	31:0	**Idle Cell Data** — Four Bytes of data for Idle cell generation. (Used in conjunction with the *TxFI Configuration* register's *Idle Cell Enable* field, and the *TxDM Header Payload Delimiter* register).

TxFDP_CTL0 Register (TxByte General Purpose Function)

Purpose	General purpose data which is accessible globally and by the FDP. There are 2 32bit registers (FDP_CTL0, FDP_CTL1).
Address	0xBDD04048 (FDP_CTL0),
Access	Global Read, Global Write, FDP Read



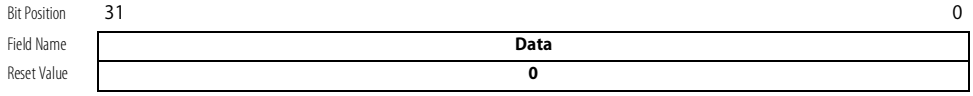
Field Name	Bit Position	Description
Data	31:0	Global Read/Write, FDP Read

TxFDP_CTL1 Register (TxByte General Purpose Function)

Purpose General purpose data which is accessible globally and by the FDP. There are 2 32bit registers (FDP_CTL0, FDP_CTL1).

Address 0xBDD0404C (FDP_CTL1)

Access Global Read, Global Write, FDP Read



Field Name	Bit Position	Description
Data	31:0	Global Read/Write, FDP Read

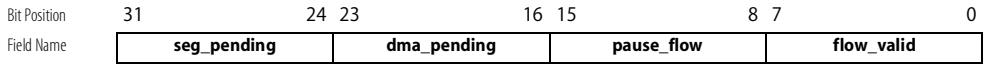
TxDebug_Internal_State Register (FP Tx DeBug Function)

Purpose The *TxDebug State* register has 4 8bit fields that provide an internal signal debug for the eight internal scopes.

While the FP Tx handles 128 flows, only eight flows (flows 0 through 7) are *active* at any one time. Thus the bit fields indicate the state of each active flow (within each field the LSB identifies flow 0 and the MSB identifies flow 7).

Address 0xBDD04050

Access Global Read, Written by FDP hardware



Field Name	Bit Position	Description
seg_pending	31:24	**Segment Pending**— 1 indicates the flow(s) (0-7) have started to transmit a segment and are waiting for the segment transmit to complete prior to getting the PDU segment.
dma_pending	23:16	**DMA Pending**— 1 indicates the flow(s) (0-7) have requested a PDU segment to be DMA'ed from DRAM to DMEM and are currently awaiting the segment DMA to complete so that it can be transmitted.

Field Name	Bit Position	Description
pause_flow	15:8	**Pause Flow**— 1 indicates the flow(s) (0 through 7) have been paused (that is, flow controlled) and will be moved to the 'sleep' state to allow another ready' flow to go *active*.
flow_valid	7:0	**Flow Valid**— 1 indicates the flow(s) (0 through 7) are actively transmitting. 0 indicates that the active flow slot is not being used, that is, there are no additional flows ready to be transmitted.

RxStatus0 Register (FP RxByte Processor Function)

Purpose Read/modified Writes governing FDP receive operation for datascope0.

Address 0xBDE04090

FDP Address 0x80

Access FDP Read/Write, Global Read

Bit Position	7	6	1	0
Field Name	OWN	Reserved		FID VLD
Reset Value	1	raz		0

Field Name	Bit Position	Description
OWN	7	**Extract Space Ownership** — 1 = FCE owns; 0 = FDP owns.
Reserved	6:1	Read as zero.
FID_VLD	0	**Flow ID Valid** — Indicates the FID is valid allowing hardware to pipeline payload transfer at the earliest possible moment. 1 = FID valid; 0 = FID not valid.

RxFlowSeg0 Register (FP RxByte Processor Function)

The *Flow Segment* register is used by the RxByte processor to associate a segment with a flow. Specifically the RxByte extracts the Flow ID, Segment type, and whether the segment is part of a multicast flow.

Purpose Associates a received Segment to a Flow

Address 0xBDE04094

Access FDP Read/Write, Global Read

Bit Position	31 27	26	25 24	23 18	17	16	15 0
Field Name	Reserved	Multicast	Seg Type	Reserved	Flow Disc	Flow Error	PDU ID
Reset Value	raz	0	0	raz	0	0	0

Field Name	Bit Position	Description
Reserved	31:27	Read as zero.
Mulitcast	26	**Multicast Enable** — Reserved. Must be set to 0.
Seg Type	25:24	**Segment Type** — Defines the Segment (PDU Segment): **Encoded Value** / **Type** 00 — Middle segment 01 — End of Message (last) segment 10 — Beginning of Message (first) segment 11 — First and only segment (last and first)
Reserved	23:18	Read as zero.
Flow Disc	17	**Flow Discard** — 1 = Discards flow; 0 = does not discard flow.
Flow Error	16	**Flow Error** — 1 = FDP detects error (packet discarded with error); 0 = FDP detects no error.
PDU ID	15:0	**PDU ID** — The 16bit PDU ID.

RxFlowSz0 Register (FP Rx Byte Processor Function)

Purpose Identifies the Length of the PDU for a given flow.

Address 0xBDE04098

Access FDP Read/Write, Global Read

Bit Position	15 0
Field Name	Rx Flow Size
Reset Value	0

Field Name	Bit Position	Description
Rx Flow Size	15:0	**Receive Flow Size** — Length in Bytes of PDU of the current flow.

RxTxCgs0 Register (FP Rx Byte Processor Function)

Purpose Transmit Congestion Flow ID, signals transmit side that a specific flow has congestion and should be stopped.

Address 0xBDE0409A

Access FDP Read/Write, Global Read

Bit Position	31 22	21	20 16	15 0
Field Name	Reserved	PAUSE_RES	Reserved	Flow ID
Reset Value	raz	0	raz	0

Field Name	Bit Position	Description
Reserved	31:22	Read as zero.
PAUSE_RES	21	**Pause Resume** — 1 = disable transmit (Pause), 0 = enables or resumes transmit. NOTE: Writing the *PAUSE_RES* bit validates the data in the register. This bit MUST be written last.
Reserved	20:16	Read as zero.
Flow ID	15:0	**Flow ID** — 16bit Flow ID (FID) that is masked and then mapped to a 7bit queue Index by the TxFDP.

RxStatus1 Register (FP RxByte Processor Function)

Purpose Read/modified Writes governing FDP receive operation for datascope1.

Address 0xBDE04290

FDP Address 0x80

Access FDP Read/Write, Global Read

Bit Position	7	6 1	0
Field Name	OWN	Reserved	FID VLD
Reset Value	1	raz	0

Field Name	Bit Position	Description
OWN	7	**Extract Space Ownership** — 1 = FCE owns; 0 = FDP owns.
Reserved	6:1	Read as zero.

Field Name	Bit Position	Description
FID_VLD	0	**Flow ID Valid** — Indicates the FID is valid allowing hardware to pipeline payload transfer at the earliest possible moment. 1 = FID valid; 0 = FID not valid

RxFlowSeg1 Register (FP RxByte Processor Function)

The *RxFlow Segment* register is used by the RxByte Byte processor to associate a segment with a flow. Specifically the RxByte extracts the Flow ID, Segment type, and whether the segment is part of a multicast flow.

Purpose Associates a received Segment to a Flow

Address 0xBDE04294

Access FDP Read/Write, Global Read

Bit Position	31 27	26	25 24	23 18	17	16	15 0
Field Name	Reserved	Multicast	Seg Type	Reserved	Flow Disc	Flow Error	PDU ID
Reset Value	raz	0	0	raz	0	0	0

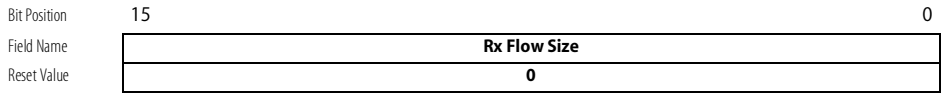
Field Name	Bit Position	Description
Reserved	31:27	Read as zero.
Mulitcast	26	**Multicast Enable** — Reserved. Must be set to 0.
Seg Type	25:24	**Segment Type** — Defines the Segment (PDU Segment): **Encoded Value** / **Type** 00 — Middle segment 01 — End of Message (last) segment 10 — Beginning of Message (first) segment 11 — First and only segment (last and first
Reserved	23:18	Read as zero.
Flow Disc	17	**Flow Discard** — 1 = Discards flow; 0 = does not discard flow.
Flow Error	16	**Flow Error** — 1 = FDP detects error (packet discarded with error); 0 = FDP detects no error.
PDU ID	15:0	**PDU ID** — The 16bit PDU ID.

RxFlowSz1 Register (FP RxByte Processor Function)

Purpose Identifies Length of PDU for a given flow.

Address 0xBDE04298

Access FDP Read/Write, Global Read



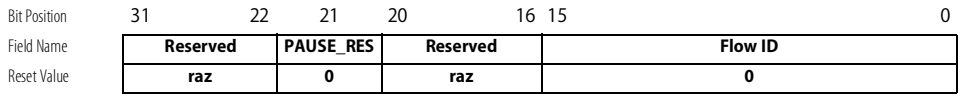
Field Name	Bit Position	Description
Rx Flow Size	15:0	**Receive Flow Size** — Length in Bytes of PDU of the current flow.

RxTxCgs1 Register (FP RxByte Processor Function)

Purpose Transmit Congestion Flow ID, signals transmit side that a specific flow has congestion and should be stopped.

Address 0xBDE0429A

Access FDP Read/Write, Global Read



Field Name	Bit Position	Description
Reserved	31:22	Read as zero.
PAUSE_RES	21	**Pause Resume** — 1 = disable transmit (Pause), 0 = enables or resumes transmit. NOTE: Writing the Byte that includes the *PAUSE_RES* bit validates the data in the register. This Byte MUST be written last.
Reserved	20:16	Read as zero.
Flow ID	15:0	**Flow ID** — 16bit Flow ID (FID) that is masked and then mapped to a 7bit queue Index by the TxFDP.

RxEnable_Configuration Register (FP Rx Enable Function)

The internal FP Rx logic is enabled via the RxEnable Register.

A separate enable bit exists in the RxFI register for the FP Rx external fabric interface.

Purpose Setting the enable to a 1 turns on the internal FP Rx logic.

Address 0xBDE04600

Access Global Read/Write

Bit Position	31	30 … 0
Field Name	Enable	Reserved
Reset Value	0	raz

Field Name	Bit Position	Description
Enable	31	**FP Rx Enable** — 1 enables FP Rx; 0 disables FP Rx and holds it in reset.

RxFI_Configuration Register (FP Rx Configuration Function)

The FDP contains eight configuration registers residing in global address space allocated to the FP block. These registers enable configuration of each of the stages and are defined below.

Purpose Allows physical configuration of the FP Rx Interconnect.

Address 0xBDE04604

Access Global Read, Global Write

Bit Position	31	30 27	26	25	24	23	22 16	15 8	7	6	5 4	3	2	1	0
Field Name	CFI Enable	Inf	Byte Parity	Odd Parity	Check Parity	Rsvd	Ctrl Mask	SEG Size	CLH	Fab Size	Fabric Width	Big Endian	Rsvd	ATM	Reg Input
Reset Value	0	0	0	1	0	raz	0	0	1	0	0	1	raz	0	1

Field Name	Bit Position	Description
CFI Enable	31	**Interface Enable Bit** — 1 enables the Fabric Interface state machines and I/O buffers; 0 disables Fabric Interface (external pins are tri-stated). This enable, separate from the Rx Enable, allows software to delay the asynchronous payload from entering the port until the C-5 NP synchronous portions of the FP are ready.

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>Interface</td><td>30:27</td><td>Interface Type — Interface type encoding.
<table>
<tr><th>Encoded Value</th><th>Function</th></tr>
<tr><td>0x0</td><td>Utopia 3</td></tr>
<tr><td>0x1</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>Reserved</td></tr>
<tr><td>0x3</td><td>PowerX</td></tr>
<tr><td>0x4</td><td>PRIZMA</td></tr>
<tr><td>0x5</td><td>Utopia1 and 2</td></tr>
<tr><td>0x6 to 0xF</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>Byte Parity</td><td>26</td><td>Byte Parity — 1 selects parity on each Byte lane. 0 deselects Byte lane parity. Used in PowerX mode only. When selected, one bit of parity is matched for each Byte of fabric width (pins CFI_CTL [6:3] = P[0:3]). Must be 1 for PowerX mode, 0 for all other modes.</td></tr>
<tr><td>Odd Parity</td><td>25</td><td>Odd Parity — 1 selects odd parity check, 0 selects even parity.</td></tr>
<tr><td>Check Parity</td><td>24</td><td>Check Parity — 1 enables parity checking, 0 disables parity checking.</td></tr>
<tr><td>Reserved</td><td>23, 2</td><td>Read as zero.</td></tr>
<tr><td>Parity Ctrl Mask</td><td>22:16</td><td>Parity Control Mask — Only used in PowerX mode and must be set to 7. Mask used to identify the valid CFI control pins that parity is calculated over. (Parity of selected control pins is always XOR’ed to least significant parity bit).</td></tr>
<tr><td>SEG Size</td><td>15:8</td><td>Segment Size — Configures segment size of fabric (legal values are from 40 to 204Bytes).
NOTE: The segment size must be a multiple of four Bytes (1 word). Unpredictable results may occur for values outside of this range or non word aligned.</td></tr>
<tr><td>CLH</td><td>7</td><td>Cell Level Handshake — Must be set to 1. When selected, flow control threshold values in RxDS Header register (see page 553) enable flow control on nearest cell boundary. When not selected, flow control threshold values are directly used to assert flow control.</td></tr>
<tr><td>Fabric Size</td><td>6</td><td>Reserved</td></tr>
</table>

Field Name	Bit Position	Description
Fabric Width	5:4	**Fabric Bus Width** — Specifies Fabric bus width: **Encoded Value** / **Width (Bits)** 00 / 8 01 / 16 10 / Reserved 11 / 32
Big Endian	3	**Big Endian** — 1 selects Big Endian mode, 0 selects Little Endian mode.
ATM	1	**ATM Mode** — 1 selects ATM mode, 0 selects PHY mode. This bit is meaningful only in Utopia mode.
Reg Input	0	**Register Control and Data Input Pins** — Used in all modes except UTOPIA1 and 2. 1 selects Registered Input; 0 selects non-Registered Input.

RxDS_Header_Change1 Register (FP Rx Configuration Function)

Purpose Configures Data Separator use of Payload Delimiter0 and Payload Delimiter1 registers

Address 0xBDE04608

Access Global Read/Write

Bit Position	31	30 — 24	23 — 16	15 — 8	7 — 0
Field Name	Change	Reserved	Change Index	Change Value	Change Mask
Reset Value	0	raz	0	0	0

Field Name	Bit Position	Description
Change	31	**Change** — Enable ability to switch from *RxDS Header Delimiter0* register to *RxDS Header Delimiter1* register.
Reserved	30:24	Read as zero.
Change Index	23:16	**Change Index** — Index into segment for match comparison Byte.
Change Value	15:8	**Change Value** — Value to match against.
Change Mask	7:0	**Change Mask** — Mask to apply to match comparison Byte.

RxDS_Header_Change2 Register (FP Rx Configuration Function)

Purpose Same as RxDS Header Change Register 1 above.

Address 0xBDE0460C

RxDS_Header/Payload_Delimiter0 Register (FP Rx Configuration Function)

Purpose Used by the Data Separator hardware to separate the header from the payload.

Address 0xBDE04610

Access Global Read/Write

Bit Position	31 24	23 16	15 8	7 0
Field Name	Reserved	Header Last Index	Payload First Index	Payload Last Index
Reset Value	raz	0	0	0

Field Name	Bit Position	Description
Reserved	31:24	Read as zero.
Header Last Index	23:16	**Header Last Index** — Byte position offset from 0 Byte (first Byte) of cell identifying the last Byte of the header. Must be less than or equal to (cell_size_5).
Payload First Index	15:8	**Payload First Index** — Byte position offset from 0 Byte (first Byte) of cell identifying the first Byte of the payload.
Payload Last Index	7:0	**Payload Last Index** — Byte position offset from 0 Byte (first Byte) of cell identifying the last Byte of the payload. Must be greater than 3. NOTE: The Payload Last Index typically equals (cell_size_1), using the Cell Size value programmed into the *RxFI* configuration register. See RxFI_Configuration Register (FP Rx Configuration Function). NOTE: There must be at least a 32Byte difference between Payload First Index and Payload Last Index.

RxDS Header/Payload Delimiter1 Register (FP Rx Configuration Function)

Purpose Same as *RxDSHeader/Payload Delimiter0* register, except for Payload Delimiter1.

Address 0xBDE04614

RxDS_Header/Payload_Delimiter2 Register (FP Rx Configuration Function)

Purpose Same as *RxDSHeader/Payload Delimiter0* register, except for Payload Delimiter2.

Address 0xBDE04618

RxDS_Configuration Register (FP Rx Configuration Function)

Purpose Configures the Data Separator Hardware.

Address 0xBDE0461C

Access Global Read/Write

Bit Position	31	30	29 23	22 20	19	18	17	16	15 8	7 0
Field Name	QUE GNT DIS	SM GNT DIS	Rsvd	CTL Word Size	CTL Word Disable	Rsvd	DROP HDR	Rx Byte EOH	Data XOFF Threshold	Data XON Threshold
Reset Value	0	0	raz	0		raz		0	32	32

Field Name	Bit Position	Description
QUE_GNT_DIS	31	**Queue Grant Disable, PRIZMA mode only** — Must be set to 1. 0 enables hardware interpretation of Queue Grants to link level flow control Tx side of C-5 NP. 1 disables hardware so that RxByte microcode can invoke per-flow flow control. When enabled, link level flow control is asserted whenever any of the Queue Grants are not enabled.
SM_GNT_DIS	30	**Shared Memory Grant Disable, PRIZMA mode only** — Must be set to 0. A 0 value enables hardware interpretation of Shared Memory Pool Grants to link level flow control Tx side of C-5 NP. 1 disables hardware so that RxByte microcode can invoke per-flow flow control. When this bit is zero, link level flow control is asserted whenever any of the Shared Memory Grants are not asserted.
Reserved	29:23, 18	Read as zero.
CTL Word Size	22:20	**Control Word Size**, **PowerX only** — Must be set to 2. Indicates the size of control words for fabrics which support them. For instance, 0=disabled, 010b=2 bytes, 100b=4 bytes. Control words are directed to the appropriate Byte processor between cells. Between cells, a Byte processor needs to test a control word condition prior to each cell being processed. The Byte processor needs to handle all control words prior to handling the next cell.
CTL Word Disable	19	**Control Word Disable** — Must be set to 0 for PowerX and 1 for all other modes. 1 deselects the Control word FIFO operation; 0 enables CTL word FIFO operation. By deselecting the CTL word FIFO, control words will be ignored by the Data Splitter logic and not presented to the Byte processor.

Field Name	Bit Position	Description
DROP_HDR	17	**Drop Header** — Must be set to 0. 1 selects header to be dropped when payload is being dropped due to congestion; 0 selects header to be forwarded to Byte processors even when payload is being dropped.
Rx Byte processor EOH	16	**Receive Byte Processor End of Header Indication** — Reserved. Must be set to 0. 1 selects RxByte Byte processor Data9 test condition to be set true coincident with the last Byte of the header (PDU + SFR) (Refer to *SDP Programmer Guide*). 0 selects RxByte Data9 test condition to be set true coincident with the first Byte of the header.
Data XOFF Threshold	15:8	**Payload FIFO Flow Control threshold for XOFF** — Number of 32bit words left in the payload input FIFO when flow control is asserted. Valid Range 0-125. NOTE: The XOFF threshold must be less than the XON threshold.
Data XON Threshold	7:0	**Payload FIFO Flow Control threshold for XON** — Number of 32bit words -1 left in the payload input FIFO when flow control is deasserted. Valid Range: 1-126. NOTE: The XON should always be greater than the Data XOFF Threshold. If Data XON Threshold is less than Data XOFF, unpredictable results will occur.

