The PETACACHE experiment at SLAC

Cluster Element Module

An I/O System On Chip

Design proposal

Document Version:	0.4
Document Issue:	3
Document Edition:	English
Document Status:	Draft - for internal distribution only
Document ID:	To be assigned
Document Date:	April 5 , 2007

Stanford Linear Accelerator Center (SLAC)
2575 Sandhill Road
Menlo Park California, 94025 USA

This document has been prepared using the Software Documentation Layout Templates that have been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics). For more information, go to http://framemaker.cern.ch/.

Abstract

To be written.

Hardware compatibility

This document assumes the following hardware revisions: TBD.

Intended audience

TBD

Conventions used in this document

Certain special typographical conventions are used in this document. They are documented here for the convenience of the reader:

- Field names are shown in bold and italics (*e.g.*, ***respond*** or ***parity***).
- Acronyms are shown in small caps (*e.g.*, SLAC or CDS).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

References

1 Preliminary Product Specification of the *Xilinx* Virtex-4 Family Overview, dated June 17, 2005

2 Data sheet for the *Cisco* SFS 7012P and 7024P *Infiniband* Server Switch

3 Data sheet for the *SilverStorm* 9024 (SDR/DDR) *Infiniband* Switch

4 Data sheet for the *DataDirect Networks* S2A9500 *Infiniband* Modular RAID Storage Networking System

5 Product Data sheet for the *Verari* VS7000i, Native *Infiniband* Storage System

6 "Proposal for LSST R & D", no date or revision given.

7 *Infiniband* Architecture Specification Volume 1, Release 1.2. Dated October 2004 (final release).

8 Optical Design Parameters, Proposed Camera Coordinate System, Proposed Numbering Systems. *Martin Nordby*, Camera Team Meeting, 1 February 2006

9 Data sheet for the *Micron* 576Mb CIO Reduced Latency (RLDRAM II), part number MT49H64M9, dated September, 2005.

10 *Xilinix* Virtex-4 Family Overview, Preliminary Product specification, dated February 10, 2006

11 Strawman design for the Science Array DAQ Subsystem (SDS), dated February 23, 2006. *Michael Huffer*

Note: For additional resources, refer to xxx on the Camera Control System:

http://www-lsst.slac.stanford.edu/Elec_DAQ/Elec_DAQ_home.htm

Click *Hardware* and then click, *List of all documents*.

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	Cluster Element Module Design proposal		
	Version:	0.4		
	Issue:	3		
	Edition:	English		
	ID:	To be assigned		
	Status:	Draft - for internal distribution only		
	Created:	February 9, 2002		
	Date:	April 5 , 2007		
	Access:	Z:\Private\pcp\CEM\v0.4\frontmatter.fm		
	Keywords:	TBD		
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	Michael Huffer		
	Written by:	Michael Huffer		
	Reviewed by:	N/A		
	Approved by:	N/A		

Document Status Sheet

Table 2 Document Status Sheet

Title:	Cluster Element Module Design proposal		
ID:	To be assigned		
Version	**Issue**	**Date**	**Reason for change**
0.0	1	6/30/2006	Initial draft

Table of Contents

List of Figures

List of Tables

Chapter 1
Overview

1.1 Introduction

TBD

1.2 The Memory Subsystem

TBD

1.3 DCR Bus Usage

TBD

1.3.1 Bus Map

All these relationships are illustrated in Figure 1:

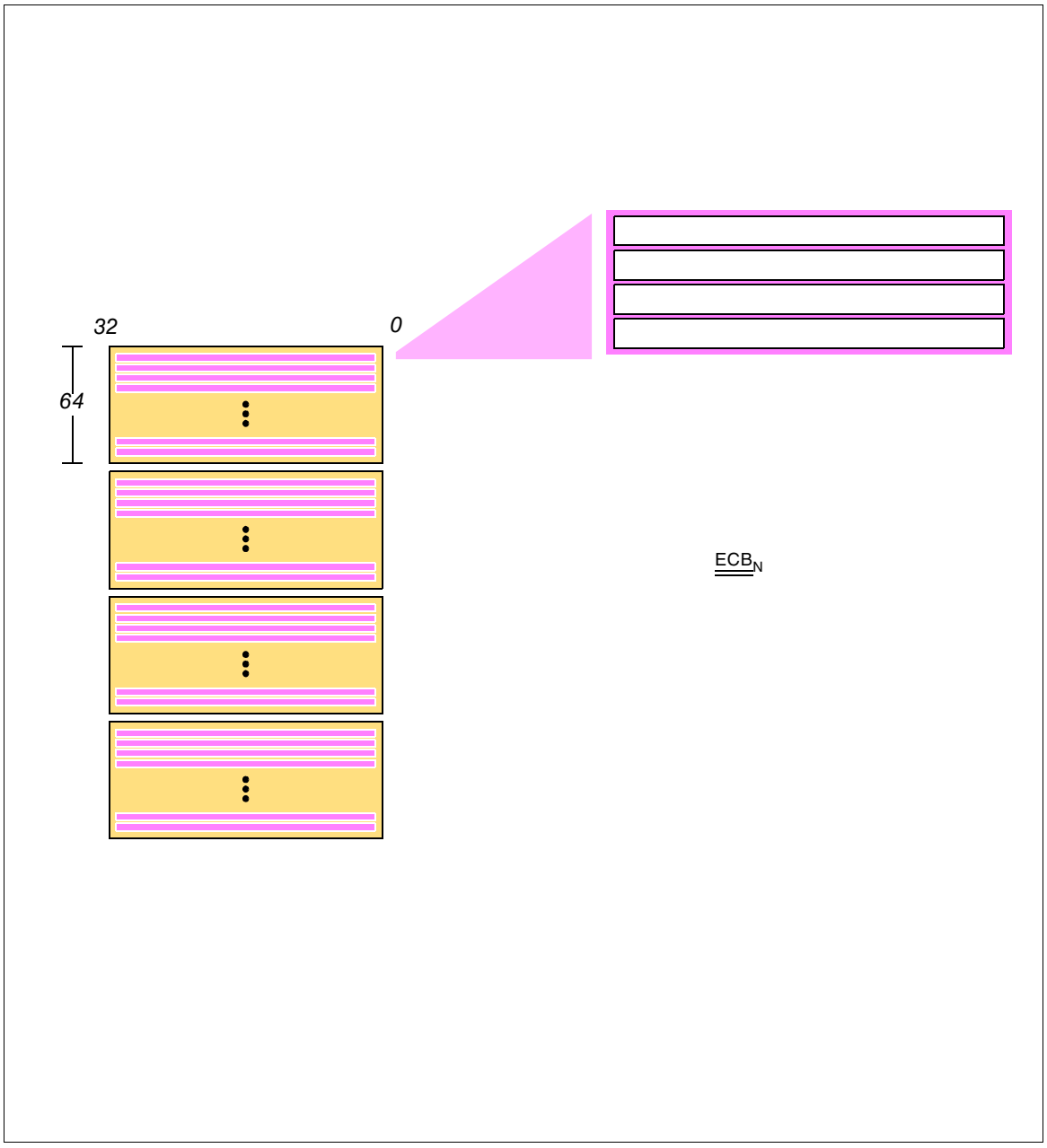

Figure 1 Data structures involved in a successful export transaction

1.3.2 Register Conventions

The application interface to the CE consists of *registers* on the processor's DCR bus (see xxx). Registers on this bus are all thirty-two (decimal) bits wide. These registers are further broken down into *fields*, where a field is specified as a bit offset and length (in number of bits). Any

field used as a boolean has a width of one bit. A value of *one* (1) is used to indicate its *set* or *true* sense and a value of *zero* (0) to indicate its *clear* or *false* sense. Field numbering (bit offsets) for registers are such that *zero* (0) corresponds to a register's Least Significant Bit (LSB) and thirty-one (31) corresponds to a register's Most Significant Bit (MSB). Bit offsets are always specified in decimal, unless otherwise noted. There are four types of generic fields:

Not defined: Undefined fields are identified as *Must Be Zer*o (MBZ) and are illustrated *greyed out*. MBZ fields will:

- read back as *zero*
- ignore writes
- reset to *zero*

Read/Write: Read/Write fields will, on *Reset*, be set to *zero*.

Selective Set and Clear (SSC): SSC fields are used where it is necessary to change one or more fields of a register and leave the remaining fields unchanged[1]. These fields will have a complementing ***Enable*** field. This field will have the same width as its corresponding SSC field. The Enable field for any arbitrary SSC field is found by shifting that field's bit offset by 16 (decimal). Enable fields are illustrated *lightly* greyed-out. These fields satisfy the following conventions:

- may only be *set*, clearing the field is ignored
- read back as *zero*

SSC fields will:

- ignore writes, *unless* their corresponding field enables are also asserted
- reset to *zero*, unless otherwise documented

Read-Only: Read-only fields are illustrated *lightly* greyed-out with their value. R*ead-Only* fields will:

- ignore writes
- reset to *zero*, unless otherwise documented

Registers on the DCR bus are read using the `mfdcr` instruction and written using the `mtdcr` instruction. The literal offset of these instructions corresponds to the register's bus address.