RxFI_CRC Register (FP Rx Configuration Function)

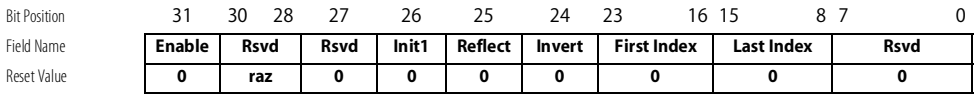
Purpose — The RxFI CRC register configures the CRC function.

- Initial Value of the FP accumulator (0 or all 1s)
- Inverted (ones complement)

The RxFI CRC provides INDEX fields which allow the CRC to be calculated over any sequential portion of the Segment and then appended or inserted anywhere afterward.

Address — 0xBDE04620

Access — Global Read/Write



Bit Position	31	30 28	27	26	25	24	23 16	15 8	7 0
Field Name	Enable	Rsvd	Rsvd	Init1	Reflect	Invert	First Index	Last Index	Rsvd
Reset Value	0	raz	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Enable	31	**CRC Enable** — 1 enables CRC mechanism; 0 disables and leaves the CRC mechanism in reset.
Reserved	30:28	Read/write
Reserved	27	Must be set to 1
Init1	26	**CRC Initialization** — 1 initializes the CRC register to all 1s; 0 initializes it to a 0.
Reflect	25	Reserved. Must be set to 0.
Invert	24	**Invert CRC** — 1 selects CRC to be inverted prior to being appended to Segment; 0 selects not inverted.
First Index	23:16	**First Index** — Offset from byte 0 of segment to start of CRC accumulation byte. Index must be a multiple of 4.
Last Index	15:8	**Last Index** — Must be set to (cell size - 1). Represents the offset from byte 0 of segment to the last byte of the CRC value itself (not the last byte of the CRC accumulation region). Must be equal to or less than (cell size -1), even if CRC is not enabled.
Reserved	7:0	**Append Index** — Unused. Byte offset from byte 0 of segment to appended CRC.

RxWCS_CAM Register (RxWCS_CAM Function)

Purpose Interface in Global address space to initialize the FP RxByte processor WCSs and CAMs along with the DBE WCS.

Address 0xBDE04624

Access Global Read / Write

Bit Position	31	24	23	22	21	20	19	18	17	16
Field Name	DBE DATA IN		DBE Scan Out	Rsvd	DBE Write	Rsvd	DBE Scan Capture	DBE Scan Update	DBE Scan1 In	DBE Scan0 In
Reset Value	0		x	raz	0	raz	0	0	0	0
Bit Position	15	8	7	6	5	4	3	2	1	0
Field Name	WCS DATA IN		WCS Scan1 Out	WCS Scan0 Out	WCS Write	Rsvd	WCS /CAM Scan Capture	WCS /CAM Scan Update	WCS /CAM Scan1 In	WCS /CAM Scan0 In
Reset Value	0		x	x	0	raz	0	0	0	0

Field Name	Bit Position	Description
DBE Data in	31:24	**WCS_CAM Data in** — DBE WCS Byte wide data

Field Name	Bit Position	Description
DBE Scan Out	23	**WCS_CAM Scan Out** — Output of DBE Scan Chain (Read Only).
DBE Write	21	**WCS_CAM Write** — DBE WCS Byte Write (1 Selects Write)
DBE Scan Capture	19	**WCS_CAM Scan Capture** — Capture data from selected address field from the DBE WCS to the DBE Scan Chain
DBE Scan Update	18	**WCS_CAM Scan Update** —S tore or Update the DBE WCS location as defined on the DBE Addr bits with the 52 bits of data in the DBE WCS.
DBE Scan1 In	17	**WCS_CAM Scan1 In** — Shift a 1 into the DBE Scan Chain.
DBE Scan0 in	16	**WCS_CAM Scan0 in**— Shift a 0 into the DBE Scan Chain.
Rsvd	22, 20, 4	Reserved, Read as Zero.
WCS Data in	15:8	**WCS_CAM Data in** — WCS Byte wide data
WCS Scan1 Out	7	**WCS_CAM Scan1 Out** — Output of WCS1 Scan Chain (Read Only).
WCS Scan0 Out	6	**WCS_CAM Scan0 Out** — Output of WCS0 Scan Chain (Read Only).
WCS Write	5	**WCS_CAM Write** — WCS Byte Write (1-Selects Write)
WCS/CAM Scan Capture	3	**WCS_CAM Scan Capture** — Capture data from selected address fields from WCS/CAM to WCS/CAM Scan Chain.
WCS/CAM Scan Update	2	**WCS_CAM Scan Update** — Store or Update the CAM entry as defined on the CAM Addr bits with the 36 bits of data in the CAM Group, CAM Pattern, and CAM Tag.
WCS/CAM Scan1 In	1	**WCS_CAM Scan1 In** — Shift a 1 into the Scan Chain.
WCS/CAM Scan0 in	0	**WCS_CAM Scan0 in** —Shift a 0 into the Scan Chain.

RxByte0 General Purpose Configuration Register (FP Rx Configuration Function)

Purpose The *FDP RxByte0 General Purpose Configuration* Register (PCR) provides an area for passing information between the XP and the FDP RxByte processor0.

Global Address 0xBDE04628

FDP Address 0xA0 – 0xA3

Access Global bus Read/Write, RxByte Read only – Byte addressable

Bit Position	31 … 0
Field Name	Data
Reset Value	0

RxByte1 General Purpose Configuration Register (FP Rx Configuration Function)

Purpose Same as the *FDP RxByte0 General Purpose Configuration* register except for RxByte Processor1.

Global Address 0xBDE0462C

FDP Address 0xA0 – 0xA3

RxFCE_Configuration0 Register (FP Rx Configuration Function)

The RxFCE contains configuration registers that reside in global address space and are allocated to the FP block. These registers enable configuration of descriptors and Buffer Pools as defined below.

Purpose Configures RxFCE descriptor size, Ring Bus response size, and flow mask.

Address 0xBDE04630

Access Global Read/Write,

Bit Position	31 … 26	25 … 24	23	22 … 18	17 … 16	15 … 0
Field Name	Reserved	Desc Size	TLU Resp	Reserved	Resp Size	PDU ID Mask
Reset Value	raz	10	0	raz	10	0xFFFF

Field Name	Bit Position	Description
Reserved	31:26	Read as zero.

<table>
<tr><th>Field Name</th><th>Bit Position</th><th>Description</th></tr>
<tr><td>Desc Size</td><td>25:24</td><td>QMU Descriptor Size — Selects the Descriptor Size and implicitly selects the extract space to 16 Byte x 16 scopes (Desc Size = 12 or 16), or 32 Byte x 8 scopes (Desc Size = 24 or 32).

<table><tr><th>Encoded Value</th><th>Size (Bytes)</th></tr><tr><td>00</td><td>32</td></tr><tr><td>01</td><td>12</td></tr><tr><td>10</td><td>16</td></tr><tr><td>11</td><td>24</td></tr></table></td></tr>
<tr><td>TLU Resp</td><td>23</td><td>Enable TLU Response — Force DBE to wait until TLU response is complete before beginning to build a descriptor.
1= enabled
0 = disabled</td></tr>
<tr><td>Reserved</td><td>22:18</td><td>Read as zero.</td></tr>
<tr><td>Resp Size</td><td>17:16</td><td>Response Size — Selects the TLU Response Size. For 16 Byte x 16 scopes, Resp Size = 16. For 32 Byte x 8 scopes, Resp Size = 16 or 32.

<table><tr><th>Encoded Value</th><th>Size (Bytes)</th></tr><tr><td>00</td><td>32</td></tr><tr><td>01</td><td>8</td></tr><tr><td>10</td><td>16</td></tr><tr><td>11</td><td>reserved</td></tr></table></td></tr>
<tr><td>PDU ID Mask</td><td>15:0</td><td>PDU ID Mask — Used to mask bits of PDU ID.
NOTE: This could be used in lieu of microcode to automatically mask a PDU ID of less than 16bits.</td></tr>
</table>

RxFCE_Configuration1 Register (FP Rx Configuration Function)

Purpose Configures the RxFCE.

Address 0xBDE04634

Access Global Write Only

Bit Position	31 16	15	14	13 10	9 8	7 0
Field Name	PDU Size	Default Size Enable	PDU LEN CK DIS	Rsvd	Drop on Flow	Drop on BTag	Allocation Delay
Reset Value	0	raz	0	raz	0	0	0

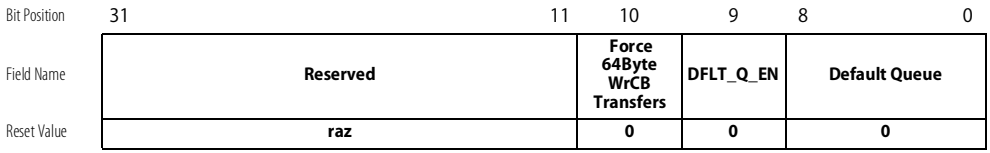
Field Name	Bit Position	Description
PDU_Size	31:16	**Default PDU Size** — Used in place of the *RxFlowSz* register if bit 10 (*DFLT_SZ_EN*) is selected.
DFLT_SZ_ EN	15	**Default Size Enable** — 1 selects default size; 0 requires RxByte processor code to fill out PDU length (usually obtained from first PDU segment header).
PDU_LEN_CK_DIS	14	**PDU Length Check Disable** — 1 disables PDU length check; 0 enables PDU length check. Used with protocols that do not supply a length field in the header of the PDU. Here the RxByte processor selects a PDU length to ensure a large enough buffer. With this bit enabled, a final PDU length check will not be performed, allowing the buffer to be enqueued without error.
Reserved	13:10	Read as zero.
Drop on Flow	9	**Drop on Flow** — This bit must be set to 1. 1 selects drop segment if a Flow Table CAM entry is not available; 0 selects do not drop flow and wait for a Flow Table entry to become available.
Drop on BTag	8	**Drop on BTag** — This bit should be set to 1. 1 selects drop segment if BTag is not available; 0 selects wait for BTag to become available. Leaving this set to 0 presumes that under a worst case scenario, the BTag will become available prior to the first segment fully arriving.
Allocation Delay	7:0	**Allocation Delay** — This bit must be set to 0. Back-off time delay between retries when the FP Rx Buffer Pool Manager is replenishing BTags. The default is 0. Can change using (number of core clocks 8), but not recommended.

RxFCE_Configuration2 Register (FP Rx Configuration Function)

Purpose Configures the FCE.

Address 0xBDE04638

Access Global Write Only (Global Reads return inaccurate data).



Field Name	Bit Position	Description
Reserved	31:11	Read as Zero
Force 64Byte WrCB Transfers	10	**Force 64Byte WrCB Transfers —** When asserted, the FP_Rx write control blocks (WrCB's) always performs a 64Byte DMA transfer to SDRAM. Otherwise, DMA transfers are in 16Byte increments (16, 32, 48 or 64).
DFLT_Q_EN	9	**Default Queue Enable —** 1 selects the use of the Default Queue (see *Default Queue* field) in the event of a TLU error; 0 disables the use of the default queue. NOTE: If both the *DFLT_Q_EN* bit is set *and* DBE code writes the TLU_ERROR to the Drop Packet Bit, the *DFLT_Q_EN* takes precedence and the packet is enqueued to the default queue.
Default Queue	8:0	**Default Queue Number** — 9bit queue number to be used if *DFLT_Q_EN* bit set *and* a TLU_ERROR occurs.

Buffer Pools

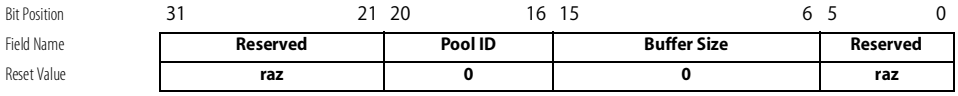
Associated with each of the four buffer pools are two configuration registers *PooIn_CFG0* and *PooIn_CFG1*, where n=0,1,2,3.

Pool0_CFG0 Register (FP Rx Pool Configuration Function)

Purpose Configures FCE Buffer Pool0 ID and buffer size parameters. See Table 163 on page 563 for similar registers.

Address 0xBDE04640

Access Global Read/Write



Bit Position	31 21	20 16	15 6	5 0
Field Name	Reserved	Pool ID	Buffer Size	Reserved
Reset Value	raz	0	0	raz

Field Name	Bit Position	Description
Reserved	31:21	Read as zero.
Pool Id	20:16	**Pool ID** — Maps to BMU Pool.
Buffer Size	15:6	**Buffer Size** — Must match BMU buffer size (between 64 and 64Kbytes) for specified pool ID. The sizes are in terms of 64 Byte blocks. Buffer Size[15:6]=1 corresponds to a size of 64. Buffer Size[15:6]=0b1111111111 corresponds to a size of 64K-64. NOTE: A size of 0 is not valid; Buffer Size[15:6]=0 corresponds to a size of 64K.
Reserved	5:0	Read as zero.

Table 163 PooIn_CFG0 Registers (for Pools 1, 2, and 3)

Address	Register Name	Purpose	Counter Width (Bits)
0xBDE04648	Pool1_CFG0	Same as *Pool0_CFG0* reg, except for pool 1.	32
0xBDE04650	Pool2_CFG0	Same as *Pool0_CFG0* reg, except for pool 2.	32
0xBDE04658	Pool3_CFG0	Same as *Pool0_CFG0* reg, except for pool 3.	32

Pool0_CFG1 Register (FP Rx Pool Configuration Function)

Purpose Configures FCE Buffer Pool0, backup, allocation threshold, and last block parameters. See Table 164 on page 564 for similar registers.

Address 0xBDE04644

Access Global Read/Write

Bit Position	31 26	25 24	23 11	10 8	7 3	2 0
Field Name	Reserved	Backup	Reserved	Alloc Threshold	Rsvd	Last Block
Reset Value	raz	n	raz	0	raz	0

Field Name	Bit Position	Description
Reserved	31:26	Read as zero.
Backup	25:24	**Backup Pool** — Identifies the backup pool (0-3). If the backup pool number equals itself, then there is no further backup. Default to *n*, where *n* is the Pool*n*_CFG1 register. For example, backup pool 0 = 0.
Reserved	23:11	Read as zero.
Alloc Threshold	10:8	**Allocation Threshold** — Specifies a threshold value such that when the number of blocks drops below that threshold, additional BTags are fetched until the 'Last Block' level is reached. Rules: Allocation Threshold must *ALWAYS* be ≤ Last Block. Last Block = 0 disables pool refill.
Reserved	7:3	Read as zero.
Last Block	2:0	**Last Block of BTags** — Specifies the blocks (0-7) that are to be filled. Specifying a Last block of 0 disables a given pool. Therefore the useful range is 1 to 7, where a value of 1 with an Allocation Threshold = 1 fills two blocks (0 and 1).

Table 164 PooIn_CFG1 Registers (for Pools 1, 2 and 3)

Register Name	Purpose	Address
Pool1_CFG1	Same as *Pool0_CFG1* reg, except for pool 1.	0xBDE0464C
Pool2_CFG1	Same as *Pool0_CFG1* reg, except for pool 2.	0xBDE04654
Pool3_CFG1	Same as *Pool0_CFG1* reg, except for pool 3.	0xBDE0465C

FDP_RxByte_Shared0 Register (FP Rx Shared Function)

The FDP provides two, 4Byte shared general purpose registers (*FDP RxByte Shared0* and *FDP RxByte Shared1*) that are accessible by both the FDP's RxByte0 and RxByte1 Byte processors (see page 559). The registers are Byte writable from each of the Byte processors.

The use of the *FDP RxByte Shared0* and *FDP RxByte Shared1* registers are application specific. For example, they could be used by the FDP microcode to pass information between the RxByte0 and RxByte1 Byte processors.

One Byte of these registers could serve as a read/modified write, (where the value of the read/modified write is passed to each Byte processor from the FDP GPR), if needed by the application.

Purpose	Provides four Bytes for passing information between the FDP's RxByte0 and RxByte1 processors.
Global Address	0xBDE04660
FDP Address	0xB0 – 0xB3
Access	XP Read/Write FDP Receive Byte processor Read/Write – Byte addressable

Bit Position	31	0
Field Name	Data	
Reset Value	0	

FDP_RxByte_Shared1 Register (FP Rx Shared Function)

Purpose	Same as the *FDP RxByte Shared0* register except that it provides an additional four Bytes for passing information between the FDP's RxByte0 and the RxByte1 processors.
Global Address	0xBDE04664
FDP Address	0xB4 – 0xB7

RxFP_Interrupt_Event Register (FP Rx Interrupt Function)

Purpose Access to interrupt events (interrupts directed to the XP via the *RxFP Interrupt Event Mask* register).

Address 0xBDE04680

Access Global Read/Write, Write 1 to Clear

Bit Position	31 … 7	6	5	4	3	2	1	0
Field Name	Reserved	Error FIFO Full	Parity Error	No BTags on Alloc	WR FAIL	BTag PRG	BTag ECC	Alloc Timeout
Reset Value	raz	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Reserved	31:7	Read as zero.
Error FIFO Full	6	**Error FIFO Full** — Indicates all 32 entries of the error FIFO stack are full.
Parity Error	5	**Parity Error** — Indicates a parity error was detected on the C-Port Fabric interface.
No BTags on Allocation	4	**No BTags on Allocation** — BTag allocation failed because of no BTags available from the BMU.
WR FAIL	3	**Payload Fail** — Indicates a Payload Bus parity error during a Payload Write via the WrCB.
BTag PRG	2	**BTag PRG** — Indicates either bad pool request (code 0x2) or a non-existent memory location (code 0x6) was attempted by the RdCB. These result from an inconsistent programming of the FP Rx Pools relative to the FP Rx BTag.
BTag ECC	1	**BTag ECC** — Indicates a double ECC error was returned as a result of an attempt by the RxCB to transfer BTags from the BMU. A code of 0x5 is returned by the RdCB.
Allocation Timeout	0	**Allocation Time-out**— Indicates the number of BTag allocation retries exceeded, (16 max.)

RxFP_Interrupt_Enable Register (FP Rx Interrupt Function)

Purpose Enable interrupts for the corresponding bits in the *RxFP Interrupt Event* register. Set a bit to 1 to enable the interrupt.

Address 0xBDE04684

Access Global Read/ Write

Bit Position	31 … 7	6	5	4	3	2	1	0
Field Name	Reserved	Error FIFO Full EN	Parity Error EN	No BTags on Alloc EN	WR FAIL EN	BTag PRG EN	BTag ECC EN	Alloc Timeout EN
Reset Value	raz	0	0	0	0	0	0	0

RxFP_Debug_Event_Mux_Control (FP Rx DeBug Function)

Four of 13 events can be viewed via the *RxDebug Event Mux Control* register. The selectable events are shown in Table 165. Many of these events relate to FIFO full conditions which, if the FP Rx is programmed correctly, should never occur.

Any event can be viewed in association with any of the four selection fields, including simultaneously being selected in more than one field, that is, viewed multiple times.

Purpose For the purposes of diagnostics and debug. Enables key test points to the system event register to be viewed.

Address 0xBDE04688

Access FDP Read/Write, Global Read

Bit Position	31	30 … 28	27 … 24	23	22 … 20	19 … 16	15	14 … 12	11 … 8	7	6 … 4	3 … 0
Field Name	EN0	RSVD	SEL0	EN1	RSVD	SEL1	EN2	RSVD	SEL2	EN3	RSVD	SEL3
Reset Value	0	0	0	0	0	0	0	0	0	0	0	0

Field Name	Bit Position	Description
EN0	31	**Enable0** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	30:28	Read as zero.
SEL0	27:24	**Select0** — Selects one of the thirteen FP Rx events to be viewed for the corresponding field. See Table 165 on page 568.
EN1	23	**Enable1** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	22:20	Read as zero.

Field Name	Bit Position	Description
SEL1	19:16	**Select1** — Selects one of the thirteen FP Rx events to be viewed for the corresponding field. See Table 165 on page 568.
EN2	15	**Enable2** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	14:12	Read as zero.
SEL2	11:8	**Select2** — Selects one of the thirteen FP Rx events to be viewed for the corresponding field. See Table 165 on page 568.
EN3	7	**Enable3** — 1 enables the associated selected events; 0 disables the associated event from being viewed.
Reserved	6:4	Read as zero.
SEL3	3:0	**Select3** — Selects one of the thirteen RxFP events to be viewed for the corresponding field. See Table 165 on page 568.

Table 165 RxFP Events

Select Value	Event Name	Description
0	RX_NEXT_SEG	Indicates that internal hardware is requesting the next segment for processing. This event should occur normally as a result of correct FP operation for each segment received independent of whether an error is detected in the segment or not.
1	RX_FLOW	Indicates a good Queue status has been received from the QMU as a result of either an enqueue or dequeue operation. This event normally occurs as a result of correct operation of the RxFP for non-errored PDUs.
2	RX_ERR	Indicates one or more of the errors (as defined in the error statistics table has occurred. This event occurs operationally as a result of the non-fatal interface type errors.
3	FLOW_ERR	Indicates a valid buffer deallocation has been made as a result of a flow error, that is, both the buffer deallocation signal is high AND the tx_bde request FIFO is not full. Flow errors occur during operation anytime an error is detected in a flow after the first segment has been successfully received, that is, resources allocated for the flow and a subsequent segment has been detected to be in error.
4	BPM_ERR_ALLOC	Indicates Buffer Pool manager Allocation Error sourced from the BMU. This can occur as a result of either a Minor NAK or a lack of BTags for a requested pool cache update.

Table 165 RxFP Events (continued)

Select Value	Event Name	Description
5	AFIFO_FULL	Indicates Async Fabric Interface FIFO is full. This should never occur under any circumstances and would indicate a hardware failure. This FIFO is 'deallocated' upon the condition that it is not empty, that is, data is moved out on the following clock cycle. Data is dropped at the synchronous payload FIFO under congested operation.
6	HFIFO_FULL	Indicates the header processing FIFO is full. This FIFO is filled by the Data Splitter with the selected header Bytes as configured in the header/payload Delimiter registers. If the header/payload Delimiter registers and RxByte Processors are programmed correctly this FIFO should never overflow. If this FIFO does overflow, then the FP Rx must be reset to re-sync header and payload.
7	EFIFO_FULL	Indicates an internal FIFO which passes detected interface errors, for example, Parity or FP, between Fabric Interface logic and internal hardware is full. This condition should never occur.
8	PFIFO_PAUSE	Indicates the pause (XOFF) threshold has been exceeded. This typically occurs during a congested or flow-controlled situation.
9	FLOW_FULL	Either of two Internal 'Flow FIFOs' are full. Each of the Rx Byte processors, upon selecting scope 'Avail', push the current flow forward into the flow handling hardware of the FP Rx. A 'Flow Full' event indicates that one of these FIFOs is full and in effect the flow handling hardware is stalled for some reason. Typically this is due to Internal hardware pending upon lack of some resource such as BTags (although DROP_ON_BTAG should be selected to prevent this). This event should never occur.
10	DBE_FULL	An Internal FIFO which passes the requests for the DBE to Build a Descriptor for a given flow is full. This event should never occur. This type of an event could occur if the DBE or TLU is improperly programmed so that the DBE is stalled, for example, waiting for a TLU operation to complete.
11	ENQ_FULL	FP Rx Enqueue Request FIFO Full. This event should never occur. This would indicate that FP Rx internal hardware has completed reassembly of a PDU and is enqueuing requests faster than the enqueue hardware / QMU can accept them. NOTE: This error could be a result of a DBE / TLU programming error such that the Q_NUM_VALID bit has not been set by the DBE.

Table 165 RxFP Events (continued)

Select Value	Event Name	Description
12	DEQ_FULL	FP Rx Dequeue Request FIFO Full. This event should never occur. This would indicate that FP Tx is requesting Dequeue operations faster than Queue Request hardware can process them.

RxMemory_Address Register (FP Rx DeBug Function)

Purpose Configures target memory address.

Address 0xBDE04690

Access Global Read/Write

Bit Position	31	0
Field Name	Address	
Reset Value	0	

RxMemory_Data Register (FP Rx DeBug Function)

Purpose Used to write or read target Flow Table and Descriptor Memory locations.

Address 0xBDE04694

Access Global Read/Write

Bit Position	31	0
Field Name	Data	
Reset Value	0	

RxPDU_ID_CAM Register (FP Rx DeBug Function)

Purpose Provides access to the FP Rx PDU_ID CAM for debug purposes.

The PDU_ID CAM is a 160-entry CAM in the FP Rx that maps a 16bit PDU ID to an 8bit internal PDU index used by hardware. The CAM can be accessed by software only for debug purposes.

Address 0xBDE04698

Access Global Read/Write. The CAM hardware updates the Free bit continuously. The hardware updates the CAM data field after a Search operation.

Bit Position	31 28	27	26	25	24	23 8	7 0
Field Name	Reserved	Free	Write	Delete	Search	Match	CAM Data
Reset Value	raz	0	raz	raz	0	0	0

Field Name	Bit Position	Description
Reserved	31:28	Read as zero.
Free	27	**Free** - Indicates that at least one entry in the CAM is free (available for use). This bit is read-only.
Write	26	**Write CAM Location** — Writes the location matched, or the next free location if nothing matches (for diagnostic purposes only). Setting this bit to a 1 launches a CAM write of the write data. The bit will then be cleared by the hardware. It will always be read as zero.
Delete	25	**Delete CAM Entry** — Deletes CAM entry matched (for diagnostic purposes only). Setting this bit to a 1 launches a CAM delete of the entry corresponding to the match value. The bit will then be cleared by the hardware. It will always be read as zero. **Encoded Value** / **CAM Action** 1 — Delete Cam entry 0 — Do not delete Cam entry
Search	24	**Search** — 1 selects CAM search, using the match value. After the search, the result is returned via the CAM data field in this register.
Match	23:8	**16bit CAM Match Value** — Value to search on.
CAM Data	7:0	**CAM Data -** Contains the 8bit value that was read out of the CAM after the most recent Search operation. Alternatively, this represents the data that is written into the CAM on a Write operation.

RxFP_Statistics Registers (FP Rx Statistics Function)

The FP Rx provides nineteen 32bit registers that accumulate statistics. These registers are only accessible via the global bus. Table 166 defines these registers and provides their Global Bus address. The RxFP Statistics Registers are read in 32bit quantities. For counters that are only 16bits, the upper 16bits of the register are read as zero (raz) and the lower 16bits hold the contents of the requested counter. A global write to one of these registers, regardless of the write data, will reset the counter to 0.

When a counter reaches its maximum value, it rolls over to 0 and starts counting again.

Table 166 Global Bus Receive FP Statistics Registers Map

Address	Name	Description	Counter Width (Bits)
0xBDE046A0	SEGS_RCVD	Number of segments received.	32
0xBDE046A4	PDUS_RCVD	Number of PDUs received.	32
0xBDE046A8	SEGS_LOST	Number of segments lost.	32
0xBDE046AC	PDUS_LOST	Number of PDUs lost for any reason.	32
0xBDE046C0	CPARITY_ERR	Number of Control Word Parity Errors (PowerX only).	16
0xBDE046C4	ERR_HDR	Number of Errored headers (HDR FIFO overrun – Fatal Error).	16
0xBDE046C8	PARITY_ERR	Number of received parity errors.	16
0xBDE046CC	LENGTH_ERR	Number of segments length errors.	16
0xBDE046D0	Reserved	Reserved. Read as zero.	16
0xBDE046D4	CRC_ERR	Number of CRC errors.	16
0xBDE046D8	ODD_PDU	Number of odd PDUs (middle or last Segment with no CAM entry).	16
0xBDE046DC	SEQ_ERR	Number of sequencer errors (set by RxByte processor).	16
0xBDE046E0	SEQ_DIS	Number of sequencer discards (set by RxByte processor).	16
0xBDE046E4	LOST_PDU	Number of PDUs lost because of missing cells (PDU length error).	16
0xBDE046E8	NO_FLOW_TBL	Number of times no Flow Table entry available (all 160 flows are in use).	16
0xBDE046EC	NO_BTAG	Number of times no BTag available from the Pool Cache in the FP Rx.	16

Table 166 Global Bus Receive FP Statistics Registers Map (continued)

Address	Name	Description	Counter Width (Bits)
0xBDE046F0	BTAG_ERR	Bits [23:16] represent the number of BTag Programming Errors, for example, because of a bad Pool ID. Bits [7:0] represent the number of BTag ECC errors. The remaining bits are unused.	32 (contains 2 8bit counters)
0xBDE046F4	ALLOC_ERR	Number of BTag allocation errors, due to lack of available BTags in the BMU.	16
0xBDE046F8	ENQUE_ERR	Number of enqueue errors (QMU responded with a NAK, i.e. due to lack of descriptors).	16

RxDebug_Internal_State Register (FP Rx Statistics Function)

Purpose Enables viewing key internal states including: SBP0/1 Program counter, Descriptor Build Engine Program Counter, Internal Buffer (BFR) State-machine states, Internal DMA Transfer ('XCB') states, and internal Enqueue Engine states.

Address 0xBDE04700

Access Global Read Only

Bit Position	31	30 25	24 19	18 14	13 8	7 6	5 0
Field Name	Rsvd	SBP0	SBP1	BFR STATE	DBE PC	XCB	ENQ STATE
Reset Value	0	0	0	0	0	0	0

Field Name	Bit Position	Description
Reserved	31	Read a zero.
SBP0	30:25	**Serial Byte Processor Program Counter0** — Program Counter0 for the Serial Byte Processor [5:0], bit 6 is not available.
SBP1	24:19	**Serial Byte Processor Program Counter1** — Program Counter1 for the Serial Byte Processor [5:0], bit 6 is not available.
BFR STATE	18:14	**Buffer State Machine States** — See Table 169 on page 575.
DBE PC	13:8	**Descriptor Build Engine Program Counter** — Program counter for the Descriptor Build Engine.
XCB	7:6	**Transfer Control Block Programming States** — See Table 168 on page 574.
ENQ STATE	5:0	**Enqueue State Machine States** — See Table 167 on page 574.

Each state machine runs off the core clock and requires at least one clock cycle to transition to the exit state. States are either *conditional*, that is, waiting for one or more conditions to become true so they can exit to an alternate state, or they are *transitional* whereby they will always exit to an alternate state in a single clock cycle.

While transitional states may have conditions that allow them to transition to one or more possible states, if no condition is true, a default exit state ensures that they cannot remain in the current state. The importance of this distinction is that when asynchronously polling the *RxDebug Internal State* register, it is important to realize that by chance you may hit upon transitional states that indicate that the machine is operating and that a cell is being processed by the FP Rx. Consistently seeing a conditional state indicates that the machine is most likely waiting for a condition to be fulfilled. This can help you identify a related illegal configuration.

Whether a state is conditional (C) or transitional (T) is called out in the following Machine State tables.

Table 167 Enqueue QMU Programing Machine States

State Number	State Type	State Name	Description / Exit Condition
0	C	IDLE	Idle state awaiting BFR Enqueue request
1	T	PEND	Pend State provided for timing purposes
10	T	PRG_CTRL	Program Control Info to QMU such as Weight, MCAST etc.
18 - 11	T	ENQ8-ENQ1	Program Words 8 – 1. NOTE: For 32Byte descriptors states ENQ8-1 are transitioned, 24Byte descriptors ENQ6-1, 16Byte descriptors ENQQ4-1, 12 Byte descriptors ENQ3-1. Once an ENQ state is entered they cycle down to ENQ1, then PEND, then back to idle, all one clock per state.

Table 168 Transfer Control Block Programing States

State Number	State Type	State Name	Description / Exit Condition
0	C	IDLE/ PRG_SYS	IDLE State awaiting BFR Payload DMEM flush to DRAM. Upon which Write CB System Register is programmed as state is exited
1	T	PRG_DMA	Write CB DMA Register being programmed
2	T	PRG_CTL	Write CB Control Register being programmed

Table 169 Buffer Engine State Machine States

State Number	State Type	State Name	Description / Exit Condition
0	C	IDLE	Idle, waiting for Valid Segment Indication from SBP0 and 1 in alternating order
1	T	CAM_WAIT	Flow ID (FID) CAM lookup delay state
2	C	SEARCH	Match FID to produce Flow Index (FIN), Determines Segment type. Requires available FID entry OR Drop on Flow bit enabled (Bit9 selected to 1 in the RxFCE Config1 Register).
3	T	RD_TABLE	FIN is used to index into Flow State Table
4	C	GET_BTAG	For beginning of messages (firsts), state to get a BTag. Technically this state is Transitional, however if no BTags exist, the BFR will alternate between states 2 and 4 until BTags with correctly sized buffers become available.
5	C	XFR_DATA	Transfer Data from Payload FIFO to 64Byte DMEM Buffer. XFR DMEM to DRAM every 64Byte boundary. Stays in this state until End of Segment (EOS) or End of Packet (EOP).
6	C	DROP_DATA	Discard state for pad Bytes of last segment or discard due to Segment/Flow error. Exit upon EOS.
7	C	ENQUEUE	Enqueue Assessment state. Waits if Enqueue and no room in Enqueue Request FIFO.
8	T	ERROR	Delay state to wait for worst case parity / FP errors
9	C	NEW_BFR	State to get a new 64Byte DMEM buffer as XFR to buffer DRAM BTag. Wait in this state until a new DMEM buffer is available.
10	C	PEND_BFR	Special case where end of segment and 64 byte DMEM buffer are full. Need to get a new buffer before updating state table. Wait in this state until new DMEM buffer is available.
11	T	RTN_PEND	Special delay state to allow DMEM buffer to be returned during transfer if an error detected.

Appendix B

Using Aggregate Mode

Appendix Overview

This appendix covers the following topics:

- Purpose of the C-5 NP Channel Aggregate Mode
- Aggregate Mode Requirements on the C-5 NP
- Packet/Cell Ordering Handling for Rx in Aggregate Mode
- Packet/Cell Ordering Handling for Tx in Aggregate Mode
- Clock Distribution in Aggregate Mode
- Aggregate Mode Application Examples

Purpose of the C-5 NP Channel Aggregate Mode

The C-5 NP Aggregate Mode enables you to scale serial bandwidths. The CPs can be aggregated into parallel clusters for wider data streams. The C-5 NP's 16 CPs can be partitioned into four (4) groups of four (4) CPs called *clusters*. Clusters allow the CPs to share resources (IMEM and DMEM) and support aggregation. A cluster of CPs can be configured, for example, to work together to support one physical interface (such as OC-12), or either the receive or transmit portion of one physical interface (such as Gigabit Ethernet).

The types of physical interfaces that require aggregation in the C-5 NP include: Gigabit Ethernet, FibreChannel, OC-12 and OC-12c.

Aggregate Mode Requirements on the C-5 NP

Aggregation requires that the C-5 NP fulfill two (2) requirements for processing data from high-speed physical interface. These are needed to ensure that the C-5 NP is able to keep up with the speed of these high-bandwidth physical interfaces, as well as effectively use the processing power of the C-5 NP; the following requirements must be met:

1 Individual packets and cells must be distributed among the CPs in a cluster to effectively share the processing load.

2 Packet and cell ordering between the ingress and egress physical interfaces must be preserved.

Supporting these two (2) requirements have the following implications for the Serial Data Processor (SDP) and Channel Processor RISC Core (CPRC) components as described in Table 170 on page 578.

Table 170 Aggregate Mode Implications (for SDP and CPRC)

Component	Implication
RxSDP	The RxSDP must distribute incoming packets and cells to the different CPs within a cluster in a round robin fashion.
CPRC	The CPRC receive program must ensure that order is maintained with enqueue operations to the QMU within a cluster.
	The CPRC transmit program must ensure that order is maintained with dequeue operations from the QMU within a cluster.
TxSDP	The TxSDP must serialize outgoing packets and cells from the CPs within a cluster to a single physical interface correctly.

Packet/Cell Ordering Handling for Rx in Aggregate Mode

Most network protocols at Layer 2 and Layer 3 require that the forwarding component maintain ordering, but some do not. At the physical layer the distinction between these two (2) scenarios cannot be made. To solve this problem, the C-5 NP maintains ordering of all packets and cells that it processes from a physical interface when aggregated.

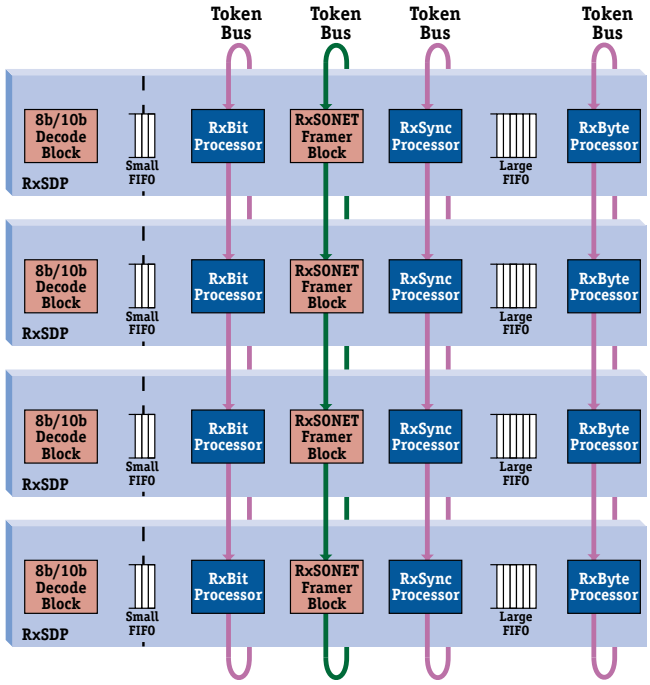
To support distribution of packets or cells evenly to the CPs within a cluster, the C-5 NP uses two (2) types of tokens:

- Hardware tokens in the RxSDP to deliver packets and cells to CPs in a round-robin fashion
- The CPRC software receive programs use software tokens to serialize enqueue operations. This maintains the ordering of descriptors to the QMU

Hardware Receive Tokens

The RxSDP processor provides four (4) *token buses* that run among the RxBit processors, the RxSONET Framer blocks, the RxSync processors, and the RxByte processors within a cluster. Refer to Figure 84 on page 580. The token buses in the RxSDP pass tokens between sequencers. A sequencer cannot forward a packet or cell upstream in the RxSDP until it has the token. The token passing function is asserted by a microprogram running in an RxSDP sequencer, and the token is typically passed by a microprogram based on packet or cell delineation. This ensures that different packets and cells are delivered to different CPs within a cluster in a round robin fashion.

Figure 84 RxSDP Token Buses



Software Receive Tokens

The CPRC receive program is typically notified of packet or cell arrival from the RxSDP. It then makes a forwarding decision and enqueues a descriptor to the QMU for forwarding to the egress interface. In the aggregated case, the software program running on the receive CPRC must ensure that the QMU receives application-defined descriptors in the same order that the packets that the descriptors represent arrived from the physical interface. To achieve this requirement, the software programs running on the CPRC must ensure that the descriptor enqueue operations are serialized.

The CPRC programs achieve this by using a piece of shared DMEM as a software token. A CPRC program only enqueues when it owns the token. The program passes the token to its neighbor after enqueueing the descriptor.

This operation ensures that transactions to the QMU are in the same order in which the packets arrived on the physical interface.

Packet/Cell Ordering Handling for Tx in Aggregate Mode

Like receive aggregation, transmit aggregation must maintain ordering from the QMU to the physical interface. This requires the same level of synchronization and ordering as receive aggregation.

Hardware Transmit Tokens

There is a single hardware token bus that is used in the TxByte processor. Refer to Figure 85 on page 582. The management of this token bus is controlled by the microcode running in the TxByte processor. This token controls the draining of the large FIFO downstream of the TxByte processor.