1. Sometimes referred to as *indivisible* read/modify/write.

Chapter 2
Configuration Memory

2.1 Introduction

TBD. This model is illustrated in Figure 2:

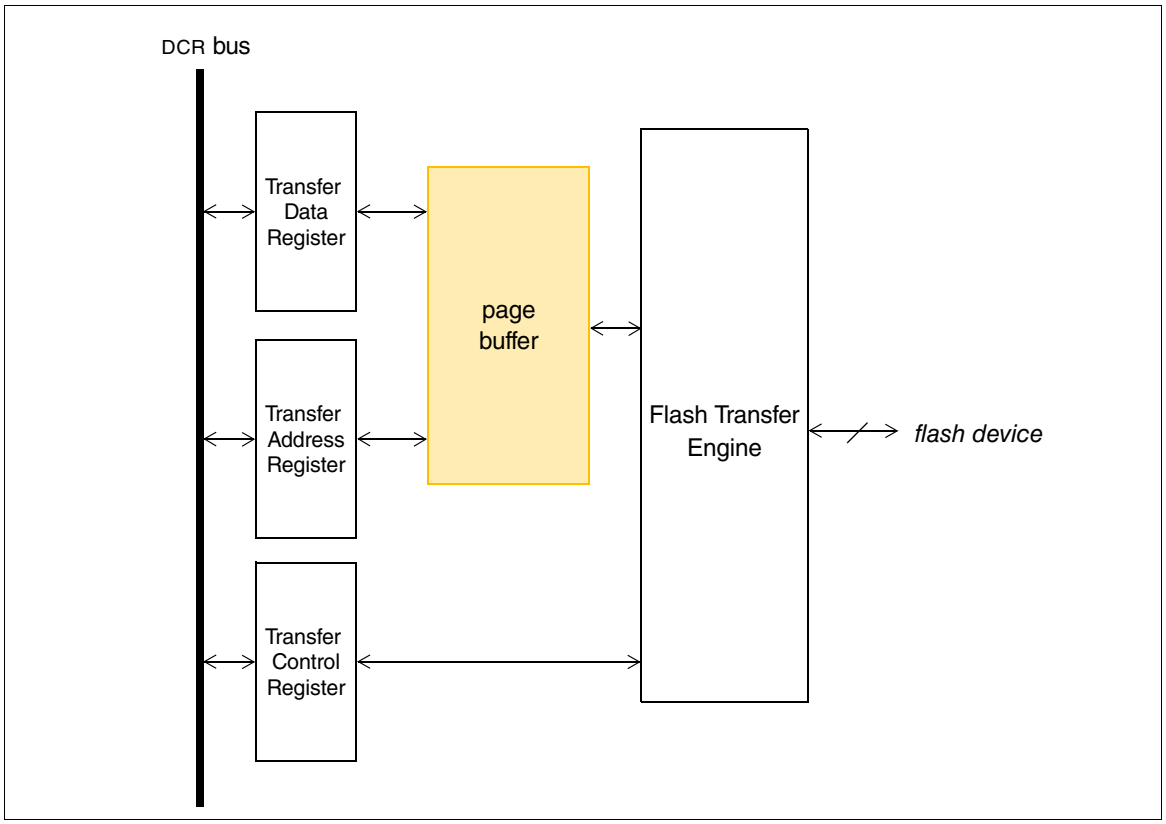

Figure 2 TBD.

2.2 Page organization and page buffer

TBD.

xxx are shown in Figure 3:

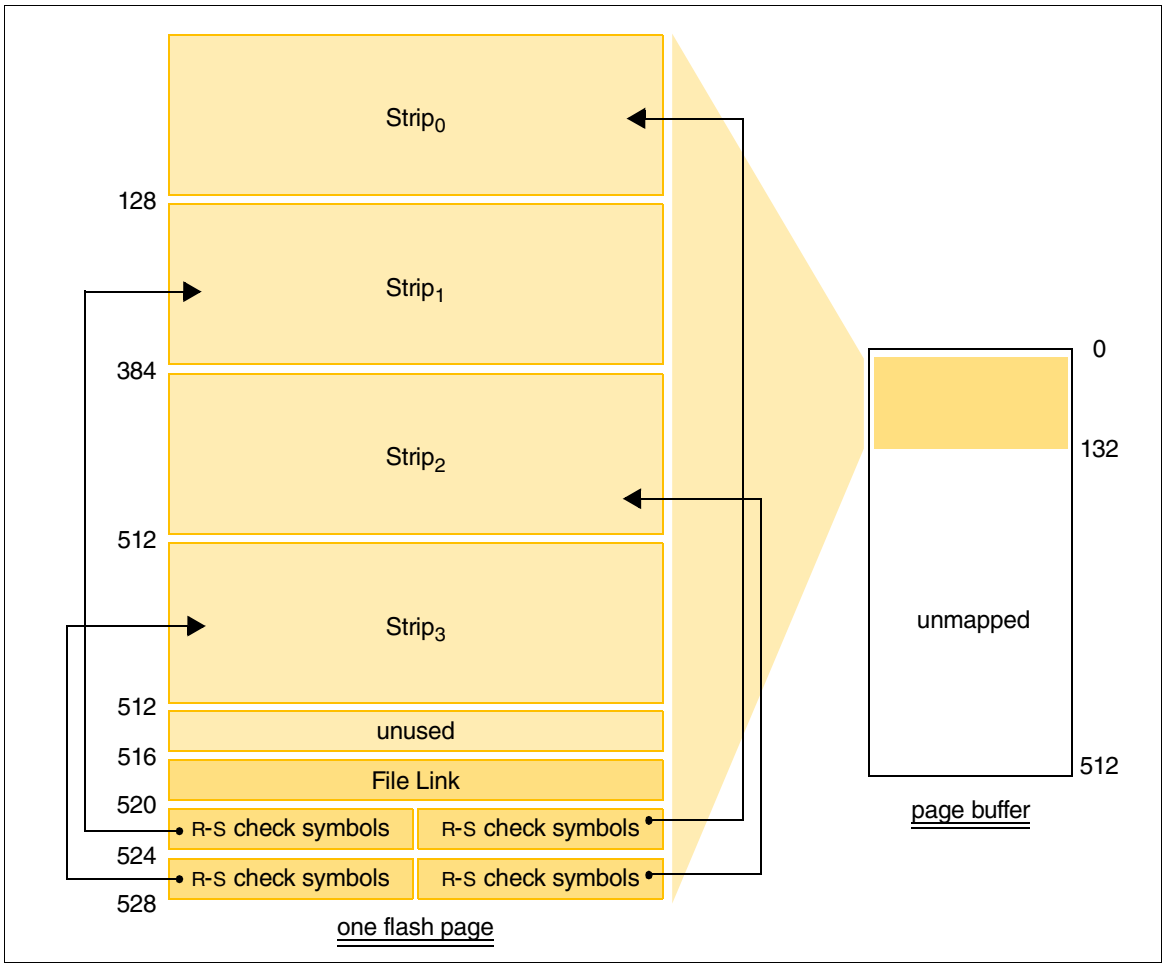

Figure 3 Entity relationships

2.3 Root block

TBD. xxx are shown in Figure 4:

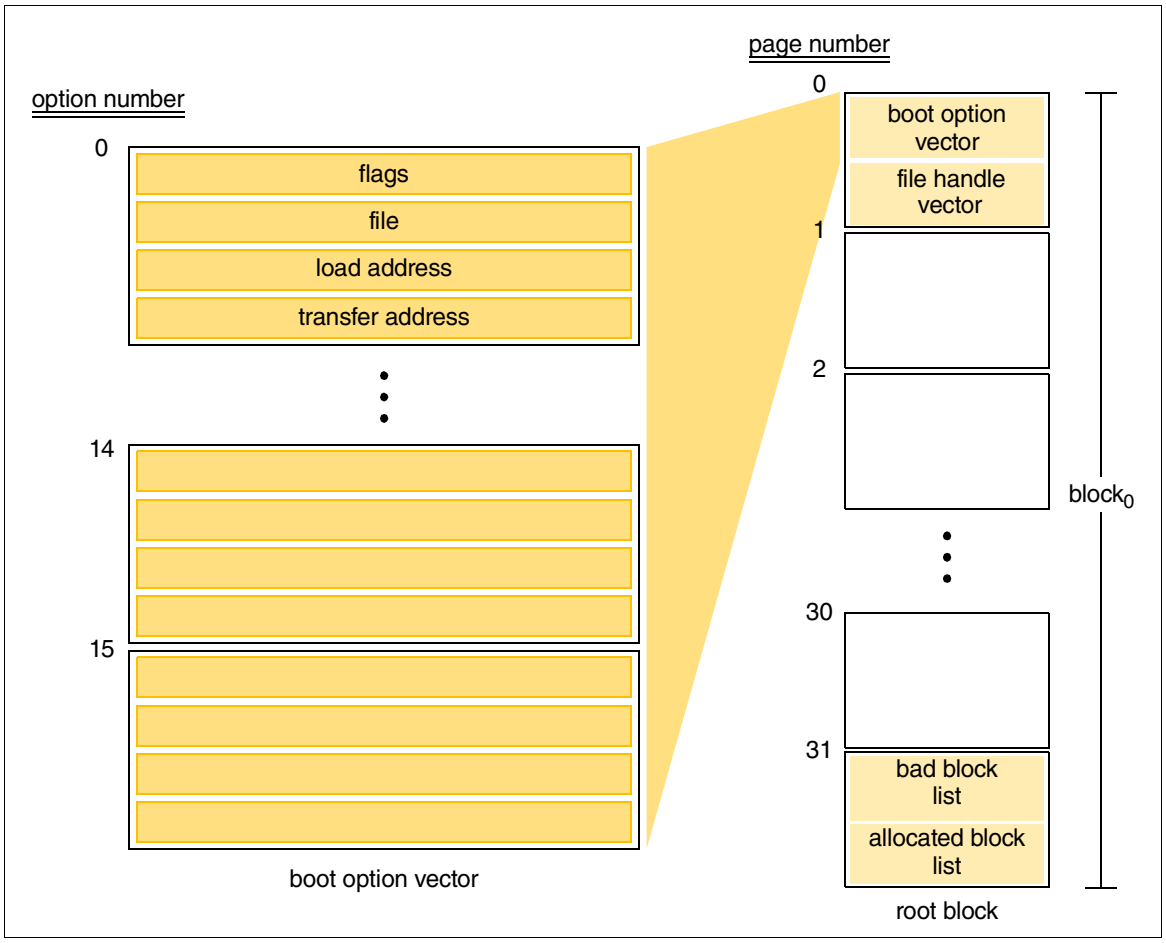

Figure 4 TBD

.