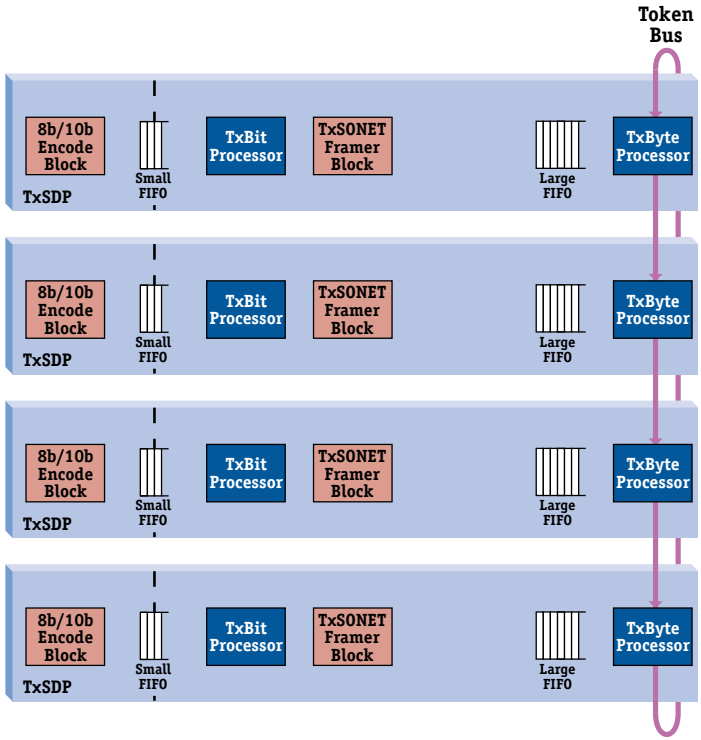
This function allows the TxByte processors to prime the FIFOs even if they do not happen to own the token. When the TxByte processor does own the token, the large FIFO is filled with enough data to keep the TxBit processor — and the physical interface — full.

There is only one (1) TxBit processor that operates per aggregated cluster. This processor gets the data from the large FIFO selected by the TxByte token that is being passed in a round-robin fashion.

Software Transmit Tokens

The CPRC transmit program maintains a software token in shared DMEM for the cluster doing transmit processing for the aggregated physical interface. The transmit program can only send the packet or cell to its TxSDP when it owns the token. After the TxSDP begins packet or cell transmission, the CPRC transmit program passes the token to its neighbor so it can begin the same process.

Figure 85 TxSDP Token Bus



Clock Distribution in Aggregate Mode

The pad data and clocks are wired so that they can be accessed equally by all CPs in the cluster.

Latching is performed at the pads using the configured clock, with the latched data circulated to the SDPs. The pads are grouped such that two (2) CPs have sufficient pads to meet the needs of either transmit or receive. During aggregation, the receive clock from one (1) CP cluster becomes the master receive clock for all of the pads in the two (2) CP clusters. In addition, the PHY chip in OC-12c generates the transmit clock, so the clock is received and then forwarded through the transmit clock mux to become the master transmit clock for the cluster.

The transmit data and clock are available to all four (4) transmit SDPs in CPs 0 to 3 through busing from the pins. Similarly, the receive data and clock is available for all four (4) receive SDPs in CPs 4 to 7 through busing from the pins. OC-12 aggregation of four (4) CPs has the transmit pins allocated from CPs 0 and 2 and receive pins allocated from CPs 1 and 3.

Aggregate Mode Application Examples

Each application that uses CP aggregation (Gigabit Ethernet, FibreChannel, OC-12 and OC-12c) has a slightly different implementation. This section provides details on each.

Gigabit Ethernet and FibreChannel Applications

Both Gigabit Ethernet and FibreChannel use the TBI (ten-bit interface) physical layer encoding. Even though Layer 2 and above of the protocol is different, each of these protocols implements aggregation in the same fashion. GMII for Gigabit Ethernet also follows this same aggregation scheme.

PHY Connectivity

The pins that connect the TBI/GMII and the C-5 NP are spread across all of the pins in a CP cluster. It so happens that the transmit pins from the TBI/GMII are on CP0 and CP1 of the cluster and the receive pins are on CP2 and CP3 of the cluster. Since every CP only has 7 pins associated with it, this configuration is necessary for processing of TBI/GMII protocols. For complete descriptions of Gigabit Ethernet and FibreChannel pinouts, see the *C-5 NP Data Sheet*.

After the pins are routed inside the C-5 NP, they are muxed together and sent as a single 10bit stream to each CP in the cluster for processing as a single stream. Refer to Figure 86 on page 585.

SDP Components

The RxSDP components that are used by these protocols are the 8b/10b Decode block, the RxBit processor, the RxSync processor, and the RxByte processor. The TxSDP components that are used by these protocols are the TxByte processor, the TxBit processor, and the 8b/10b Encode block. The receive path is shown in Figure 86 on page 585, and the transmit path is shown in Figure 87 on page 587.

8b/10b Decode Block

Since Gigabit Ethernet and FibreChannel are 10bit protocols, they each use the 8b/10b Decode block in the RxSDP to convert the 10bit physical layer encoding into the 8bit data on which the C-5 NP can operate.

FibreChannel has a slightly different Loss-of-Synchronization state machine than Gigabit Ethernet that is implemented in this block.

RxBit Processor

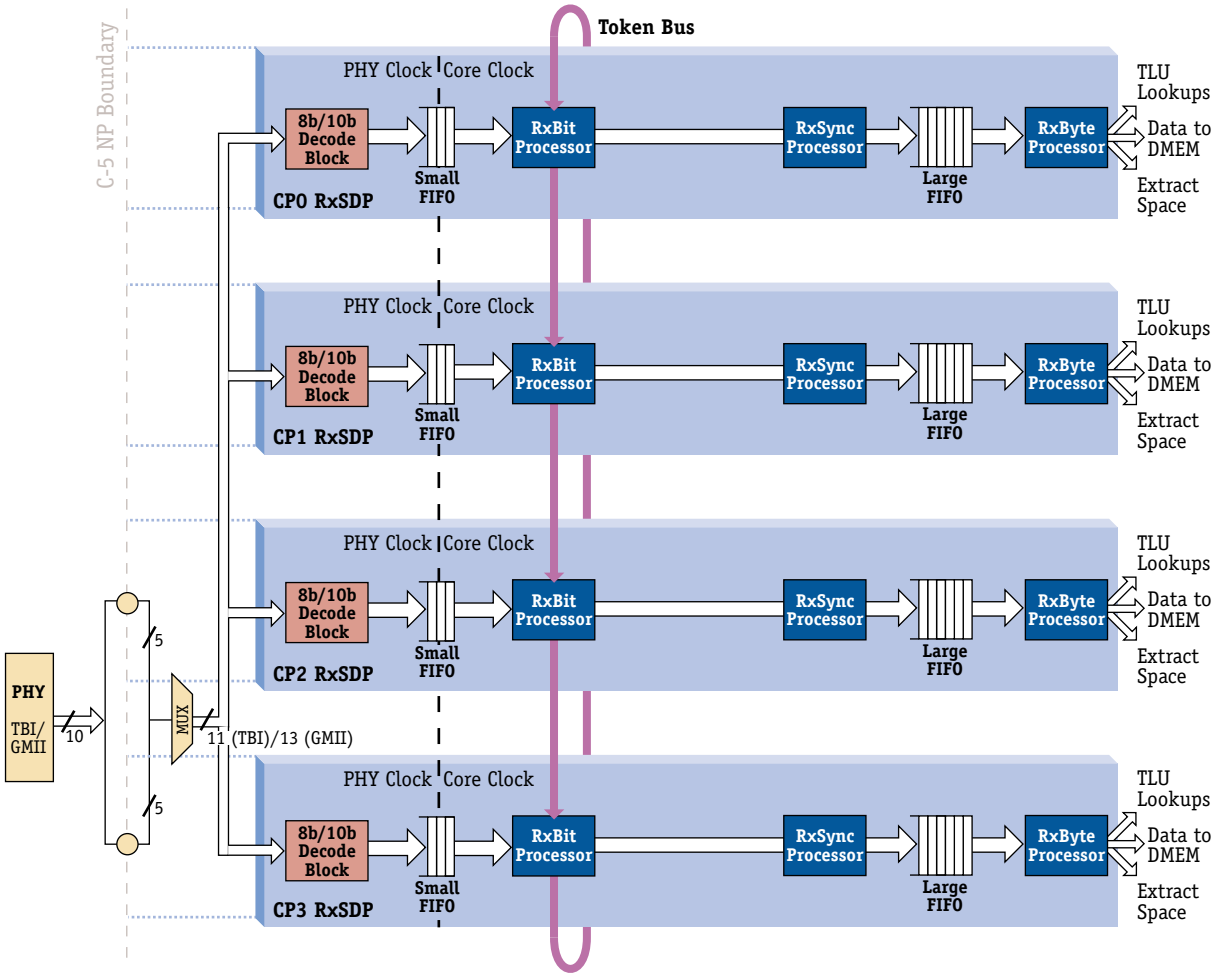
The RxBit processor delivers received packets to the RxSync processor upstream when it owns the token. It then passes the token to the next RxBit processor in line. If the token is not owned by the RxBit processor, the bits are dropped.

Functionally, the RxBit processor is used in these applications for frame delineation, stripping of control characters, and preamble for delivery of the packet without physical layer or control information into the RxSync processor.

RxSync and RxByte Processors

The RxSync processor is used in both FibreChannel and Gigabit for auxiliary processing. The RxByte processor is used for function parsing and processing of Layer 2 and above in Gigabit Ethernet and FibreChannel.

Figure 86 SDP Receive Path for Gigabit Ethernet and FibreChannel



TxByte Processor

The TxByte processor is used in these protocols to distribute packets from the cluster of CPs to a single physical interface by way of the TxBit processor. This is done by each of the TxByte processors in a cluster filling the large FIFO with bytes, but not allowing the TxBit processor to empty its FIFO until it owns the hardware token.

Functionally, the TxByte processor does header updating and replacement, CRC and checksum recalculation of data flowing through to the physical interface.

TxBit Processor

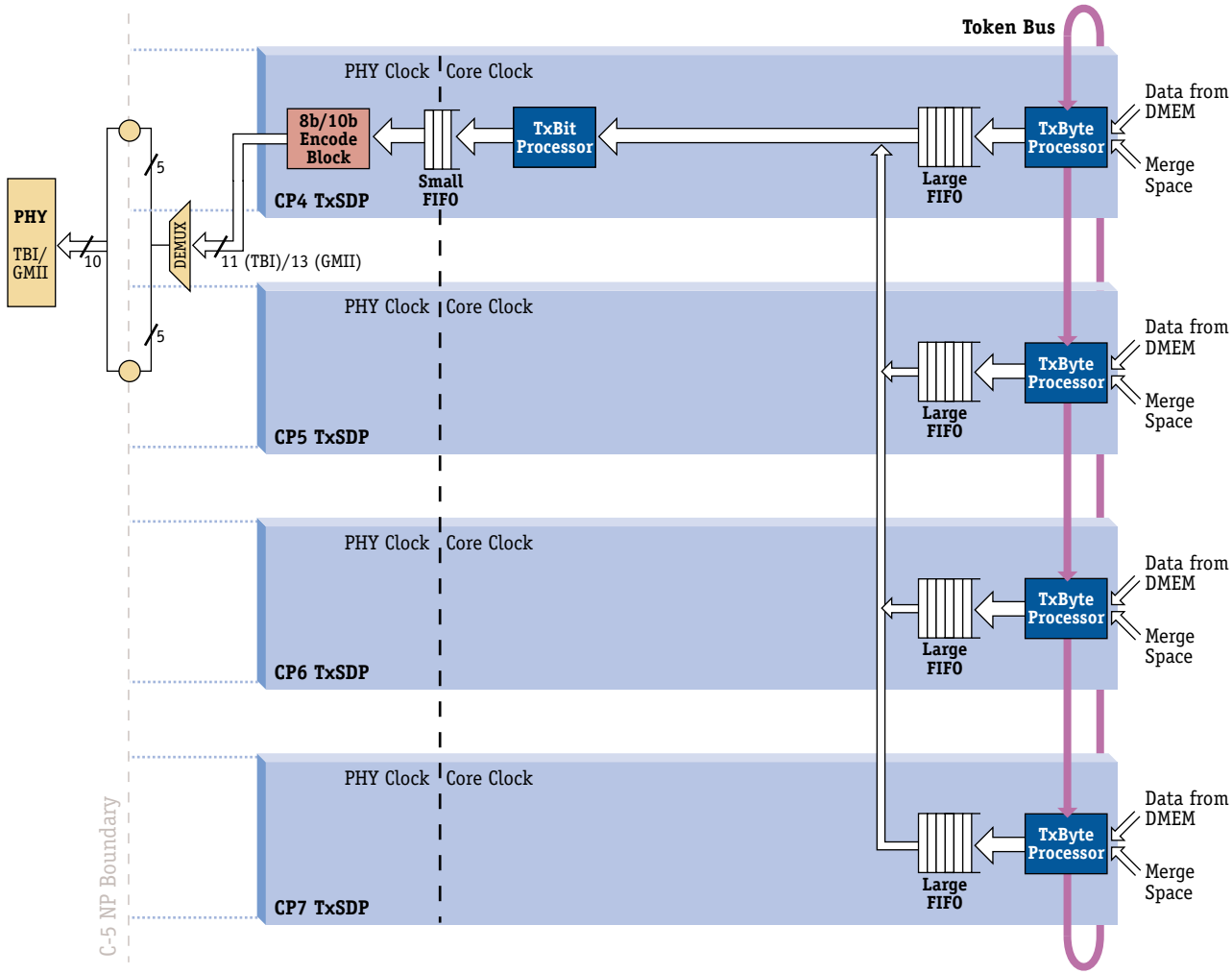
The TxBit processor is used to send the 8bit data bytes to the 8b/10b block for 10bit encoding and transmission out to the physical interface. The TxBit processor appears to receive a single stream of data from the set of large FIFOs from other TxSDPs (in order of what TxByte processor owns the hardware token). Functionally, the TxBit processor is responsible for inserting idle characters, control characters, and inter-packet gaps.

8b/10b Encode Block

The 8b/10b Encode block takes the 8bit data from the TxBit processor and does the proper 10bit encoding before transmission to the physical interface.

In the case of FibreChannel, this encoder inserts the correct End-of-Frame (EOF) word based on the 8b/10b running disparity. There is no such requirement in Gigabit Ethernet.

Figure 87 SDP Transmit Path for Gigabit Ethernet and FibreChannel



Implementation Options

Both Gigabit Ethernet and FibreChannel can be implemented in two (2) different designs.

Non-blocking Operation

For non-blocking switching, one (1) CP cluster can be dedicated to receive processing and one to transmit processing. The reason that this is necessary is to ensure that the CPRC has enough processor cycles to keep up with the speed of the physical interface and not to overload the BMU for memory bandwidth.

This is how the reference applications in the *C-Ware Applications Library* are implemented. Clusters 0 and 2 are dedicated to receive processing and clusters 1 and 3 are dedicated to transmit processing.

The physical design implications are that the transmit pins on CP0 and CP1 on clusters 0 and 2 are tied down since they are not used. Likewise, the receive pins on CP2 and CP3 on clusters 1 and 3 are tied down since they are not used.

Blocking Operation

To have a system with twice the port density, but potentially blocks the C-5 NP, can be configured to have four (4) Gigabit Ethernet or FibreChannel ports by wiring all the receive and transmit data pins on a cluster.

That is each cluster would have a full-duplex PHY connected to it by wiring all the pins on the cluster to one (1) PHY.

OC-12 and OC-12c Applications

Both OC-12 and OC-12c applications, which include Packet-Over-SONET (POS) and ATM, can be implemented with the C-5 NP using CP aggregation. Both of these protocols use the SDP in the same configuration but have subtle differences in serialization and synchronization of the SDPs.

PHY Connectivity

The pins that the physical interface are wired to on the C-5 NP are spread out over the cluster. This is because each CP on the C-5 NP only has 7 external pins associated with it and that is too few to support OC-12 or OC-12c. Internally, the OC-12 or OC-12c data pins are replicated and sent to each of the CPs within the cluster for processing. For complete descriptions of OC-12/OC-12c pinouts, see the *C-5 NP Data Sheet*.

SDP Components

The RxSDP components that are used by these protocols are the RxBit processor, the RxSONET Framer block, the RxSync processor, and the RxByte processor. The TxSDP components that are used by these protocols are the TxByte processor, the TxSONET Framer block, and the TxBit processor. The receive path is shown in Figure 88 on page 590, and the transmit path is shown in Figure 89 on page 592.

RxBit Processor

The RxBit processor is used to perform SONET Frame delineation, that is, it locates the A1/A2 bytes and determines when it is in the Loss-Of-Frame (LOF) .

RxSONET Framer

The RxSONET Framer block has different responsibilities in OC-12 and OC-12c.

- In OC-12, each SONET frame contains four (4) independent OC-3c streams. Each RxSONET Framer block processes the SONET overhead for its OC-3c stream and sends the associated payload up to the RxSync processor.
- In OC-12c, each RxSONET Framer block processes the overhead for the entire OC-12c SONET frame and sends all SONET payload up to the RxSync processor.

RxSync Processor

The RxSync processor performs ATM cell delineation for an STM cell stream, or a Point-to-Point Protocol (PPP) packet processing for Packet-over-SONET (POS) data stream.

The data stream is sightly different between OC-12 and OC-12c applications.

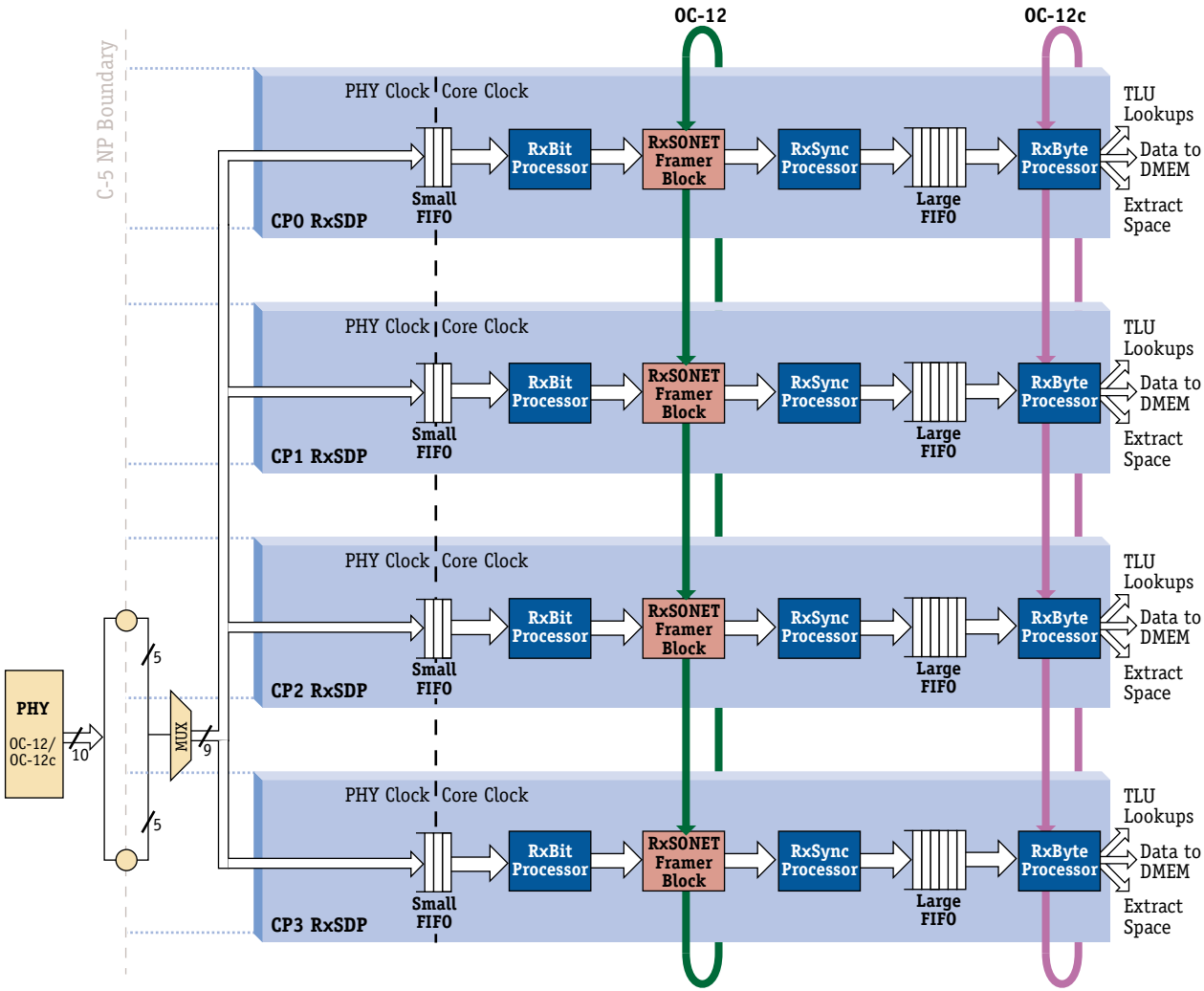
- In OC-12, the RxSync processor receives one (1) of the four (4) OC-3c payload streams.

- In OC-12c, each RxSync processor receives *all* of the payload for the entire OC-12c stream.

RxByte Processor

The RxByte processor does functional parsing of ATM cells or PPP packets and has no special aggregation function.

Figure 88 SDP Receive Path for OC-12 and OC-12c



TxByte Processor

Functionally, the TxByte processor does any type of translation, re-encapsulation, or checksum/CRC regeneration for the protocol.

Transmit FIFO Automatic Token Passing:

All aggregated applications must pass a token in TxByte to switch the data source from one Channel Processor to the next. For the C-5 NP Version C0, TxBit was required to tell TxByte when to pass the token; TxBit waited for the end of a packet to notify TxByte. If TxByte passed the token before TxBit received the last byte of a packet, a new packet from the next Channel Processor would truncate the current packet, leaving the remainder in the previous Channel Processor's FIFO.

The side effect of this technique for handling aggregation is that the transmit FIFO is filled up with a packet, and must be drained until empty. Thus, the effectiveness of the FIFO is greatly reduced.

This technique made it extremely difficult for the Reference Library's OC-12 applications to work properly before the D0 changes. When the TxByte processor passes the token, the TxSONETFramer has already popped several more bytes from the TxLargeFIFO. To work around this problem, the Packet-Over-SONET (POS) applications were required to add approximately five PPP flag characters at the end of each packet so that the TxSONETFramer had valid data to unload "pop" from the TxLargeFIFO as the token was being passed. This same problem prevented ATM applications from working at all in OC-12c.

For the C-5 NP Version D0 a new automatic token passing mechanism was implemented. The new method is for the TxLargeFIFO to check whether the *Merge9* bit is set on a payload byte. When it sees the ninth bit set, it automatically passes the token enabling the TxLargeFIFO on the next Channel Processor.

The advantage of this approach is that TxByte and TxBit no longer need to coordinate the token passing. It also allows TxByte to take full advantage of the transmit FIFO. There is no need to load a packet and wait for the FIFO to drain before sending the next packet, as it can remain as full as possible all the time.

For the POS application the problem of padding out packets with PPP flags is also removed. The minimum number of PPP flag characters can be placed between packets. OC-12 ATM applications now work with the C-5 NP Version D0.

To enable this feature for the C-5 NP Version D0, set the new *Auto Token Enable* bit in the *SDP_MODE4* register. Refer to "SDP_Mode4 Register (CP Mode Configuration Function)" on page 422.

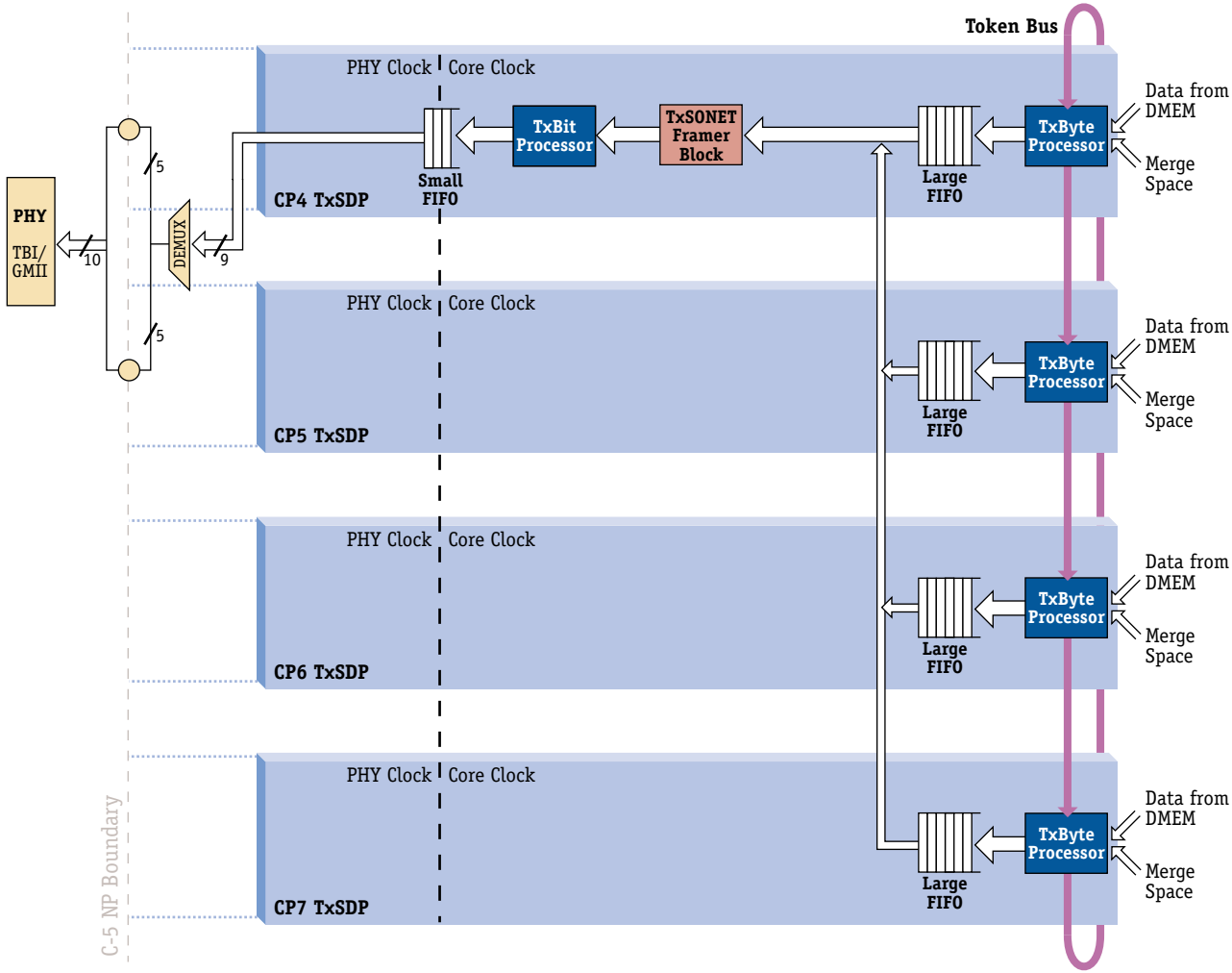
TxSONET Framer

The TxSONET Framer block in these applications adds the correct SONET overhead to the OC-12 or OC-12c payload for transmission out onto the physical medium. The output of the TxSONET Framer block goes to the TxBit processor in the base CP of the cluster.

TxBit Processor

The TxBit processor sends the data bytes to the physical interface for transmission by way of the small FIFO.

Figure 89 SDP Transmit Path for OC-12 and OC-12c



Appendix C

SONET/SDH CP Support

Appendix Overview

This appendix covers the following topics:

- SONET/SDH Overview
- SONET Overhead Access
- CP Configuration Space (SONET Specific)
- SONET/SDH Monitoring Example

SONET/SDH Overview

SONET/SDH provides a diagnostic utility for monitoring line quality and fault isolation.

The C-5 NP SONET/SDH transmit support consists of inserting payload into the SONET/SDH frame on the transmit side. The SONET frame data is read from the Channel Processor (CP) Configuration registers.

The SONET/SDH receive support consists of extracting payload from the SONET/SDH frame and forwarding this payload to the large FIFO of the CPRC. The SONET/SDH frame data is written to the Channel Processor (CP) Configuration registers.

The C-5 NP allows access to a large portion of the SONET Overhead Read Directory by the CPRC or the XP/Host via the Global Bus. This allows a given application the flexibility to add code to support such features as Orderwire or Data Communication Channels.

The C-5 NP Supports three (3) SONET/SDH (Synchronous Digital Hierarchy) configurations:

- OC-3c,
- OC-12c,
- OC-12 with OC-3c streams embedded.

In addition, to the SONET Overhead access on both the transmit (Tx) and receive (Rx) side, the SONET/SDH block provides a function for defect monitoring purposes. SONET defect events are flagged in the *SONET_Event* register. An example of the types of events supported are listed in Table 171 on page 596.

Table 171 Example of Events Reported in the SONET_Event Register

Event Category	Examples	Use
Defects (Non-Pointer related)	LOS, LOF, C2 error (PLM-P), LCD-P	Near-end fault detection.
	AIS-L, RDI-L, AIS-P, RDI-P	Far-end fault detection.
Counters	B1, B2, B3	Near-end error rate detection.
	REI-L, REI-P	Far-end error rate detection.
Pointer Defects	LOP-P, PTR-Change, NDF, H4 Change	Near-end fault detection.
APS Support	APS-Error, K2 Change	Switching protection.
Other OH Support	Z1, Z2, Z3, Z4, Z5 Change, S1 Change	Synchronization states and country specific SDH support.
Trail Trace Support	J0 Change, J1 Available	J0/J1 Patt & Section trace support.

Table 171 Example of Events Reported in the SONET_Event Register (continued)

Event Category	Examples	Use
General Support	Tx overhead complete, Rx transport overhead available, Rx patt overhead available	Useful for updating overhead on Tx and checking other SONET overhead on Rx when no other interrupt is available.

In addition, the SONET block can automatically forward far-end alarm indications on the same port when errors are detected on the line (e. g. LOS, LOF, B1, etc.) This is done when the *SDP_Mode3* register bit [13] *Manual_FEBE* field is set to 0.

SONET Overhead Access

Configuration Space of the CPs includes a number of registers that are pertain to implementing and using the SONET functions. Primarily, thirty-two (32) (4Byte) (128Byte wide) registers are used for storing receive (Rx) and transmit (Tx) SONET overhead. The SDP RxSONET Framer block writes bytes to the space in a manner similar to way the SDP RxByte Processor writes to Extract Space. The CPRC can read the Receive SONET registers at any time. The CPRC can write to the Receive SONET registers, but only during initialization and test periods (when the *SDP_Mode3* register bit [30] *RxEnable* field is clear).

The SDP TxSONET Framer block reads bytes from the space, similar to the way merge registers are used by the SDP TxByte Processor. The CPRC process can read or write the Transmit SONET registers at any time. Refer to "Rx_SONETOH0 to Rx_SONETOH31 Registers (CP SONET Rx Control Function)" on page 408 and "Tx_SONETOH0 to Tx_SONETOH31 Registers (CP SONET Tx Control Function)" on page 408.

The SONET Overhead positions are shown:

- For SONET OC-3 refer to Figure 90 on page 599,
- For SONET OC-12c refer to Figure 91 on page 600, and
- For SONET OC-12 refer to Figure 92 on page 601.

The detail mapping information listing the SONET overhead definitions and C-5 NP addresses are grouped by protocol (OC-3c or OC-12/OC-12c) and whether the overhead contents are Transport or Path bytes are shown:

- For Rx SONET OC-3 Transport Overhead Byte Address refer to Table 172 on page 602,
- For Rx SONET OC-3c Path Overhead Definitions refer to Table 175 on page 604,
- For Rx SONET OC-12/OC-12c Transport Overhead Definitions refer to Table 176 on page 605,
- For Rx SONET OC-12/OC-12c Path Overhead Definitions refer to Table 177 on page 610,
- For Tx SONET OC-3 Transport Overhead Definitions refer to Table 178 on page 612,
- For Tx SONET OC-3 Path Overhead Definitions refer to Table 179 on page 613,
- For Tx SONET OC-12/OC-12c Transport Overhead Definitions refer to Table 180 on page 614,
- For Tx OC-12/OC-12c Path Overhead Definitions refer to Table 181 on page 618.

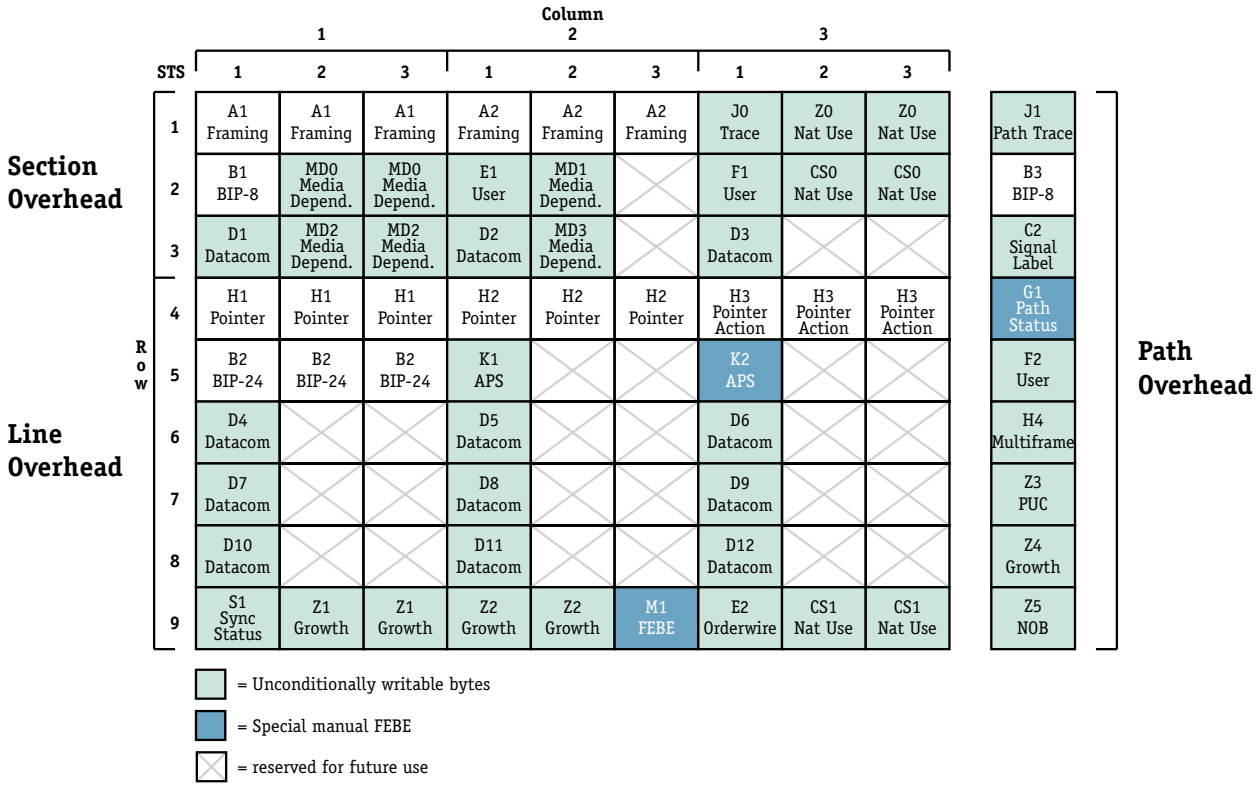
SONET Overhead Writable Bytes

The following figures show the writable bytes in the SONET Overhead.

OC-3c Writable Overhead Bytes

Figure 90 on page 599 shows the writable bytes in the OC-3c SONET Overhead.

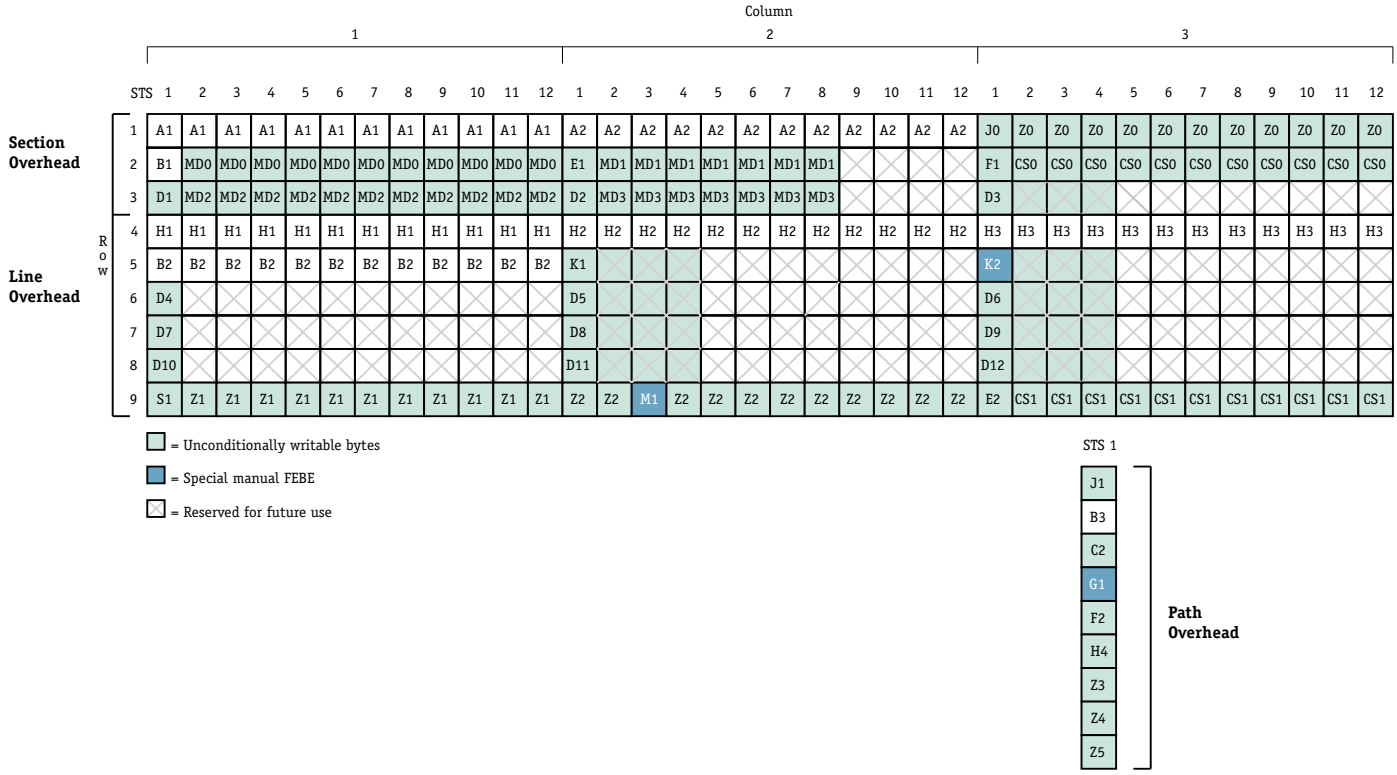
Figure 90 SONET OC-3c Writable Overhead Bytes



OC-12c Writable Overhead Bytes

Figure 91 on page 600 shows the writable bytes in the OC-12c SONET Overhead.