2.4 File organization

TBD. This model is illustrated in Figure 5:

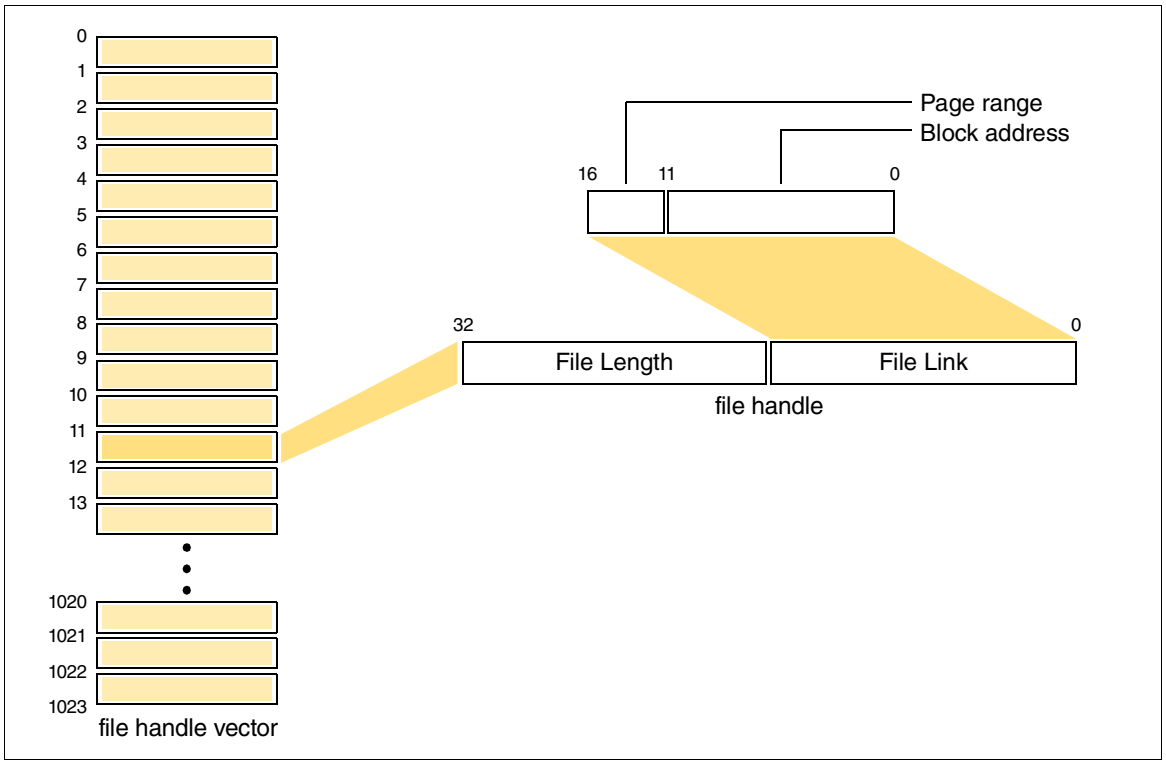

Figure 5 TBD

TBD. This model is illustrated in Figure 6:

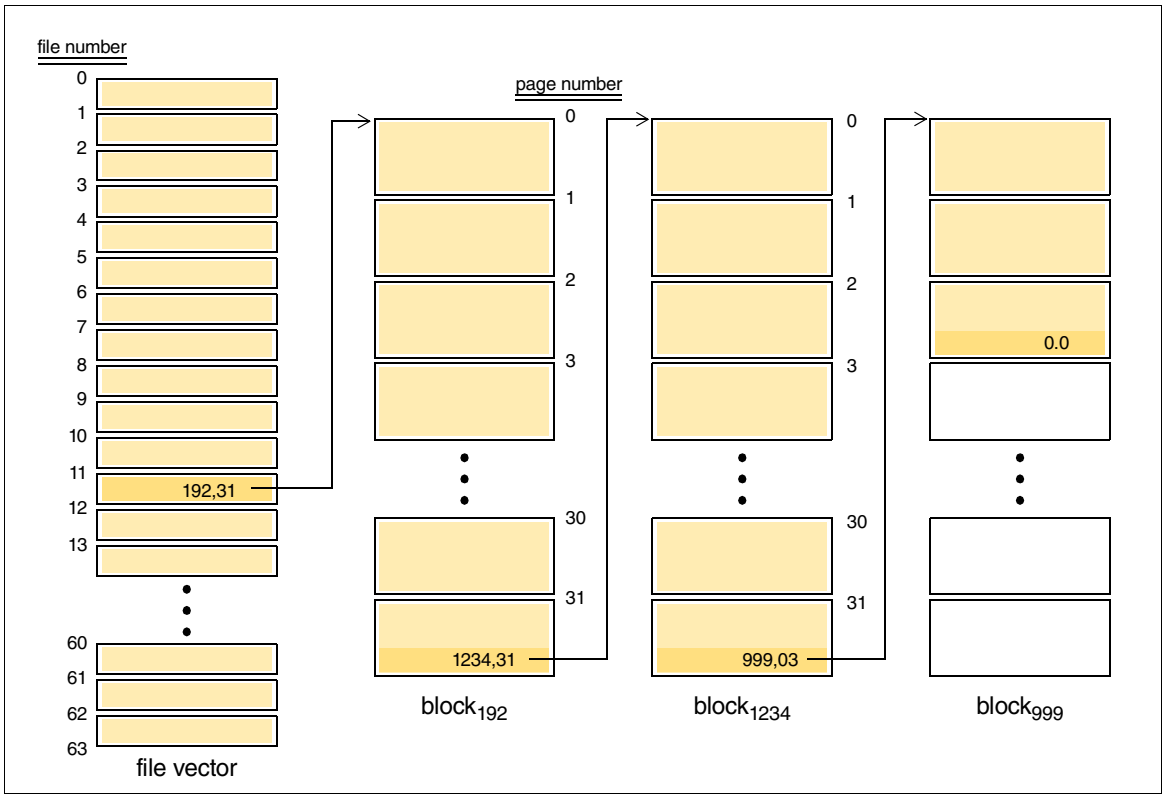

Figure 6 TBD

2.5 DCR Interface

The Back-End interface to the XXX consists principally of three registers. The location of these registers on the DCR bus relative to one another is enumerated in Table 3. The absolute location of these registers is specified as a block instantiation parameter.

Table 3 Register offsets for the PEB

Offset[1]	Name	Description
0	TCR	Transfer Control Register. See Section 2.5.1
4	TAR	Transfer address FIFO. See Section 2.5.2
8	TDR	Transfer Data REgister. Contains either the data read or the data to be written. See Section 2.5.3

1. In bytes

2.5.1 Transfer Control Register

TBD. The structure of this register is illustrated in Figure 7:

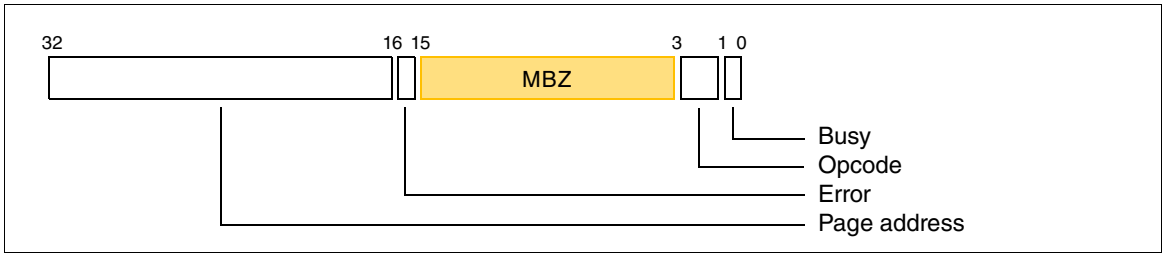

Figure 7 Transfer Control register

Where:

XXX: TBD. Note: this is a ***Selective Set and Clear*** field (see Section 2.5). In order to reset the block the appropriate field enable must also be *set* (see below).