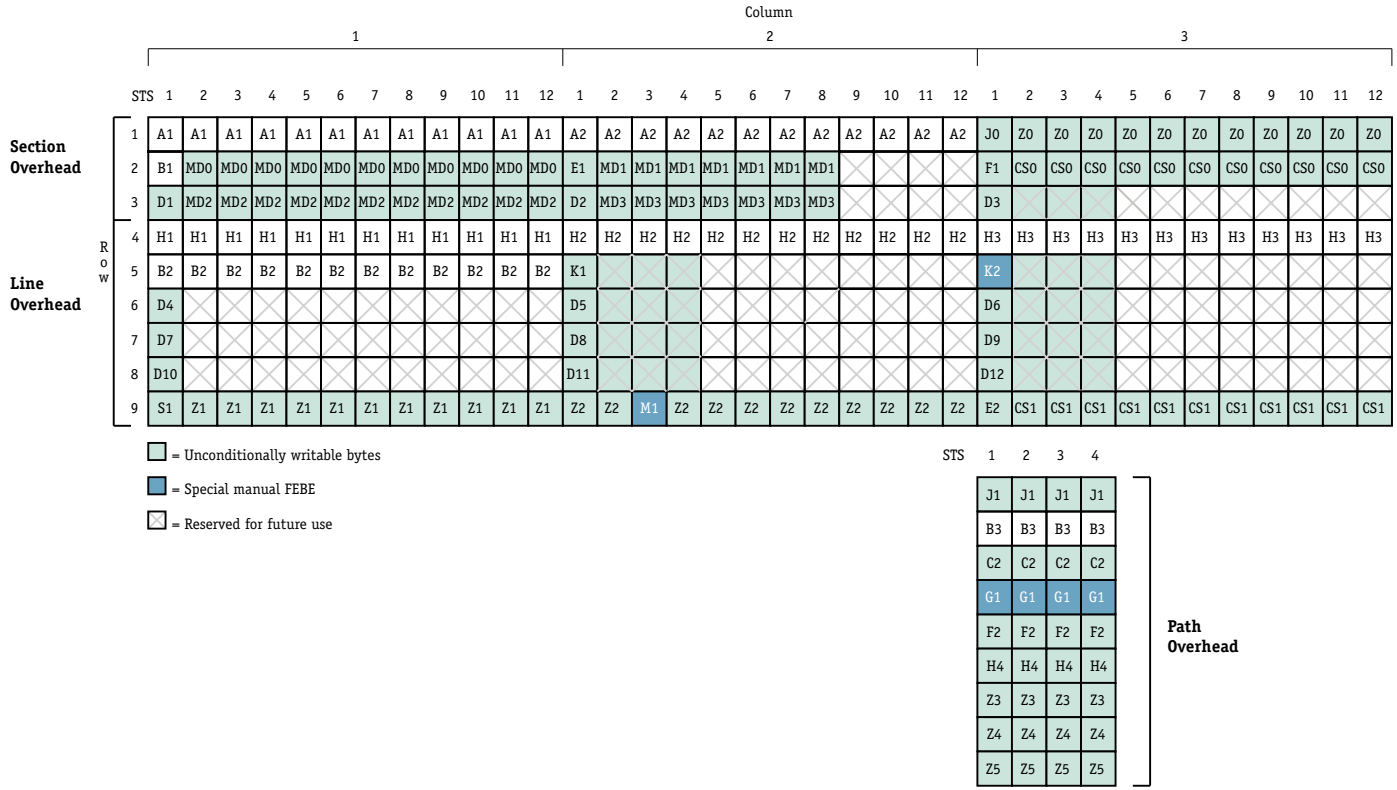
Figure 91 SONET OC-12c Writable Overhead Bytes



OC-12 Writable Overhead Bytes

Figure 92 on page 601 shows the writable bytes in the OC-12 SONET Overhead.

Figure 92 SONET OC-12 Writable Overhead Bytes



SONET Overhead Definitions

The following tables list the SONET Overhead definitions and addresses.

For detail descriptions of the SONET transport overhead bytes. Refer to *Telcordia Generic Requirements (GR-253-CORE) Synchronous Optical Network: (SONET) Transport Systems: Generic Criteria (Issue 2, Revision 2)*.

Receive OC-3c Transport Overhead Definitions

Table 172 Receive SONET OC-3 Transport Overhead Byte Addresses

Transport Overhead Byte	C-5 NP Address
J0	0xBC*n*04500
Z0, STS #2	0xBC*n*04501
Z0, STS #3	0xBC*n*04502
B1*	0xBC*n*04503
MD0, STS #2†	0xBC*n*04504
MD0, STS #3†	0xBC*n*04505
E1	0xBC*n*04506
MD1, STS #2†	0xBC*n*04507
F1	0xBC*n*04508
CS0, STS #2‡	0xBC*n*04509
CS0, STS #3‡	0xBC*n*0450A
D1	0xBC*n*0450B
MD2, STS #2†	0xBC*n*0450C
MD2, STS #3†	0xBC*n*0450D
D2	0xBC*n*0450E
MD3, STS #2†	0xBC*n*0450F
D3	0xBC*n*04510
H1, STS #1**	0xBC*n*04511
H2, STS #1**	0xBC*n*04512
B2, STS #1††	0xBC*n*04513
B2, STS #2††	0xBC*n*04514
B2, STS #3††	0xBC*n*04515
K1‡‡	0xBC*n*04516
K2‡‡	0xBC*n*04517

Table 172 Receive SONET OC-3 Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	C-5 NP Address
D4	0xBC*n*04518
D5	0xBC*n*04519
D6	0xBC*n*0451A
D7	0xBC*n*0451B
D8	0xBC*n*0451C
D9	0xBC*n*0451D
D10	0xBC*n*0451E
D11	0xBC*n*0451F
D12	0xBC*n*04520
S1	0xBC*n*04521
Z1, STS #2	0xBC*n*04522
Z1, STS #3	0xBC*n*04523
Z2, STS #1	0xBC*n*04524
Z2, STS #2	0xBC*n*04525
M1	0xBC*n*04526
E2	0xBC*n*04527
CS1, STS #2‡	0xBC*n*04528
CS1, STS #3‡	0xBC*n*04529

* For the parity bytes locations for B1 and B3, it is not the actual parity byte written to the register, but the number of bit lanes in error. The number of errors reported is therefore 0 through 8.

† These refer to the SDH media dependent overhead.

‡ These refer to the SDH country specific overhead.

** For the pointer byte slots H1 and H2, it is not the actual values of H1 and H2 that are written, but the pointer processing results listed in Table 173 on page 604.

†† For the parity bytes locations for B2, it is not the actual parity byte written to the register, but an error count. For the least significant (lowest index) STS of a CPRC, the number of errors reported is the sum of bit lanes in error present in all three of the STSs received by that CPRC. The number of errors reported is therefore 0 through 24. For other than the least significant STS received by the CPRC, the reported error count is the number of bit lanes in error for that one STS. The number of errors reported is therefore 0 through 8.

‡‡ These APS bytes are written to the registers only when three identical bytes have been received in consecutive frames.

Table 173 Pointer Values for H1 and H2

Bit	H1 Value	H2 Value
7	pointer status (bit 1)	current pointer bit 7
6	pointer status (bit 0)	current pointer bit 6
5	new data flag	current pointer bit 5
4	pointer increment	current pointer bit 4
3	pointer decrement	current pointer bit 3
2	zero	current pointer bit 2
1	current pointer bit 9	current pointer bit 1
0	current pointer bit 8	current pointer bit 0

The decode for pointer status (H1 bits [7:6]) is shown in Table 174.

Table 174 Decode for Pointer Status [1:0]

Pointer Status (H1 bits [7:6])	Condition
00	good pointer
01	unused
10	path AIS (AIS-P)
11	loss of pointer (LOP)

Receive OC-3c Path Overhead Definitions

Table 175 Receive SONET OC-3c Path Overhead Byte Addresses

Path Overhead Byte	C-5 NP Address
J1, STS #1*	0xBC*n*0452C
B3, STS #1†	0xBC*n*0452D
C2, STS #1	0xBC*n*0452E
G1, STS #1	0xBC*n*0452F
F2, STS #1	0xBC*n*04530
H4, STS #1	0xBC*n*04531
Z3, STS #1	0xBC*n*04532
Z4, STS #1	0xBC*n*04533
Z5, STS #1	0xBC*n*04534

* The contents of this register is the Nth J1 of the 64 byte path trace message. N starts at zero when the receive SONET logic is enabled and increments modulo 64 every SONET SPE.

† For the parity bytes locations for B1 and B3, it is not the actual parity byte written to the register, but the number of bit lanes in error. The number of errors reported is therefore 0 through 8.

Receive OC-12/OC-12c Transport Overhead Definitions

Table 176 Receive SONET OC-12 and OC-12c Transport Overhead Byte Addresses

Transport Overhead Byte	**CP# Within a Cluster**	**C-5 NP Address***
J0	0	0xBC*n*04500
Z0, STS #2	1	0xBC*(n+1)*04500
Z0, STS #3	2	0xBC*(n+2)*04500
Z0, STS #4	3	0xBC*(n+3)*04500
Z0, STS #5	0	0xBC*n*04501
Z0, STS #6	1	0xBC*(n+1)*04501
Z0, STS #7	2	0xBC*(n+2)*04501
Z0, STS #8	3	0xBC*(n+3)*04501
Z0, STS #9	0	0xBC*n*04502
Z0, STS #10	1	0xBC*(n+1)*04502
Z0, STS #11	2	0xBC*(n+2)*04502
Z0, STS #12	3	0xBC*(n+3)*04502
B1†	0	0xBC*n*04503
MD0, STS #2‡	1	0xBC*(n+1)*04503
MD0, STS #3‡	2	0xBC*(n+2)*04503
MD0, STS #4‡	3	0xBC*(n+3)*04503
MD0, STS #5‡	0	0xBC*n*04504
MD0, STS #6‡	1	0xBC*(n+1)*04504
MD0, STS #7‡	2	0xBC*(n+2)*04504
MD0, STS #8‡	3	0xBC*(n+3)*04504
MD0, STS #9‡	0	0xBC*n*04505
MD0, STS #10‡	1	0xBC*(n+1)*04505
MD0, STS #11‡	2	0xBC*(n+2)*04505
MD0, STS #12‡	3	0xBC*(n+3)*04505

Table 176 Receive SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	**CP# Within a Cluster**	**C-5 NP Address***
E1	0	0xBC*n*04506
MD1, STS #2‡	1	0xBC*(n+1)*04506
MD1, STS #3‡	2	0xBC*(n+2)*04506
MD1, STS #4‡	3	0xBC*(n+3)*04506
MD1, STS #5‡	0	0xBC*n*04507
MD1, STS #6‡	1	0xBC*(n+1)*04507
MD1, STS #7‡	2	0xBC*(n+2)*04507
MD1, STS #8‡	3	0xBC*(n+3)*04507
F1	0	0xBC*n*04508
CS0, STS #2**	1	0xBC*(n+1)*04508
CS0, STS #3**	2	0xBC*(n+2)*04508
CS0, STS #4**	3	0xBC*(n+3)*04508
CS0, STS #5**	0	0xBC*n*04509
CS0, STS #6**	1	0xBC*(n+1)*04509
CS0, STS #7**	2	0xBC*(n+2)*04509
CS0, STS #8**	3	0xBC*(n+3)*04509
CS0, STS #9**	0	0xBC*n*0450A
CS0, STS #10**	1	0xBC*(n+1)*0450A
CS0, STS #11**	2	0xBC*(n+2)*0450A
CS0, STS #12**	3	0xBC*(n+3)*0450A
D1	0	0xBC*n*0450B
MD2, STS #2‡	1	0xBC*(n+1)*0450B
MD2, STS #3‡	2	0xBC*(n+2)*0450B
MD2, STS #4‡	3	0xBC*(n+3)*0450B
MD2, STS #5‡	0	0xBC*n*0450C
MD2, STS #6‡	1	0xBC*(n+1)*0450C
MD2, STS #7‡	2	0xBC*(n+2)*0450C
MD2, STS #8‡	3	0xBC*(n+3)*0450C
MD2, STS #9‡	0	0xBC*n*0450D

Table 176 Receive SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	**CP# Within a Cluster**	**C-5 NP Address***
MD2, STS #10‡	1	0xBC*(n+1)*0450D
MD2, STS #11‡	2	0xBC*(n+2)*0450D
MD2, STS #12‡	3	0xBC*(n+3)*0450D
D2	0	0xBC*n*0450E
MD3, STS #2‡	1	0xBC*(n+1)*0450E
MD3, STS #3‡	2	0xBC*(n+2)*0450E
MD3, STS #4‡	3	0xBC*(n+3)*0450E
MD3, STS #5‡	0	0xBC*n*0450F
MD3, STS #6‡	1	0xBC*(n+1)*0450F
MD3, STS #7‡	2	0xBC*(n+2)*0450F
MD3, STS #8‡	3	0xBC*(n+3)*0450F
D3	N/A	0xBC*n*04510
H1, STS #1††	0	0xBC*n*04511
H1, STS #2††	1	0xBC*(n+1)*04511
H1, STS #3††	2	0xBC*(n+2)*04511
H1, STS #4††	3	0xBC*(n+3)*04511
H2, STS #1††	0	0xBC*n*04512
H2, STS #2††	1	0xBC*(n+1)*04512
H2, STS #3††	2	0xBC*(n+2)*04512
H2, STS #4††	3	0xBC*(n+3)*04512
B2, STS #1‡‡	0	0xBC*n*04513
B2, STS #2‡‡	1	0xBC*(n+1)*04513
B2, STS #3‡‡	2	0xBC*(n+2)*04513
B2, STS #4‡‡	3	0xBC*(n+3)*04513
B2, STS #5‡‡	0	0xBC*n*04514
B2, STS #6‡‡	1	0xBC*(n+1)*04514
B2, STS #7‡‡	2	0xBC*(n+2)*04514
B2, STS #8‡‡	3	0xBC*(n+3)*04514
B2, STS #9‡‡	0	0xBC*n*04515

Table 176 Receive SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	**CP# Within a Cluster**	**C-5 NP Address***
B2, STS #10‡‡	1	0xBC*(n+1)*04515
B2, STS #11‡‡	2	0xBC*(n+2)*04515
B2, STS #12‡‡	3	0xBC*(n+3)*04515
K1***	N/A	0xBC*n*04516
K2***	N/A	0xBC*n*04517
D4	N/A	0xBC*n*04518
D5	N/A	0xBC*n*04519
D6	N/A	0xBC*n*0451A
D7	N/A	0xBC*n*0451B
D8	N/A	0xBC*n*0451C
D9	N/A	0xBC*n*0451D
D10	N/A	0xBC*n*0451E
D11	N/A	0xBC*n*0451F
D12	N/A	0xBC*n*04520
S1	0	0xBC*n*04521
Z1, STS #2	1	0xBC*(n+1)*04521
Z1, STS #3	2	0xBC*(n+2)*04521
Z1, STS #4	3	0xBC*(n+3)*04521
Z1, STS #5	0	0xBC*n*04522
Z1, STS #6	1	0xBC*(n+1)*04522
Z1, STS #7	2	0xBC*(n+2)*04522
Z1, STS #8	3	0xBC*(n+3)*04522
Z1, STS #9	0	0xBC*n*04523
Z1, STS #10	1	0xBC*(n+1)*04523
Z1, STS #11	2	0xBC*(n+2)*04523
Z1, STS #12	3	0xBC*(n+3)*04523
Z2, STS #1	0	0xBC*n*04524
Z2, STS #2	1	0xBC*(n+1)*04524
M1	2	0xBC*(n+2)*04524

Table 176 Receive SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	**CP# Within a Cluster**	**C-5 NP Address***
Z2, STS #4	3	0xBC*(n+3)*04524
Z2, STS #5	0	0xBC*n*04525
Z2, STS #6	1	0xBC*(n+1)*04525
Z2, STS #7	2	0xBC*(n+2)*04525
Z2, STS #8	3	0xBC*(n+3)*04525
Z2, STS #9	0	0xBC*n*04526
Z2, STS #10	1	0xBC*(n+1)*04526
Z2, STS #11	2	0xBC*(n+2)*04526
Z2, STS #12	3	0xBC*(n+3)*04526
E2	0	0xBC*n*04527
CS1, STS #2**	1	0xBC*(n+1)*04527
CS1, STS #3**	2	0xBC*(n+2)*04527
CS1, STS #4**	3	0xBC*(n+3)*04527
CS1, STS #5**	0	0xBC*n*04528
CS1, STS #6**	1	0xBC*(n+1)*04528
CS1, STS #7**	2	0xBC*(n+2)*04528
CS1, STS #8**	3	0xBC*(n+3)*04528
CS1, STS #9**	0	0xBC*n*04529
CS1, STS #10**	1	0xBC*(n+1)*04529
CS1, STS #11**	2	0xBC*(n+2)*04529
CS1, STS #12**	3	0xBC*(n+3)*04529

* *n* can be CP0, CP4, CP8, or CP12

† For the parity bytes locations for B1 and B3, it is not the actual parity byte written to the register, but the number of bit lanes in error. The number of errors reported is therefore 0 through 8.

‡ These refer to the SDH media dependent overhead.

** These refer to the SDH country specific overhead.

†† For the pointer byte slots H1 and H2, it is not the actual values of H1 and H2 that are written, but the pointer processing results listed in Table 173.

‡‡For the parity bytes locations for B2, it is not the actual parity byte written to the register, but an error count. For the least significant (lowest index) STS of a CPRC, the number of errors reported is the sum of bit lanes in error present in all three of the STSs received by that CPRC. The number of errors reported is therefore 0 through 24. For other than the least significant STS received by the CPRC, the reported error count is the number of bit lanes in error for that one STS. The number of errors reported is therefore 0 through 8.

***These APS bytes are written to the registers only when three identical bytes have been received in consecutive frames.

Receive OC-12/OC-12c Path Overhead Definitions

Table 177 Receive SONET OC-12 and OC-12c Path Overhead Byte Addresses

Path Overhead Byte	CP# Within a Cluster	C-5 NP Address*
J1, STS #1†	0	0xBC*n*0452C
J1, STS #2†	1	0xBC*(n+1)*0452C
J1, STS #3†	2	0xBC*(n+2)*0452C
J1, STS #4†	3	0xBC*(n+3)*0452C
B3, STS #1‡	0	0xBC*n*0452D
B3, STS #2‡	1	0xBC*(n+1)*0452D
B3, STS #3‡	2	0xBC*(n+2)*0452D
B3, STS #4‡	3	0xBC*(n+3)*0452D
C2, STS #1	0	0xBC*n*0452E
C2, STS #2	1	0xBC*(n+1)*0452E
C2, STS #3	2	0xBC*(n+2)*0452E
C2, STS #4	3	0xBC*(n+3)*0452E
G1, STS #1	0	0xBC*n*0452F
G1, STS #2	1	0xBC*(n+1)*0452F
G1, STS #3	2	0xBC*(n+2)*0452F
G1, STS #4	3	0xBC*(n+3)*0452F
F2, STS #1	0	0xBC*n*04530
F2, STS #2	1	0xBC*(n+1)*04530
F2, STS #3	2	0xBC*(n+2)*04530
F2, STS #4	3	0xBC*(n+3)*04530
H4, STS #1	0	0xBC*n*04531
H4, STS #2	1	0xBC*(n+1)*04531

Table 177 Receive SONET OC-12 and OC-12c Path Overhead Byte Addresses (continued)

Path Overhead Byte	CP# Within a Cluster	C-5 NP Address*
H4, STS #3	2	0xBC*(n+2)*04531
H4, STS #4	3	0xBC*(n+3)*04531
Z3, STS #1	0	0xBC*n*04532
Z3, STS #2	1	0xBC*(n+1)*04532
Z3, STS #3	2	0xBC*(n+2)*04532
Z3, STS #4	3	0xBC*(n+3)*04532
Z4, STS #1	0	0xBC*n*04533
Z4, STS #2	1	0xBC*(n+1)*04533
Z4, STS #3	2	0xBC*(n+2)*04533
Z4, STS #4	3	0xBC*(n+3)*04533
Z5, STS #1	0	0xBC*n*04534
Z5, STS #2	1	0xBC*(n+1)*04534
Z5, STS #3	2	0xBC*(n+2)*04534
Z5, STS #4	3	0xBC*(n+3)*04534

* *n* can be CP0, CP4, CP8, or CP12

† The contents of this register is the Nth J1 of the 64 byte path trace message. N starts at zero when the receive SONET logic is enabled and increments modulo 64 every SONET SPE.

‡ For the parity bytes locations for B1 and B3, it is not the actual parity byte written to the register, but the number of bit lanes in error. The number of errors reported is therefore 0 through 8.

Transmit OC-3c Transport Overhead Definitions

Table 178 Transmit SONET OC-3c Transport Overhead Byte Addresses

Transport Overhead Byte	C-5 NP Address
J0	0xBC*n*04580
Z0, STS #2	0xBC*n*04581
Z0, STS #3	0xBC*n*04582
MD0, STS #2*	0xBC*n*04584
MD0, STS #3*	0xBC*n*04585
E1	0xBC*n*04586
MD1, STS #2*	0xBC*n*04587
F1	0xBC*n*04588
CS0, STS #2*	0xBC*n*04589
CS0, STS #3*	0xBC*n*0458A
D1	0xBC*n*0458B
MD2, STS #2*	0xBC*n*0458C
MD2, STS #3*	0xBC*n*0458D
D2	0xBC*n*0458E
MD3, STS #2*	0xBC*n*0458F
D3	0xBC*n*04590
K1	0xBC*n*04591
K2	0xBC*n*04592
D4	0xBC*n*04593
D5	0xBC*n*04594
D6	0xBC*n*04595
D7	0xBC*n*04596
D8	0xBC*n*04597
D9	0xBC*n*04598
D10	0xBC*n*04599
D11	0xBC*n*0459A
D12	0xBC*n*0459B
S1	0xBC*n*0459C

Table 178 Transmit SONET OC-3c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	C-5 NP Address
Z1, STS #2	0xBC*n*0459D
Z1, STS #3	0xBC*n*0459E
Z2, STS #1	0xBC*n*0459F
Z2, STS #2	0xBC*n*045A0
M1	0xBC*n*045A1
E2	0xBC*n*045A2
CSE2, STS #2†	0xBC*n*045A3
CSE2, STS #3†	0xBC*n*045A4

* These refer to the SDH media dependent overhead.
† These refer to the SDH country specific overhead.

Transmit OC-3c Path Overhead Definitions

Table 179 Transmit SONET OC-3c Path Overhead Byte Addresses

Path Overhead Byte	C-5 NP Address
C2, STS #1	0xBC*n*045A5
G1, STS #1	0xBC*n*045A6
F2, STS #1	0xBC*n*045A7
H4, STS #1	0xBC*n*045A8
Z3, STS #1	0xBC*n*045A9
Z4, STS #1	0xBC*n*045AA
Z5, STS #1	0xBC*n*045AB
J1, STS #1*	0xBC*n*045AC to 0xBC*n*045EB
H1, STS #1	0xBC*n*45EC

* 64 bytes of J1 are buffered. All other transmit overhead in single buffered.

Transmit OC-12/OC-12c Transport Overhead Definitions

Table 180 Transmit SONET OC-12 and OC-12c Transport Overhead Byte Addresses

Transport Overhead Byte	CP# Within a Cluster	C-5 NP Address*
J0	0	0xBC*n*04580
Z0, STS #2	1	0xBC*(n+1)*04580
Z0, STS #3	2	0xBC*(n+2)*04580
Z0, STS #4	3	0xBC*(n+3)*04580
Z0, STS #5	0	0xBC*n*04581
Z0, STS #6	1	0xBC*(n+1)*04581
Z0, STS #7	2	0xBC*(n+2)*04581
Z0, STS #8	3	0xBC*(n+3)*04581
Z0, STS #9	0	0xBC*n*04582
Z0, STS #10	1	0xBC*(n+1)*04582
Z0, STS #11	2	0xBC*(n+2)*04582
Z0, STS #12	3	0xBC*(n+3)*04582
MD0, STS #2†	1	0xBC*(n+1)*04583
MD0, STS #3†	2	0xBC*(n+2)*04583
MD0, STS #4†	3	0xBC*(n+3)*04583
MD0, STS #5†	0	0xBC*n*04584
MD0, STS #6†	1	0xBC*(n+1)*04584
MD0, STS #7†	2	0xBC*(n+2)*04584
MD0, STS #8†	3	0xBC*(n+3)*04584
MD0, STS #9†	0	0xBC*n*04585
MD0, STS #10†	1	0xBC*(n+1)*04585
MD0, STS #11†	2	0xBC*(n+2)*04585
MD0, STS #12†	3	0xBC*(n+3)*04585
E1	0	0xBC*n*04586
MD1, STS #2†	1	0xBC*(n+1)*04586
MD1, STS #3†	2	0xBC*(n+2)*04586
MD1, STS #4†	3	0xBC*(n+3)*04586

Table 180 Transmit SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	CP# Within a Cluster	C-5 NP Address*
MD1, STS #5†	0	0xBC*n*04587
MD1, STS #6†	1	0xBC*(n+1)*04587
MD1, STS #7†	2	0xBC*(n+2)*04587
MD1, STS #8†	3	0xBC*(n+3)*04587
F1	0	0xBC*n*04588
CS0, STS #2†	1	0xBC*(n+1)*04588
CS0, STS #3†	2	0xBC*(n+2)*04588
CS0, STS #4†	3	0xBC*(n+3)*04588
CS0, STS #5‡	0	0xBC*n*04589
CS0, STS #6‡	1	0xBC*(n+1)*04589
CS0, STS #7‡	2	0xBC*(n+2)*04589
CS0, STS #8‡	3	0xBC*(n+3)*04589
CS0, STS #9‡	0	0xBC*n*0458A
CS0, STS #10‡	1	0xBC*(n+1)*0458A
CS0, STS #11‡	2	0xBC*(n+2)*0458A
CS0, STS #12‡	3	0xBC*(n+3)*0458A
D1	0	0xBC*n*0458B
MD2, STS #2†	1	0xBC*(n+1)*0458B
MD2, STS #3†	2	0xBC*(n+2)*0458B
MD2, STS #4†	3	0xBC*(n+3)*0458B
MD2, STS #5†	0	0xBC*n*0458C
MD2, STS #6†	1	0xBC*(n+1)*0458C
MD2, STS #7†	2	0xBC*(n+2)*0458C
MD2, STS #8†	3	0xBC*(n+3)*0458C
MD2, STS #9†	0	0xBC*n*0458D
MD2, STS #10†	1	0xBC*(n+1)*0458D
MD2, STS #11†	2	0xBC*(n+2)*0458D
MD2, STS #12†	3	0xBC*(n+3)*0458D
D2	0	0xBC*n*0458E

Table 180 Transmit SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	CP# Within a Cluster	C-5 NP Address*
MD3, STS #2†	1	0xBC*(n+1)*0458E
MD3, STS #3†	2	0xBC*(n+2)*0458E
MD3, STS #4†	3	0xBC*(n+3)*0458E
MD3, STS #5†	0	0xBC*n*0458F
MD3, STS #6†	1	0xBC*(n+1)*0458F
MD3, STS #7†	2	0xBC*(n+2)*0458F
MD3, STS #8†	3	0xBC*(n+3)*0458F
D3	N/A	0xBC*n*04590
K1	N/A	0xBC*n*04591
K2	N/A	0xBC*n*04592
D4	N/A	0xBC*n*04593
D5	N/A	0xBC*n*04594
D6	N/A	0xBC*n*04595
D7	N/A	0xBC*n*04596
D8	N/A	0xBC*n*04597
D9	N/A	0xBC*n*04598
D10	N/A	0xBC*n*04599
D11	N/A	0xBC*n*0459A
D12	N/A	0xBC*n*0459B
S1	0	0xBC*n*0459C
Z1, STS #2	1	0xBC*(n+1)*0459C
Z1, STS #3	2	0xBC*(n+2)*0459C
Z1, STS #4	3	0xBC*(n+3)*0459C
Z1, STS #5	0	0xBC*n*0459D
Z1, STS #6	1	0xBC*(n+1)*0459D
Z1, STS #7	2	0xBC*(n+2)*0459D
Z1, STS #8	3	0xBC*(n+3)*0459D
Z1, STS #9	0	0xBC*n*0459E
Z1, STS #10	1	0xBC*(n+1)*0459E

Table 180 Transmit SONET OC-12 and OC-12c Transport Overhead Byte Addresses (continued)

Transport Overhead Byte	CP# Within a Cluster	C-5 NP Address*
Z1, STS #11	2	0xBC*(n+2)*0459E
Z1, STS #12	3	0xBC*(n+3)*0459E
Z2, STS #1	0	0xBC*n*0459F
Z2, STS #2	1	0xBC*(n+1)*0459F
M1	2	0xBC*(n+2)*0459F
Z2, STS #4	3	0xBC*(n+3)*0459F
Z2, STS #5	0	0xBC*n*045A0
Z2, STS #6	1	0xBC*(n+1)*045A0
Z2, STS #7	2	0xBC*(n+2)*045A0
Z2, STS #8	3	0xBC*(n+3)*045A0
Z2, STS #9	0	0xBC*n*045A1
Z2, STS #10	1	0xBC*(n+1)*045A1
Z2, STS #11	2	0xBC*(n+2)*045A1
Z2, STS #12	3	0xBC*(n+3)*045A1
E2	0	0xBC*n*045A2
CSE2, STS #2‡	1	0xBC*(n+1)*045A2
CSE2, STS #3‡	2	0xBC*(n+2)*045A2
CSE2, STS #4‡	3	0xBC*(n+3)*045A2
CS2, STS #5‡	0	0xBC*n*045A3
CS2, STS #6‡	1	0xBC*(n+1)*045A3
CS2, STS #7‡	2	0xBC*(n+2)*045A3
CS2, STS #8‡	3	0xBC*(n+3)*045A3
CS3, STS #9‡	0	0xBC*n*045A4
CS3, STS #10‡	1	0xBC*(n+1)*045A4
CS3, STS #11‡	2	0xBC*(n+2)*045A4
CS3, STS #12‡	3	0xBC*(n+3)*045A4

* *n* can be CP0, CP4, CP8, or CP12

† These refer to the SDH media dependent overhead.

‡ These refer to the SDH country specific overhead.

Transmit OC-12/OC-12c Path Overhead Definitions

Table 181 Transmit SONET OC-12 and OC-12c Path Overhead Byte Addresses

Path Overhead Byte	CP# Within a Cluster	C-5 NP Address*
C2, STS #1	0	0xBC*n*045A5
C2, STS #2	1	0xBC*(n+1)*045A5
C2, STS #3	2	0xBC*(n+2)*045A5
C2, STS #4	3	0xBC*(n+3)*045A5
G1, STS #1	0	0xBC*n*045A6
G1, STS #2	1	0xBC*(n+1)*045A6
G1, STS #3	2	0xBC*(n+2)*045A6
G1, STS #4	3	0xBC*(n+3)*045A6
F2, STS #1	0	0xBC*n*045A7
F2, STS #2	1	0xBC*(n+1)*045A7
F2, STS #3	2	0xBC*(n+2)*045A7
F2, STS #4	3	0xBC*(n+3)*045A7
H4, STS #1	0	0xBC*n*045A8
H4, STS #2	1	0xBC*(n+1)*045A8
H4, STS #3	2	0xBC*(n+2)*045A8
H4, STS #4	3	0xBC*(n+3)*045A8
Z3, STS #1	0	0xBC*n*045A9
Z3, STS #2	1	0xBC*(n+1)*045A9
Z3, STS #3	2	0xBC*(n+2)*045A9
Z3, STS #4	3	0xBC*(n+3)*045A9
Z4, STS #1	0	0xBC*n*045AA
Z4, STS #2	1	0xBC*(n+1)*045AA
Z4, STS #3	2	0xBC*(n+2)*045AA
Z4, STS #4	3	0xBC*(n+3)*045AA
Z5, STS #1	0	0xBC*n*045AB
Z5, STS #2	1	0xBC*(n+1)*045AB
Z5, STS #3	2	0xBC*(n+2)*045AB
Z5, STS #4	3	0xBC*(n+3)*045AB

Table 181 Transmit SONET OC-12 and OC-12c Path Overhead Byte Addresses (continued)

Path Overhead Byte	CP# Within a Cluster	C-5 NP Address*
J1, STS #1†	0	0xBC*n*045AC to 0xBC*n*045EB
J1, STS #2†	1	0xBC*(n+1)*045AC to 0xBC*(n+1)*045EB
J1, STS #3†	2	0xBC*(n+2)*045AC to 0xBC*(n+2)*045EB
J1, STS #4†	3	0xBC*(n+3)*045AC to 0xBC*(n+3)*045EB
H1, STS #1	0	0xBC*n*045EC
H1, STS #2	1	0xBC(n+1)045EC
H1, STS #3	2	0xBC(n+2)045EC
H1, STS #4	3	0xBC(n+3)045EC

* *n* can be CP0, CP4, CP8, or CP12

† 64 bytes of J1 are buffered. All other transmit overhead in single buffered.

CP Configuration Space (SONET Specific)

In addition to the CPs thirty-two (32) (4Byte) (128Byte wide) registers used for storing receive (Rx) and transmit (Tx) SONET overhead (*Rx_SONETOH0* to *Rx_SONETOH31* registers and the *Tx_SONETOH0* to *Tx_SONETOH3)* that were mentioned before there are additional registers in the Configuration Space of the CPs. Some of them are described here.

CP Mode (SONET Specific Enable) Registers

The RxSONET and TxSONET blocks are enabled and disabled using the *SDP_Mode3* register bit [30] *RxEnable* field and the *SDP_Mode5* register bit [30] *TxEnable*. The mode registers also contain other configuration bits which control SONET scrambling, OC-12/OC-12c/OC-3c modes, automatic insertion of far-end alarms (via clearing the Manual FEBE bit) and other SONET configurations.

Table 182 SONET Specific Configuration Registers

Register Name	Purpose	Address	Details
SDP_Mode3	Collects configuration mode bits relevant for programming the RxSDP machines.	0xBC*n*0464C	See "SDP_Mode3 Register (CP Mode Configuration Function)" on page 418.
SDP_Mode5	Collects configuration mode bits relevant for programming the TxSDP machines.	0xBC*n*04654	See "SDP_Mode5 Register (CP Mode Configuration Function)" on page 424.

CP Event and Interrupt (SONET Specific Event) Registers

SONET/SDH events including as LOS, LOF, LOP, RDI-L, RDI-P, AIS-L, and AIS-P are monitored via the *SONET_Event*, *SONET_Event_Mask*, and *SDP_Mode2* registers. The *SONET_Event* register indicates whether a change in the state of any SONET event has occurred. The *SONET_Event_Mask* register is used to select which SONET events are of interest. If a SONET event occurs and that particular event has been enabled in the *SONET_Event_Mask* register, the *Event0* register bit [50] *SONETOH* field is set. This mechanism can be used to generate interrupts on state changes for various SONET defect conditions. Refer to "Event0 Register (CP Event and Interrupt Function)" on page 435.

The *SDP_Mode2* register bits [31:22] *SONET State* field detail the current defect condition (ON or OFF). Using this mechanism, defects can be monitored via interrupts on the CPRC (where interrupts are generated when the defect condition goes ON or OFF).

Table 183 SONET Specific Event Registers

Register Name	Purpose	Address	Details
SDP_Mode2	Collects SONET alarm and status information.	0xBC*n*04648	See "SDP_Mode2 Register (CP Mode Configuration Function)" on page 417.
SONET_Event	Collects together SONET event bits from the SDP's.	0xBC*n*046C0	See "SONET_Event Register (CP Event and Interrupt Function)" on page 442.
SONET_Mask	Provides mask that selects bits in the SONET_Event register for event access.	0xBC*n*046C4	See "SONET_Mask Register (CP Event and Interrupt Function)" on page 445.

SONET/SDH Monitoring Example

The following is an example of a procedure for monitoring a given defect in the C-5 NP. Specifically, to detect loss of signal (LOS) in an interrupt handler.

1. Set bit [31] *Loss of Signal* field in the *SONET_Mask* register. This allows LOS to be flagged in the *SONET_Event* register.
2. Set bit [50] *SONETOH Event* field in the *Event_Mask0* register. This allows interrupts to be generated from changes in the *SONET_Event* register.
3. Using *KsEventRegisterInterrupt(),* link the interrupt handler function with the interrupt using the C-Port CPI software. This allows the CPRC to call the SONET interrupt handler when the *SONETOH Event* field, bit [50] state changes, in the *Event_Mask0* register.
4. In the interrupt handler function, check bit [31] *Loss of Signal* field in the *SDP_Mode2* register to determine the state of the defect (on or off).
5. Set bit [31] *Loss of Signal* field in the *SONET_Event* register to clear the event, so that future LOS events are not missed.
6. Perform any notification services required to the XPRC, or write information to local DMEM.
7. Exit the interrupt handler function.
8. The CPI clears the *SONETOH Event* field bit [50] before existing, thus enabling new SONET events to generate future interrupts.

Appendix D

PCI Byte Swapping

Appendix Overview

This appendix covers the following topics:

- PCI Byte Swapping Overview
- PCI Inbound and Outbound Byte Swap Registers

PCI Byte Swapping Overview

The C-5 NP provides a byte swapping feature used to move data between the PCI Bus Little Endian environment and the C-5 NP Big Endian environment.

Most endianess issues come from the difference in the addressing (and/or byte enable assignment) and the position of bytes within a 32bit double word between the two (2) environments as shown in Figure 93.

Figure 93 Little Endian vs. Big Endian

PCI Little Endian Environment

Bits	31:24	23:16	15:8	7:0
Address	3	2	1	0
Contents	0xAA	0xBB	0xCC	0xDD

C-5 NP Big Endian Environment

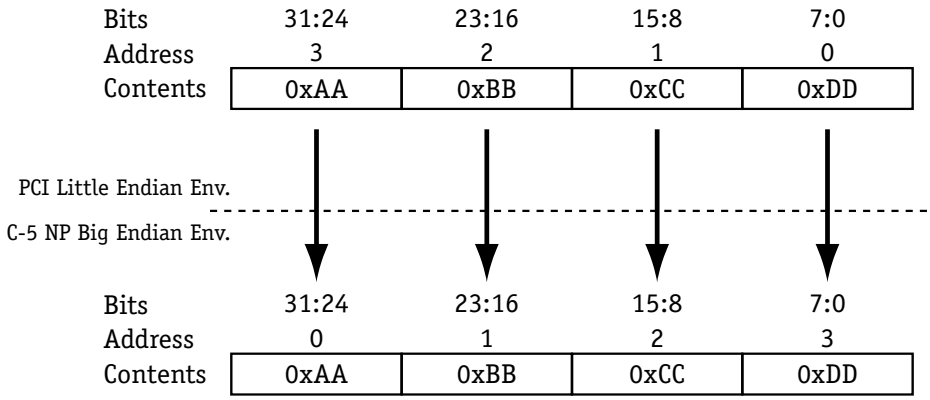
Bits	31:24	23:16	15:8	7:0
Address	0	1	2	3
Contents	0xAA	0xBB	0xCC	0xDD

These differences create a problem when 32bit double word and byte transactions pass from one environment to the other.

Default Mode

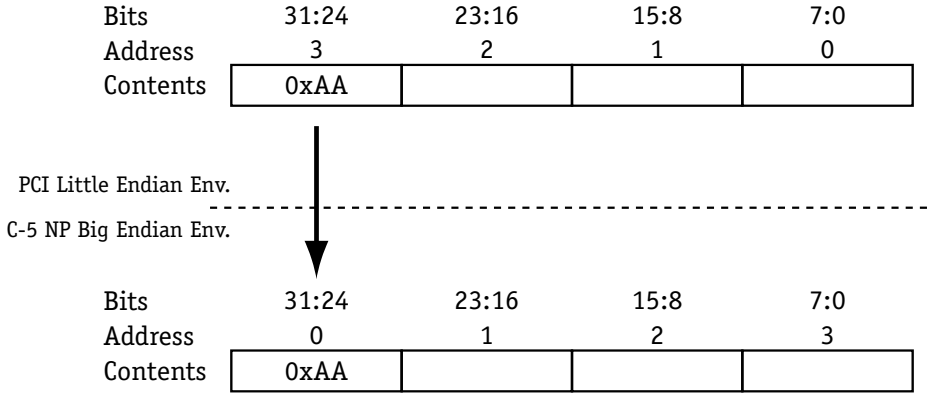
The current hardware handles this transition by maintaining the data byte positions within the 32bit double word and effectively swapping the byte address of the bytes. Figure 94 illustrates a 32bit double word write from the PCI bus into a register somewhere in the Big Endian C-5 NP environment. Notice how the data that was in bits [31:24] on the PC I bus are stored in bits [31:24] in the C-5 NP register even though the address of that byte is 3 on the PCI Bus and 0 within the C-5 NP.

Figure 94 PCI 32bit Aligned Double Word Access to C-5 NP



This means that software must be very careful with the address when performing accesses to smaller than a 32bit double word. For example, in Figure 95 the PCI performs a byte write to address 3 on the PCI Bus. Again the data on bits [31:24] on the PCI Bus arrives in bits [31:24] in the C-5 NP register; however, as far as a processing element within the C-5 NP is concerned that byte is at address 0.