2.5.2 Transfer Address Register

TBD. The structure of this register is illustrated in Figure 8:

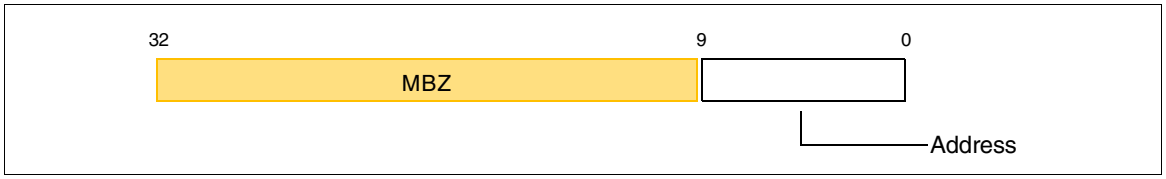

Figure 8 Transfer Address Register

2.5.3 Transfer Data Register

TBD. The structure of this register is illustrated in Figure 9:

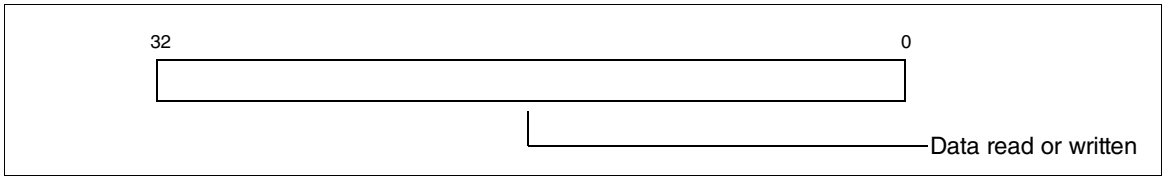

Figure 9 Transfer Data Register

Chapter 3
Processor Bootstrapping

3.1 Introduction

TBD. This model is illustrated in Figure 10:

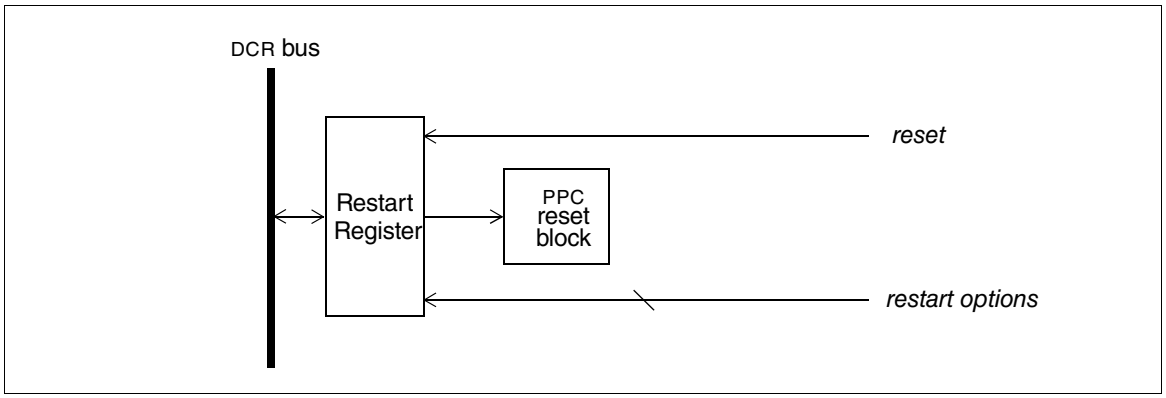

Figure 10 TBD.

TBD. This model is illustrated in Figure 10

3.2 Bootstrap Block

TBD.

3.3 DCR Interface

The Back-End interface to the XXX consists of a single register. The location of this register on the DCR bus relative to one another is enumerated in Table 4. The absolute location of these registers is specified as a block instantiation parameter.

Table 4 Register offsets for the PEB

Offset[1]	Name	Description
0	RESTART	Restart options. See xxx

1. In bytes

3.3.1 Restart Options Register

TBD. The structure of this register is illustrated in Figure 11:

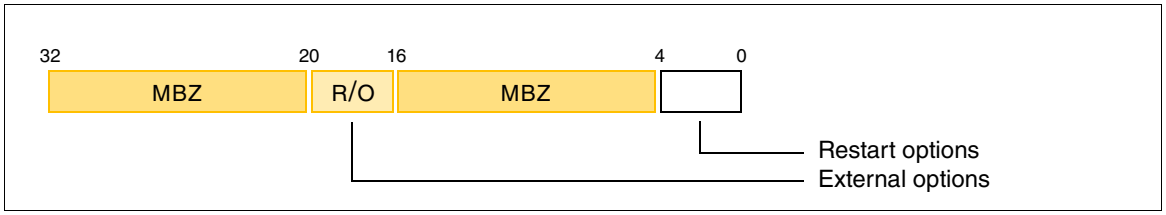

Figure 11 Restart options register

Where:

XXX: TBD. Note: this is a ***Selective Set and Clear*** field (see Section 1.3.2). In order to reset the block the appropriate field enable must also be *set* (see below).

Chapter 4
The Packet Interface Core (PIC)

4.1 Introduction

The CEM's I/O model allows external data transfers to move directly (with no processor intervention) from outside world to and from processor memory. Data is transferred from one or more protocol specific *networks*. The number and types of networks that are accessible to the CE are only constrained by the amount of resources available on the CE. While the detailed structure of data within each network is protocol specific, the model assumes data is always transferred between network and CE in units of *Packets*, as illustrated in Figure 12:

The Packet Interface Core (PIC) is a set of five different types of IP blocks designed to facility protocol implementation by providing a common interface, independent of protocol to efficiently transfer packets between networks and processor memory. The implementor interacts with these blocks through a series of *Interfaces*. The model assumes the sum of the intellectual effort to implement a specific protocol stack is partitioned into two domains:

The Protocol *Core*: The set of hardware (VHDL) instructions which interact with the PIC. The protocol core uses the PIC's *Front-End* interfaces. The specification for these interfaces is given in Chapter 5. Typically, the protocol engine is comprised of a *protocol core*, elements of the PIC's Front-End Interface, and the logic to glue these components all together.

The Protocol *Driver*: The set of software (processor) instructions which interact with the PIC. The protocol core uses the PIC's *Back-End* interfaces. The specification for these interfaces is given in Chapter 6.

The relationship between a core, its corresponding driver and the PIC is illustrated in Figure 12:

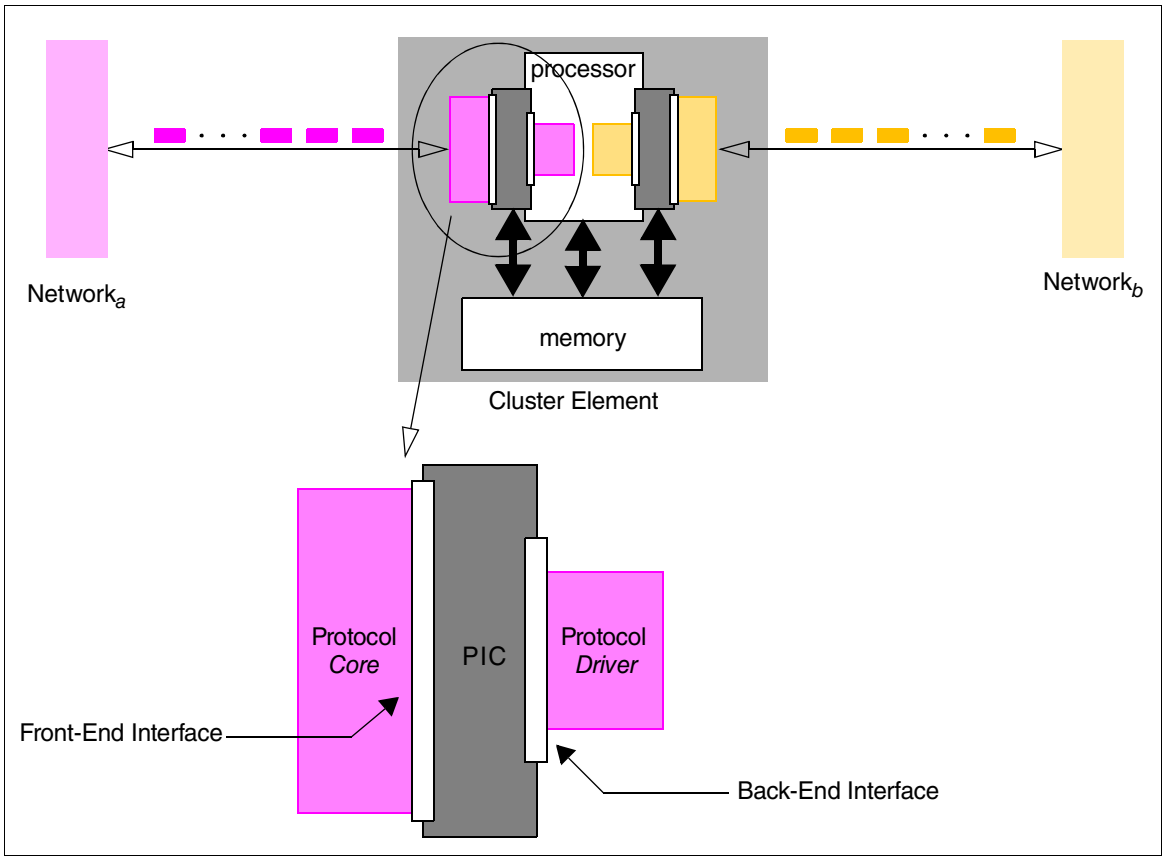

Figure 12 CEM I/O model

As a CE is realized using FPGAs from the *Virtex*-4 family (see [1]), it is assumed that a protocol core's physical layers will most likely (but, not necessarily) depend on the Multi-Gigibit-Transceivers (MGTs) provided by the FPGA. The dependencies between MGTs, potential protocol cores and drivers and the PIC are illustrated in Figure 14:

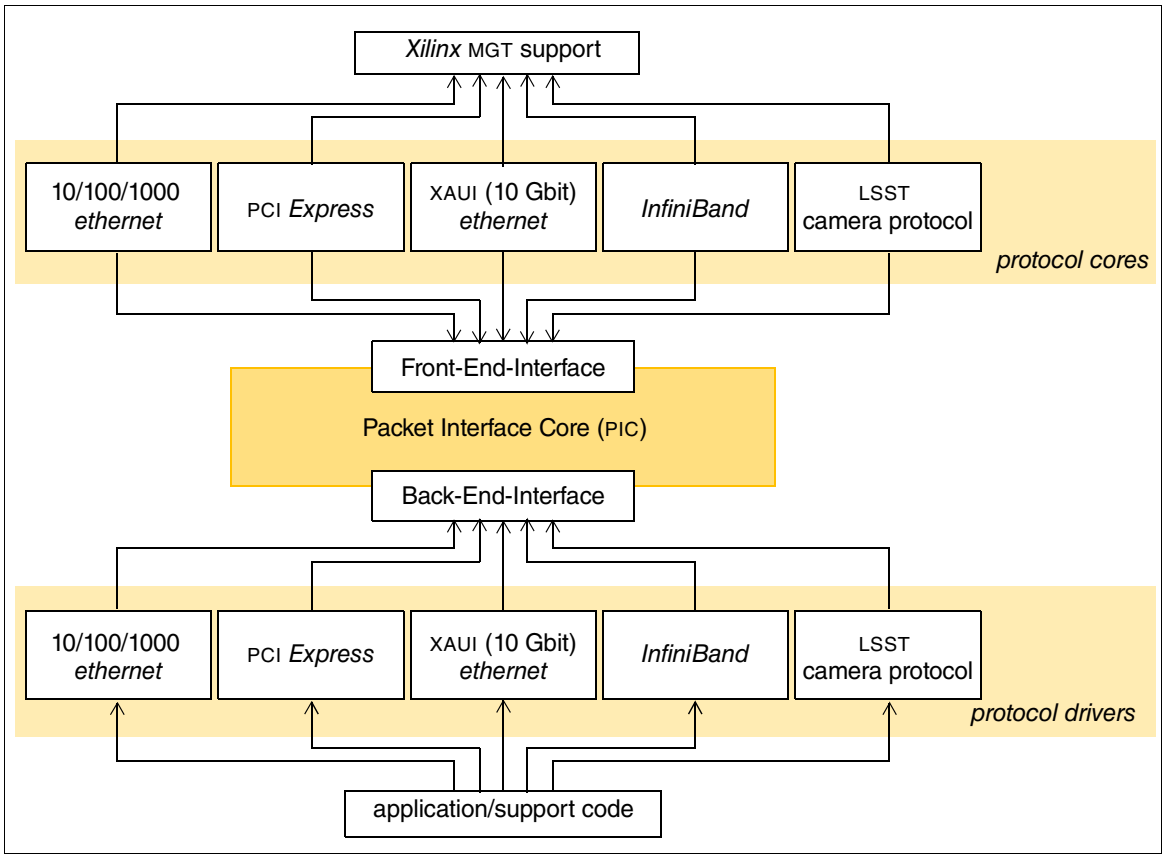

Figure 13 Entity relationships

4.1.1 Packet model

While the detailed structure of a packet is clearly protocol dependent, packets are assumed to share some common characteristics independent of protocol. In particular, all packets can be decomposed into three different components:

Header: The information necessary to describe and manage the packet content (see the ***Payload*** component below). For example, source and destination address, sequence numbers, QOS identifiers, etc... The Header always constitutes the first bytes of a packet either sent or received. The Header is always an integral number of *bytes*. For any given protocol the length of the Header is arbitrary, but *fixed*. This parameter is called the *Maximum Header Length* or MHL.

Payload: The transferred information or content. The Payload immediately follows the Header. The Payload is always an integral number of bytes. Unlike Header length, Payload length is variable. For any given protocol the payload length can vary as a function of packet. However, to facility memory management, for any one protocol, packets must have a *maximum* length. This parameter is called the *Maximum Payload Length*, or MPL.

Trailer: Packet integrity identifiers, for example, packet CRCs. The Trailer immediately follows the Payload and is an indeterminate number of bits. The model assumes the construction and use of the Trailer is outside its scope. (need some explanation here).

One of the responsibilities of the PIC is to enforce length constraints on both header and payload. This topic is discussed in 4.3. The abstract structure of a packet and its relationship with these constraints is illustrated within Figure 14:

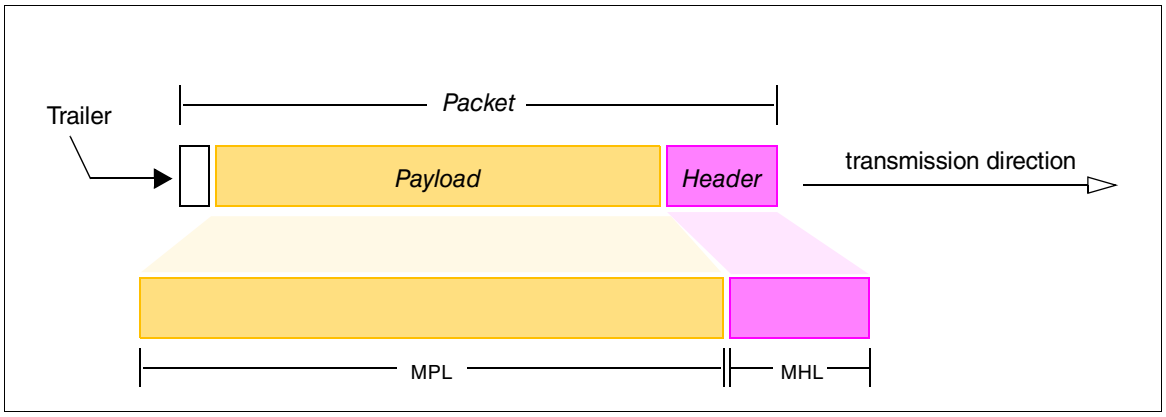

Figure 14 Packet model

4.1.2 The Transfer Model and Transactions

The PIC transfers packets in one of two directions. Any one transfer, in any one direction, is called a ***Transaction***. Therefore:

— An ***Export*** transaction transfers a single packet *from* processor memory to network. The *Export Pending* and *Export Complete* blocks are the two principal PIC blocks used for an export transaction. These blocks are introduced in Section 4.1.3.1.

— An ***Import*** transaction transfers a single packet *to* processor memory from network. The *Import Pending* and *FreeList* blocks are the two principal PIC blocks used for an import transaction. These blocks are introduced in Section 4.1.3.1.

To the protocol core the PIC represents a packet as a single contiguous stream of data, while to the protocol driver, the PIC represents the packet as two separate components located in processor memory, one for header and one for payload. These components can to be located either contiguous with respect to one another or separately, at arbitrary locations within processor memory.

For an export transaction the export blocks will gather both header and payload from memory and transfer there contents to the protocol core for transmission as a single packet. For an import transaction, the protocol core hand a packet to the import blocks where its contents will be scattered to two individual locations in memory, one for the header and one for its payload. The relationship between a packet as seen by the protocol core and its representation in memory is illustrated in Figure 15:

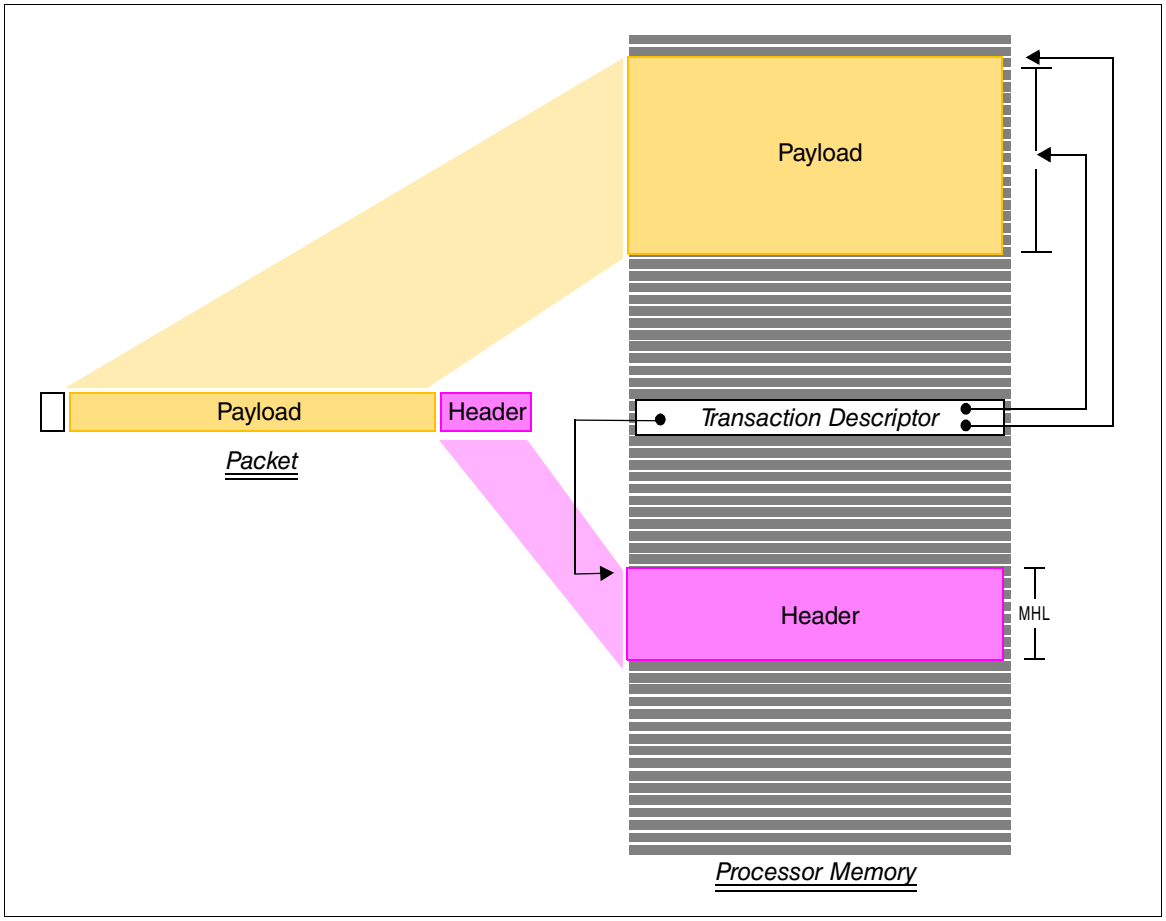

Figure 15 Packet transfer model

Note that header and payload and are no longer necessarily contiguous with respect to one another in memory. The specification of *where* to either scatter or gather the components of a packet are specified in a data structure called the *Transaction Descriptor*. This is the central structure around which much of the dynamics of packet transfer revolve and is discussed below, in Section 4.2.

4.1.3 Building blocks

From the user's perspective, the PIC consists of five different types of building blocks. Four of these blocks are used in packet transfers, one pair (the PEB and ECB) for export transactions and one pair (FLB and PEB) for import transactions. The fifth block (the ISB) is used to couple actions in the transfer blocks with interrupts in the processor. The export blocks communicate internally with each other through their *export* cross-point, while the import blocks communicate with each other through their *import* cross-point. Note; that both cross-points are actually hidden from the implementor and only come into existence when a system is instantiated (see xxx).

4.1.3.1 Transfer Blocks

Pending Export Block (PEB): Used to *initiate* export transactions. The block's Back-End Interface is used by a protocol driver to post export requests. This request is called a *Transaction Descriptor* (see Section 4.2). The block's Front-End Interface allows the protocol core to respond to those requests. When the protocol core is ready, the PEB gathers the packet's components and DMAs the assembled packet to the protocol core who transmits the packet. Once transmission is finished, the protocol core signals completion status to this block, which switches this information, through the export cross-point, to the appropriate ECB (see below). The PEB is discussed in additional detail within Section 4.7.

Export Complete Block (ECB): Used to *complete* export transactions. The block's Front-End Interface is not directly accessible to the implementor, instead, this interface is connected to the export cross-point where it waits for completion messages from one or more PEBs (see above). These potential messages are re-formatted and DMA'd to memory as Transaction Completion Descriptors (see Section 4.3). The block's Back-End Interface allows the protocol driver to wait on completed export transactions. The ECB is discussed in additional detail within Section 4.8.

Free-List Block (FLB): This block is used implicitly by a protocol core to allocate buffer space in memory for packets which are pending reception. Each buffer on the freelist describes one Transaction Descriptor. The block's Front-End Interface is not directly accessible to the implementor, instead, this interface is connected to the import cross-point where it waits for buffer requests from one or more PIBs (see below). The block's Back-End Interface is used by the protocol driver to replenish the freelist with the memory needed by the protocol core to buffer incoming packets. The FLB is discussed in additional detail within Section 4.8.3.

Pending Import Block (PIB): The block's Front-End Interface is used by a protocol core to post completed import transactions. In order to determine where to locate the received packet, the interface uses a Transfer Descriptor allocated from a FLB (see above) and obtained through the import cross-point. The protocol driver uses the block's Back-End Interface to wait on completed import transactions. The PIB is discussed in additional detail within Section 4.10.

For each CE the total number of transfer style blocks which may be instantiated[1] is sixty-four (64).

4.1.3.2 Interrupt Summary Block (ISB)

The Interrupt Summary Block (ISB) aggregates the `Event` and `Fault` signals from any one of the four different types of transfers blocks. The ISB is discussed in additional detail within Section 4.11. For each CE there will be one, and only one ISB instantiated.

4.1.4 Connecting the blocks

The number and types of blocks instantiated are specified at system build time. Each of the four transfer blocks are configurable and their exact configuration is also established at build time. Both number and type of block in any given system are driven by the specific protocol requirements. For example, if the protocol demands *QOS* (Quality-Of Service), it might require multiple pairs of blocks for different service levels or virtual channels. However, in the most trivial (and perhaps most common) case, a specific protocol would require only four blocks: A single PEB/ECB pair for packet transmission and one FLB/PEB pair for packet reception. This particular configuration is also sufficient to illustrate (see Figure 17) the connectivity established between the blocks by the PIC when a system is instantiated:

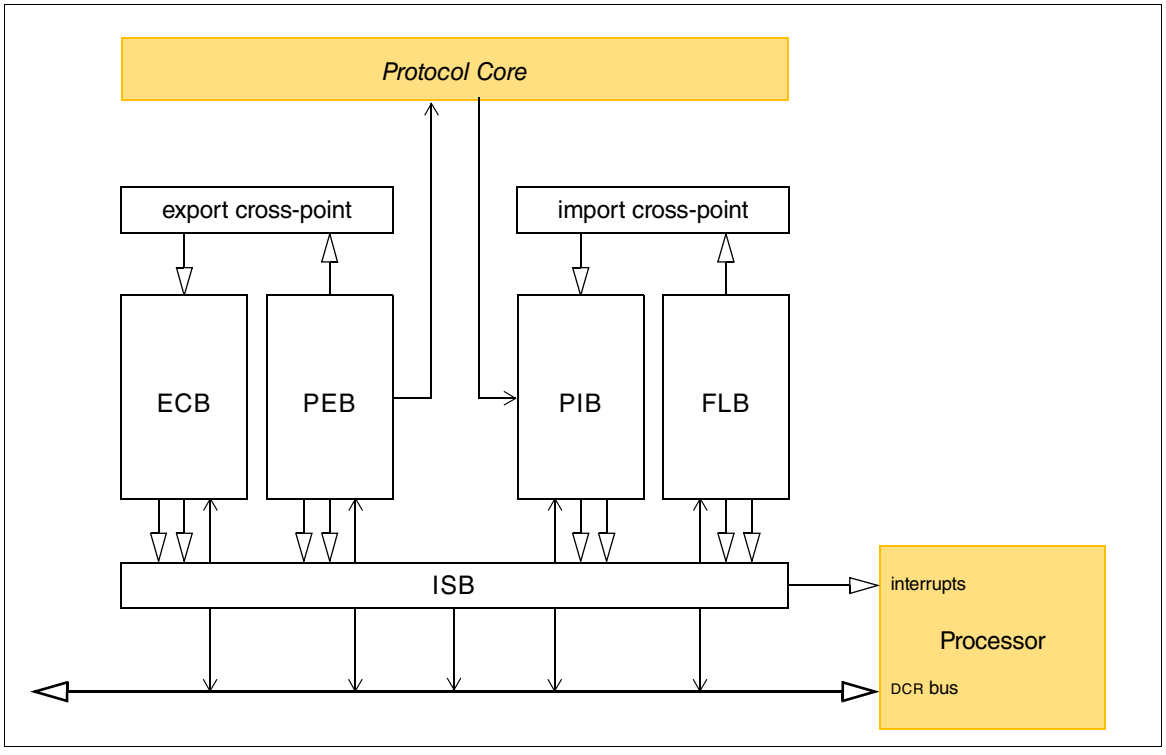

Figure 16 Connecting the blocks

1. Assuming the FPGA has sufficient resources

Therefore, to instantiate any one system, the PIC will:

- Instantiate both export and import cross-points.
- Instantiate an ISB.
- Instantiate all specified blocks using their corresponding parameters.
- Connect all the blocks Front-End interfaces to the DCR bus.
- Connect all the blocks Fault and Event signals to the ISB.
- Connect the ISB to the processor interrupt controller.
- Connect the ISB connected to DCR bus.
- Connect all ECBs and PEBs to the *export* cross-point.
- Connect all FLBs and PIBs to the *import* cross-point.

4.2 The Transaction Descriptor

The Transaction *Descriptor* is a 16-byte data structure which resides in physical memory and specifies all the information necessary to perform a single transaction. In the case of transmission the protocol driver posts this description to the PIC. In the case of reception, the order is reversed and the PIC posts this description to the protocol driver. The structure of a Transfer Descriptor is illustrated in Figure 17:

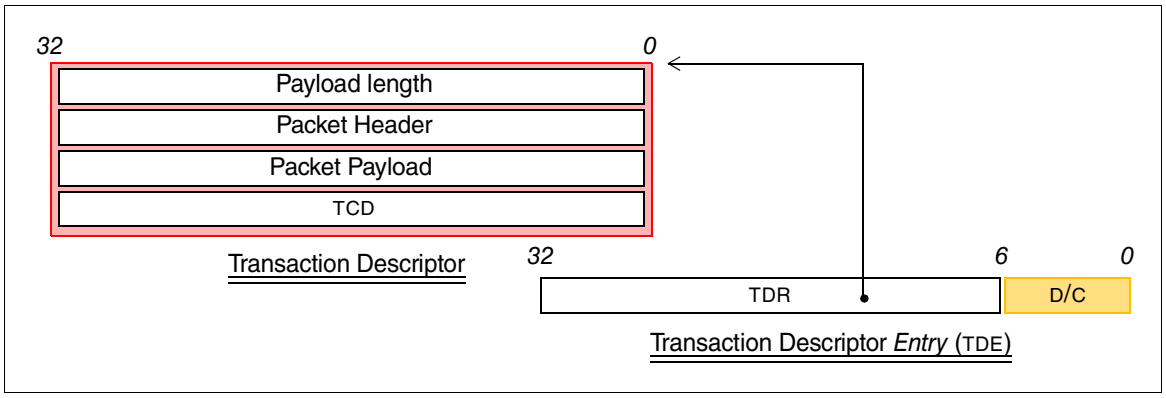

Figure 17 Transfer Descriptor

In actuality what is posted is not the descriptor, but instead its *address*. This address is called a Transaction Descriptor *Reference* or TDR. By convention all descriptors must be *64-byte* aligned, therefore the value of the low-order six bits of the address are redundant and a TDR can be (and is) represented in twenty-six bits. This allows the TDR to be encapsulated within a 32-bit word, while at the same time as reserving six of the word's bits for additional functionality. This word is called Transaction Descriptor *Entry*, or TDE. The relationship between Transaction Descriptor, TDR and TDE is illustrated within Figure 17. TDEs are the entities which are actually inserted, buffered and removed by the five different types of PIC blocks (see Section 4.4). The low-order six-bits of the TDE are overloaded and each block interprets these bits in a block specific fashion. For example, see the definition of the TDE used by the PIB

as described in Section 4.7.2. and illustrated within Figure 20. Returning to the Transaction Descriptor its fields are defined as follows:

Payload length: The size of the packet payload expressed in bytes. In a descriptor used to *export* a packet the value of this field specifies the size of the payload of the packet to be transferred from memory to network. There are two constraints on the value of this field:

— It must be less then or equal to the MPL (see Section 4.1.1).

— It must not be *zero*,(0), unless the ***IsContiguous*** field of the corresponding TDE (see Section 4.8.2) is *true*.

If either of these two constraints are violated the corresponding transaction will abort with the EDW Reason code: INVALID_PAYLOAD_LENGTH (see Table 5).

In a descriptor used to *import* a packet, the value of this field sets the threshold for the maximum number of bytes which can be received for the corresponding packet. The value of this field must never exceed the MPL (see Section 4.1.1). If either this constraint is violated or the received packets exceeds the value of this field the corresponding transaction will abort with the EDW Reason code: INVALID_PAYLOAD_LENGTH (see Table 5).

Packet Header: The base address of the packet *header* expressed in units of bytes with a value of *zero* corresponding to the first location in memory. This address must be *32-byte* aligned. In a descriptor used to export a packet (see Section 4.12), the value of this field specifies the address of the packet header to be transferred from memory to network. In a descriptor used to import a packet (see Section 4.13) this field specifies the address of the buffer which is to contain the packet header transferred from network to memory. When used in a transaction, if the address is invalid the corresponding transaction will abort with the EDW Reason code: INVALID_HEADER_ADDRESS (see Table 5). A header address is invalid if it is either not *32-byte* aligned (low-order five bits of the field are *not zero*), or any locations within the corresponding header are outside the physical address range of the CE.

Packet Payload: The base address of the packet *payload* expressed in units of bytes with a value of *zero* corresponding to the first location in memory. This address must be *32-byte* aligned. In a descriptor used to export a packet (see Section 4.12), the value of this field specifies the address of the payload of the packet to be transferred from memory to network. In a descriptor used to import a packet (see Section 4.13) this field specifies the address of the buffer which is to contain the packet payload transferred from network to memory. The address is checked for validity in all cases, *except* an export transaction in which the ***IsContiguous*** field of the corresponding TDE (see Section 4.8.2) is *true*. In all other cases, if the address is invalid the corresponding transaction will abort with the EDW Reason code: INVALID_PAYLOAD_ADDRESS (see Table 5). A payload address is invalid if it is either not *32-byte* aligned (low-order five bits of the field are *not zero*), or any locations within the corresponding payload are outside the physical address range of the CE.

TCD: The address of an *Transaction Completion Descriptor* (TCD) expressed in units of bytes with a value of *zero* corresponding to the first location in memory. This address must be *32-byte* aligned.The structure and usage of the TCD are discussed in Section 4.3. When used in a transaction, if this address is invalid a *fault* will be declared (see Section 4.6). A TCD address is invalid if it is either not *32-byte* aligned (low-order five bits of the field are *not zero*), or the location is outside the physical address range of the CE.

4.3 The Transaction Completion Descriptor (TCD)

Transactions can either succeed or fail. Their completion status is *defined* by a combination of the PIC and the appropriate protocol core and *read* and *used* by the protocol driver. And while all successes are the same, all failures are different, occurring at different points in a transaction and for different reasons. For example, the contents of the Transfer Descriptor could be invalid, or a packet received by a protocol driver could contain a CRC error. All this transaction completion information is encapsulated in a 32-byte structure, residing in physical memory called the *Transaction Completion Descriptor*, or TCD. The address of the TCD for any particular transaction is contained in a *Transaction Descriptor* described above (see Section 4.2). It is the responsibility of the protocol core and the PIC, working in concert, to deliver for each transaction a well-formed TCD. For the PIC, this is an explicit operation and for the protocol core this is an implicit operation. Implicit, because the protocol core is not exposed directly to the TCD. Instead it contributes completion status indirectly through a PIC block's Front-End interface. For example, when the protocol core finishes packet transmission, it passes its completion status by writing to the PEB's `Status` port (see Section 5.2.2.9). The structure of the TCD and its EDW is illustrated in Figure 18. The first word of the TCD is called the *Error Description Word*, or EDW, the second word is the actual transfer count and the remaining words are reserved and must be *zero*. The structure of the first word is called out separately, as the completion status value written by the protocol core is actually an EDW (see Section 5.2.5).

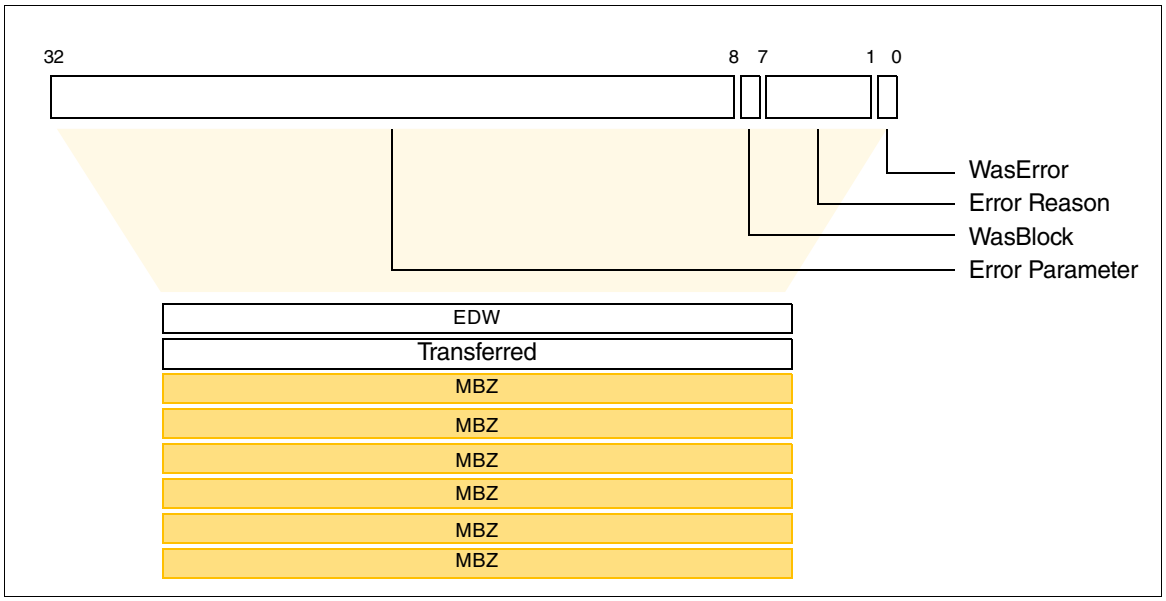

Figure 18 Transaction Completion Descriptor (TCD) and its EDW

where:

WasError: This field determines the interpretation of the values of the higher-order fields. If this field is true (*set*) the values of the remaining fields are interpreted as specified below. If this field is false (*clear*), the value of the remaining fields are unspecified.

Error Reason: A six-bit value which enumerates the reason the reported transaction failed. Reason values are protocol dependent, however, if the ***WasBlock*** field is *set* (see below), the origin of the error was the PIC itself. In such a case, the potential values for this field are enumerated in Table 5.

WasBlock: This field determines whether the error originated within the PIC (i., e., in one or more of the four interface blocks described below). In such a case, this field is true (*set*) and both the ***Error Reason*** and ***Error Parameter*** fields are specified in Table 5. If this field is false (*clear*), the value of both the ***Error Reason*** and ***Error Parameter*** fields are protocol dependent.