Figure 95 PCI Byte Access to C-5 NP (PCI Address 3)



The same rules apply when data flows in the opposite direction. Byte position remain the same, but the addresses associated with the bytes swap as the data crosses the interface. Figure 96 illustrates a 32bit double word access and Figure 97 shows a byte access flowing from the C-5 NP to the PCI.

Figure 96 C-5 NP 32bit Aligned Double Word Access to PCI

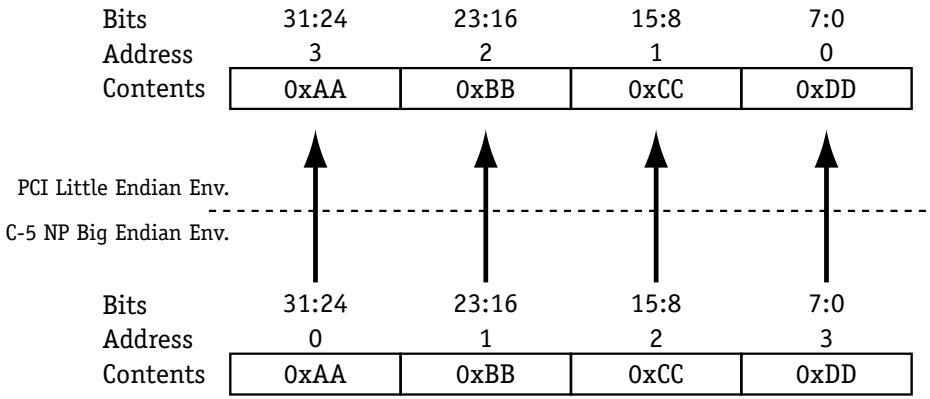
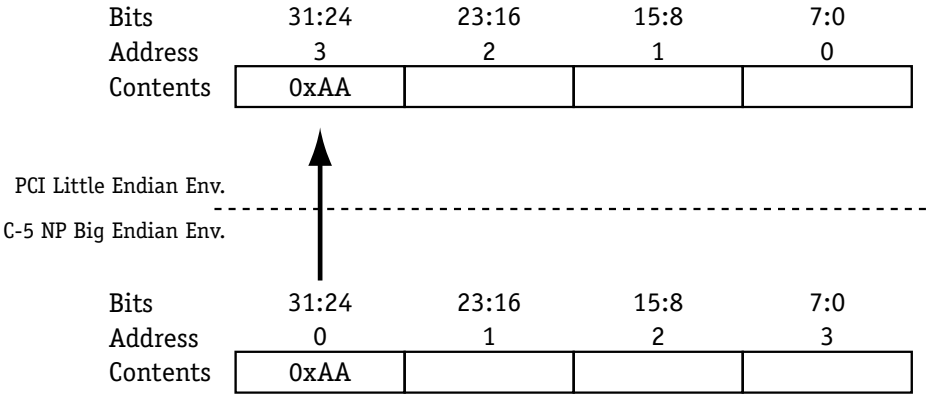


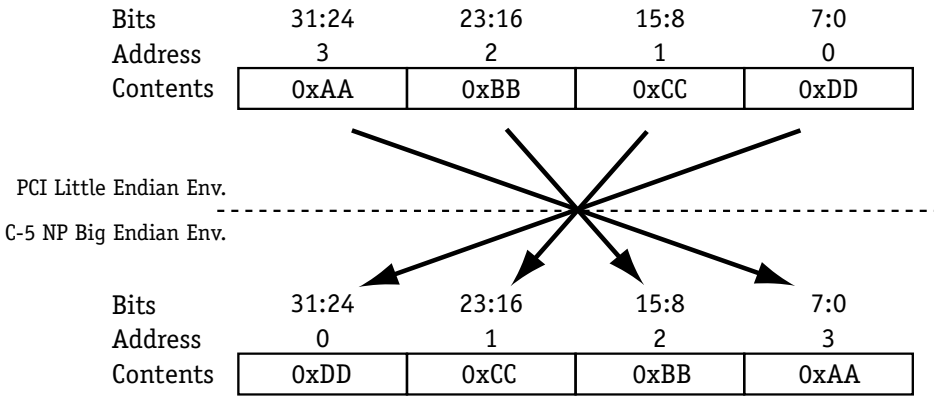
Figure 97 C-5 NP Byte Access to PCI (C-5 NP Address 0)



Byte Swapping Mode

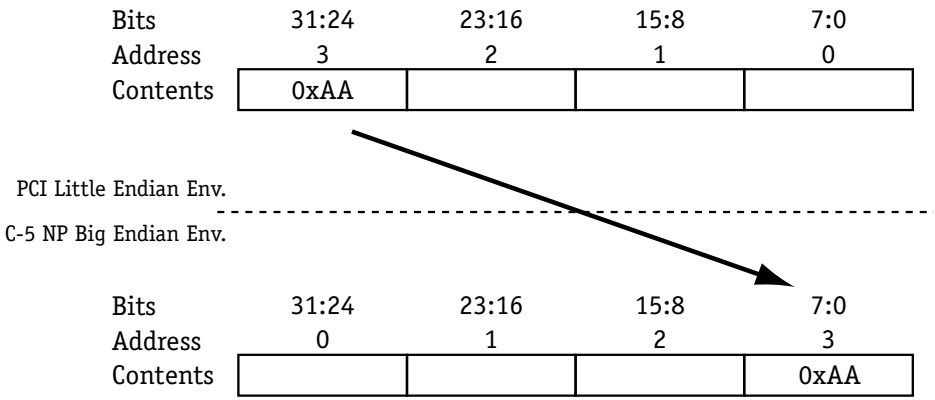
The Byte Swapping Mode is an alternative to the Default Mode that handles this transition by maintaining consistent byte addresses and swapping the data byte positions. Figure 98 illustrates a 32bit double word write from the PCI bus into a register somewhere in the Big Endian C-5 NP environment. Notice how the data that was in bits [31:24] on the PC I bus are stored in bits [7:0] in the C-5 NP register, which is the byte at the same address as that on the PCI bus. This means that if a C-5 NP internal processing element accesses the same 32bit double word, the data will be byte swapped from what the PCI originally wrote.

Figure 98 PCI 32bit Aligned Double Word Access to C-5 NP



In this mode, the software does not have to worry about the byte addresses, because they remain consistent across the interface. For example, in Figure 99 the PCI is doing a byte write to address 3 on the PCI Bus. Again the data on bits [31:24] on the PCI Bus arrive in bits [7:0] in the C-5 NP register, which is also C-5 NP internal byte address 3.

Figure 99 PCI Byte Access to C-5 NP (PCI Address 3)



The same rules apply when data is flowing in the opposite direction. Byte address remains the same, but the byte positions are swapped as the data crosses the interface. Figure 100 illustrates a 32bit double word access and Figure 101 shows a byte access flowing from the C-5 NP to the PCI.

Figure 100 C-5 NP 32bit Aligned Double Word Access to PCI

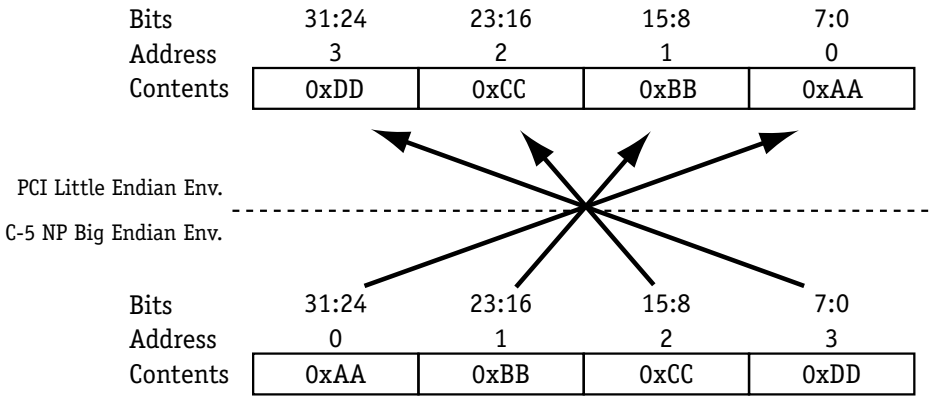
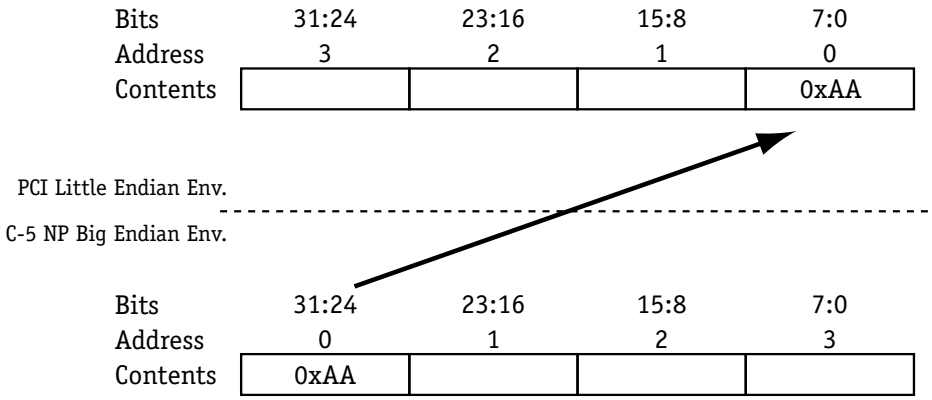


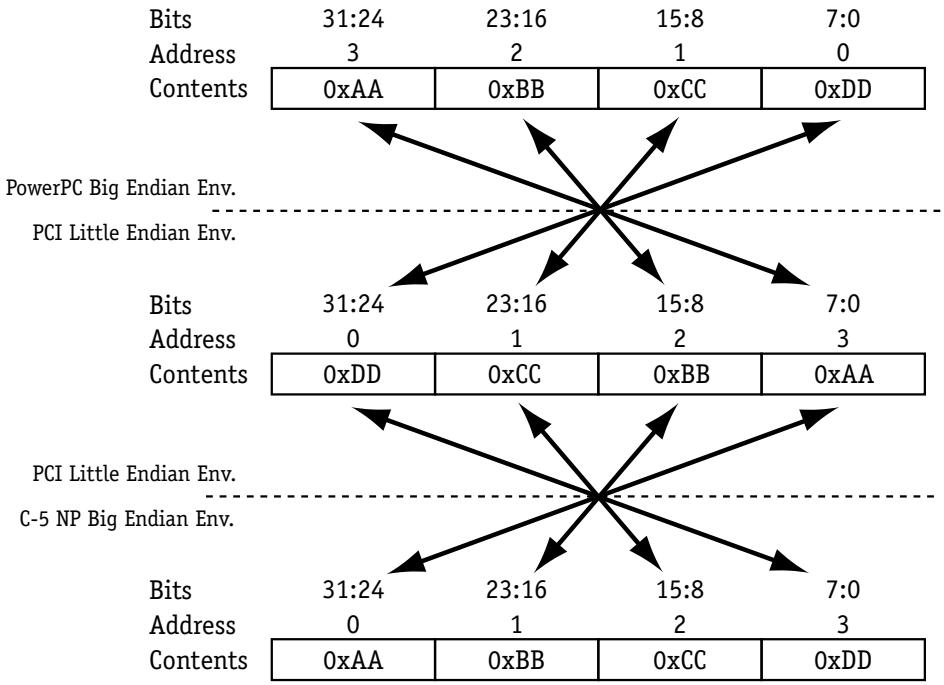
Figure 101 C-5 NP Byte Access to PCI (C-5 NP Address 0)



Primary Application Using Byte Swapping Mode

The primary application of the Byte Swapping Mode is when the host processor in the system is a Big Endian processor and the processor's bridge to the Little Endian PCI Bus performs byte swapping, as illustrated in Figure 102. Note that the two (2) byte swapping operations that occur as data passes from or to the PowerPC Big Endian Environment, over the PCI Bus and to or from the C-5 NP Big Endian Environment cancel each other out. This provides both processing environments with a consistent view of all of the data.

Figure 102 C-5 NP 32bit Aligned Double Word Access to PCI



Implementing Byte Swapping Mode

The default configuration for access to or from the C-5 NP does *not* perform byte swapping, that is, the Default Mode. The Byte Swapping Mode can be enabled for specific accesses based on programmable bits within the two (2) *PCI Inbound BARn Translation* registers, the eight (8) *PCI Outbound BARn Translation* registers, or the control registers associated with the PCI transmit and receive transfer control blocks. Table 184 on page 630 defines the details for each transaction source/target pair that uses the PCI interface. Table 185 on page 631 lists the Inbound and Outbound Bar Translation registers.

Table 184 Byte Swapping Support Specification

Source	Target	Byte Swapping Support
PCI Bus	C-5 NP PCI Config. Registers	No Byte Swapping
PCI Bus	XP Config. Registers C-5 NP Serial Bus PROM Interface CPs via Global Bus C-5 NP Ring Bus DMEM24 DMEM25	Byte swapping controlled by bit 0 (BAR0 accesses) and bit 1 (BAR 1 accesses) of the PCI Inbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI. Each enable bit control s the byte swapping for only transactions decoded by its corresponding base address register.
XP/RC	C-5 NP PCI Config Regs	No Byte Swapping
XP/RC	External PCI Space	Byte swapping controlled by bits 0 through 7 of the PCI Outbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI. Each enable bit controls the byte swapping for only transactions decoded by its corresponding base address register, where bits 0 through 7 correspond with BARs 0 through 7, respectively.
Rx XCB#24 (Scope 0)	External PCI Space	Byte swapping controlled by bit 8 of PCI Outbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI interface.
Rx XCB#24 (Scope 1)	External PCI Space	Byte swapping controlled by bit 9 of PCI Outbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI interface.
Tx XCB #24 (Scope 0)	External PCI Space	Byte swapping controlled by bit 10 of PCI Outbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI interface.
Tx XCB #24 (Scope 1)	External PCI Space	Byte swapping controlled by bit 11 of PCI Outbound Byte Swap Control register. If the bit is set to a 1, byte swapping occurs as the data passes through the PCI interface.

Table 185 Inbound and Outbound Barn Transaction Registers

Address	Register Name	Details
0xBD808040	PCI Inbound BAR0 Translation	See "PCI Inbound BAR0 Translation Register (XP PCI Configuration Function)" on page 464
0xBD808044	PCI Inbound BAR1 Translation	See "PCI Inbound BAR1 Translation Register (XP PCI Configuration Function)" on page 464
0xBD808220 to 0xBD80823C	Outbound BAR0 Transaction to Outbound Bar7 Transaction	See "Outbound BAR0 Translation Register (XP Configuration Function)" on page 472

PCI Inbound and Outbound Byte Swap Registers

The control bits for byte swapping are located in dedicated configuration registers, to allow these bits to be configured and then "forgotten". This makes it possible for more of the code to be written without knowing about the proper setting of these bits. Table 186 on page 632 lists the two (2) PCI Inbound and Outbound Byte Swap Control registers.

Table 186 PCI Inbound and Outbound Byte Swap Control Registers

Address	Register Name	Details
0xBD808050	PCI Inbound Byte Swap Control	See "PCI Inbound Byte Swap Control Register (XP PCI Configuration Function)" on page 466.
0xBD80828C	PCI Outbound Byte Swap Control	See "PCI Outbound Byte Swap Control Register (XP Configuration Function)" on page 480.

Glossary

Aggregate Channel Mode	A mode of the C-5 NP that use the CPs in parallel clusters for wider data stream processing for higher OC speeds like OC-12 or Gigabit Ethernet.
Allocate	To assign a BTag to a given requester (CP, XP or FP) from the BTags configured in BMU.
Buffer	A partitioned area of the BMU's SDRAM that holds data.
Buffer Tag (BTag)	A partitioned area of the BMU's SDRAM that is used to identify a buffer's location in a pool. It points the buffer.
Buffer Pool	A partitioned area of SDRAM that contains buffers.
Configure Queue	To send configuration information from the DMEM of a requesting processor (CP, or XP).
Constant Bit Rate (CBR)	Specifies a fixed bit rate so that data is sent in a steady stream. This is analogous to a leased line.
Content Addressable Memory (CAM)	A memory area that contains programmable content that can be searched.
Deallocate	To return a BTag from a given requester (CP, XP or FP) back to the BMU.
Descriptor	A specific type of data used for traffic forwarding.
Descriptor Buffer	A specific type of data used for traffic forwarding stored in a buffer in the QMU's external SRAM.

Dequeue	To read descriptor data from a queue in QMU's SRAM into the DMEM of the requesting processor (CP, or XP).
Dynamic Descriptor Buffer Pool	A partitioned area of the QMU's external SRAM that contains descriptor buffers.
FP Payload	A portion of segment, after the header, that is taken from or stored into a BMU buffer.
FP Protocol Data Unit (PDU)	Data to be transmitted from or received into a BMU buffer.
FP PDU ID	An identifier that allows a receiving FP to associate segments for reassembly. This ID need only be unique in the FPRx while the PDU is being reassembled; that is, a given PDU ID can be reused by a subsequent PDU. Typically, a "flow" of PDUs, transmitted from a single queue of a Network Processor, might all use the same PDU ID.
FP Segment	A basic package of FP data and header information that is transferred on fabric interface, usually a fixed size.
FP Segment Header	The beginning portion of a segment containing information used to route the segment through the fabric and allow the receiving FP to reassemble the PDU from segments.
FP Segment Type	Indicates the portion of the PDU that the segment carries. There are four types; first, middle, last, and only (first and last).
FP Scope	A set of internal hardware resources. The FPTx has 8 scopes. The FPRx can be configured for either 8 or 16 scopes.
Link-List	Tracks the free descriptor buffers, used descriptor buffers for queueing, and the location of the data (descriptor data) in queues in the SRAM.
Literals	A value written exactly as it's meant to be interpreted.
Multicast Enqueue	To write a single descriptor's data into multiple queues in the QMU's SRAM from the DMEM of the requesting processor (CP, or XP).

Multi-Use Control Blocks A group of registers that can be programmed to make data moves to/from SDRAM, the BMU, or the QMU. (WrCB0_Sys_Addr, WrCB0_Ctl, WrCB0_DMA_Addr, WrCB0_SDP_Addr; RxCB0_Sys_Addr, RxCB0_Ctl, RxCB0_DMA_Addr, RxCB0_SDP_Addr; RdCB0_Sys_Addr, RdCB0_Ctl, RdCB0_DMA_Addr, RdCB0_SDP_Addr; and TxCB0_Sys_Addr, TxCB0_Ctl, TxCB0_DMA_Addr, TxCB0_SDP_Addr).

Multi-Use Counter When a BTag is assigned to more than one target (CP, XP or FP), a counter is needed to track the multi-use BTag.

Pipeline Channel Mode A mode of the C-5 NP that links the individual CPs together for processing a single data stream to achieve higher processing speeds.

PowerX Mode A FP mode that interfaces to a Power X TeraChannel® switch fabric product.

PRIZMA Mode A FP mode that interfaces to a IBM PRIZMA-E™ or PRIZMA-EP™ switch fabric product.

Queue A FIFO that contains descriptor data.

Queue Level A index to the queue number with a port to a processor (CP, or XP). It's purpose is to copy a single descriptor to multiple queues mapped to multiple processors.

Queue Status To read a single queue's length and weight from the QMU into the DMEM of the requesting processor (CP, or XP).

Recirculation A method used to pipeline your C-5 NP. To configure the C-5 Channel Processors (CPs) RxSDP and TxSDP so that the output from the TxSDP is routed to the input of its corresponding RxSDP.

Reference Count An initial count that is entered by hardware into the 8bit counter that is used to track multi-use BTags.

Single Channel Mode A mode of the C-5 NP where the CPs operate independently of each other at full duplex and can support for example, OC-3.

Unicast Enqueue To write descriptor data into a queue in the QMU's SRAM from the DMEM of the requesting processor (CP, or XP).

User-Defined Inter-processor Message Small fixed-sized (12, 16, 24, or 32Bytes) data structures that contain user defined information. Generally, inter-processor messages are used to orchestrate control plane activities such as flow control, statistics gathering, or table maintenance.

Variable Bit Rate (VBR) A specified throughput capacity but data is not sent evenly. This is a popular choice for voice and video-conference data.

Index

D

E

F

R

S

T

V

W

X