Error Parameter: A twenty-four-bit value which parameterizes the ***Error Reason*** field (see above). Parameter values are protocol dependent, however, if the ***WasBlock*** field is *set* (see above), the origin of the error was in the PIC itself. In such a case, the potential values for this field are enumerated in Table 5.

Transferred: This field contains the number of bytes of the packet successfully transmitted or received.

Note: While an import transaction always writes the TCD on completion, on an *export* transaction the PIC only references and writes the TCD for a transaction when it *fails*. By not writing this structure when the transaction completes successfully (hopefully, the predominate case), the PIC saves both a cache-line fetch and write, thus contributing to improving the overall system throughput. Therefore, on any successful export transaction its

TCD will retain the values it had at the time its corresponding Transaction Descriptor was posted. It is considered good practice for the protocol driver to ensure the ***wasError*** field of the EDW is false (*clear*), before the corresponding descriptor is posted.

A TCD is composed by the PIC under two circumstances:

— Whenever either a PEB or FLB discovers inconsistency within a Transfer Descriptor.

— The protocol core generates an error in processing the request.

Its important to realize that in the first case the transaction takes an early exit and the request never reaches the protocol core.

Table 5 Reason and Parameter values for errors defined by the PIC

Reason		Parameter	description
Name	Value[1]		
INVALID_PAYLOAD_LENGTH	01	None (*Must be Zero*).	Section 4.3.0.1
INVALID_HEADER_ADDRESS	02	None (*Must be Zero*).	Section 4.3.0.2
INVALID_PAYLOAD_ADDRESS	03	None (*Must be Zero*).	Section 4.3.0.3
DATA_OVER_RUN	62	None (*Must be Zero*).	Section 4.3.0.4
DATA_UNDER_RUN	63	None (*Must be Zero*).	Section 4.3.0.4

1. In decimal.

4.3.0.1 Invalid payload length

The value specified in the ***Payload Length*** field of the corresponding Transaction Descriptor was greater then MPL (see Section 4.1.1). In such a case the value of the ***Parameter*** field will be *zero* (0). Note: If more then one of the first three fields in the descriptor is invalid, this error will take precedence over the other two.

4.3.0.2 Invalid header address

The value specified in the ***Packet Header*** field of the corresponding Transaction Descriptor is invalid for one or more of the following reasons:

— Its most significant bits do not match the RLDRAM base address.

— Its not cache (*32-byte*) aligned.

— Its relative offset within the RLDRAM is such that the data contained in the header will run past the end-of-memory.

In such a case the value of the ***Parameter*** field will be *zero* (0). Note: If the payload address is also invalid, this error will take precedence.

4.3.0.3 Invalid payload address

The value specified in the ***Packet Payload*** field of the corresponding Transaction Descriptor is invalid for one or more of the following reasons:

— Its most significant bits do not match the RLDRAM base address.

— Its not cache (*32-byte*) aligned.

— Its relative offset within the RLDRAM is such that the data contained in the header will run past the end-of-memory.

In such a case the value of the ***Parameter*** field will be *zero* (0).

4.3.0.4 Data under-run

The protocol core was attempting to export a packet, however the PEB's Front-End interface could not keep up with the rate at which the protocol core would wish to transmit the packet. See Section 5.2.5 for a more detailed discussion of this error. In such a case the value of the ***Parameter*** field will be *zero* (0).

4.4 Events, Conditions and the Transaction FIFO

Each one of the four different types of transfer blocks contains a *Transaction* FIFO (see, for example, the ECB described in Section 4.8). The transaction FIFO buffers transaction oriented requests between the protocol core and its corresponding protocol driver. For any one block, one port of the FIFO appears on the block's Front-End interface, while the other port is on its Back-End Interface. These are *asynchronous* FIFO's with the capacity to store up to 512 *x* 36 bit entries[1]. The abstract function of a transaction FIFO is two-fold:

— Decouple the processor's clock domain (the Back-End) from the clock domain required to operate any arbitrary protocol core (the Front-End).

— Decouple the rate a protocol core can post/retire transactions from the rate at which its corresponding protocol driver can post/retire transactions.

When in use, a FIFO can be in one of four states:

— Not-Empty

— Almost-Empty

— Almost-Full

— Full

1. With the intent of allowing FIFO implementation to map directly to the FPGA's block memory.

While the definition of ***Almost-Empty*** and ***Full*** are block invariant, the thresholds for the ***Almost-Full*** and ***Almost-Empty*** states may be specified on a per block basis. See Section 4.4.1 below for more information. When in use, a FIFO's current state is called its *condition*. The FIFO's condition can be monitored by the protocol driver through a field within a block's CSR register (see for example, the PEB's CSR register described in Section 6.1.1). In addition, some FIFO conditions are presented to the driver implicitly. For example, the PIB returns a sentinel value when its Transaction FIFO is read while *Empty* (see Section 6.4.2). However, the most important use of a FIFO's condition is to drive a block's `Event` signal. This signal is discussed in Section 4.5.

4.4.1 FIFO parameters

When a block is instantiated the user may specify the thresholds for two conditions of a block's transaction FIFO as follows:

Almost-Empty: The threshold for asserting the ***Almost-Empty*** condition. The condition is asserted while the number of TDEs in the corresponding FIFO is less than or equal to the value of this parameter. The parameter is expressed in number of TDEs. The valid parameter values are between *zero* (0) and 510 (decimal) inclusive.

Almost-Full: The threshold for asserting the ***Almost-Full*** condition. This condition is asserted while the number of TDEs in the corresponding FIFO is greater than or equal to the value of this parameter. The valid parameter values are between one (1) and 511 (decimal) inclusive.

4.5 Events

Each block defines an `Event` signal. When the block is instantiated, this signal is connected to the ISB (see Section 4.11). The ISB maps this signal to the processor's *non-critical* interrupt signal. Thus, whenever the block asserts its `Event` signal, a non-critical interrupt is triggered in the processor. This signal is formed by using, as input, the current condition of the block's Transaction FIFO (see Section 4.4) as well as the values of the ***Event Triggers*** and ***Event Enable*** fields found in the block's CSR register (see for example Section 6.1.1). Using these inputs the logic to form this signal is as follows:

— mask the current condition against the value of the ***Event Triggers*** field

— ORs the result

— mask this result against the ***Event Enable*** field

— if the result is *true*, the `Event` signal is *asserted*

The `Event` signal remains asserted until the condition which caused its assertion is removed. For example, if the *Not-Empty* condition of the transaction FIFO is used to trigger an event, then the `Event` signal will be deasserted only while the FIFO is empty.

Events are intended to represent state transitions of a protocol engine which, in the normal course of operation, are expected to occur and are expected to be used by protocol core and driver to communicate changes in their relative states. For example, a protocol driver will need to "wait" for received packets. These packets will be delivered by the corresponding protocol core through its PIB (see Section 4.10). When the core has received a packet the PIB's transaction FIFO will be *Not-Empty*. Consequently, the protocol driver could use the *Not-Empty* event as an indication of packet arrival and could wait on its corresponding interrupt.

4.6 Faults

Each block contains one `Fault` signal. When the block is instantiated, this signal is connected to the ISB (see Section 4.11). The ISB maps this signal to the processor's *critical* interrupt signal. Thus, whenever a block asserts its `Fault` signal, a critical interrupt will be triggered in the processor.

Blocks may generate a fault through actions initiated by either the protocol driver, protocol core, or both. The description of each block enumerates the potential faults generated by that block, broken down by whether the fault originated through an action of the core or the driver. See for example, Section 4.7.3 and Section 4.7.4 which enumerate the potential faults generated by the PEB.

When the block determines from its inputs that a fault should be generated the following actions are taken by the block:

- The action (from either protocol core or driver) triggering the fault is rejected.
- If a fault is currently pending, no further action is taken.
- If a fault is *not* pending:
  - The ***Fault Pending*** field of the block's CSR is *set.*
  - The offending TDE is written to the block's ***TDE Fault*** register.

Unlike an event a fault persists (the signal remains asserted) even after the conditions which precipitated the fault are removed. Only the protocol driver can clear a pending fault. To clear a pending fault (deassert the signal) requires one of the following two actions:

- Clear the ***Fault pending*** field of the block's CSR (see, for example Section 6.1.3).
- Reset the block through its csr register.

For a description of these registers see Chapter 6. Note: Unlike an *Event* (see Section 4.5), a fault is *not* user maskable.

Note: Faults should be considered, in the normal course of operation, to never occur and their presence should be considered a logic or design error. Basically, they identify a non-recoverable error.

4.7 The Pending Export Block

The Pending Export Block (PEB) is used to initiate a packet transfer from memory to network (an *export* transaction). A protocol driver initiates a transaction by queuing a TDE to the block's transaction FIFO through the block's Back-End Interface. When the PEB dequeues that request, it signals through its Front-End Interface, the corresponding protocol core. This core then requests (through this interface) the data corresponding to the transfer, packetizes these data and transmits the resulting packet onto the appropriate network.

Typically (depending on transaction parameterization), once the packet has been transmitted, the PIC locates and uses a ECB (see Section 4.8) to signal back to the protocol driver that the transaction is finished.

A more detailed discussion of the export transaction is found in Section 4.12. A block diagram of this interface is illustrated in Figure 19:

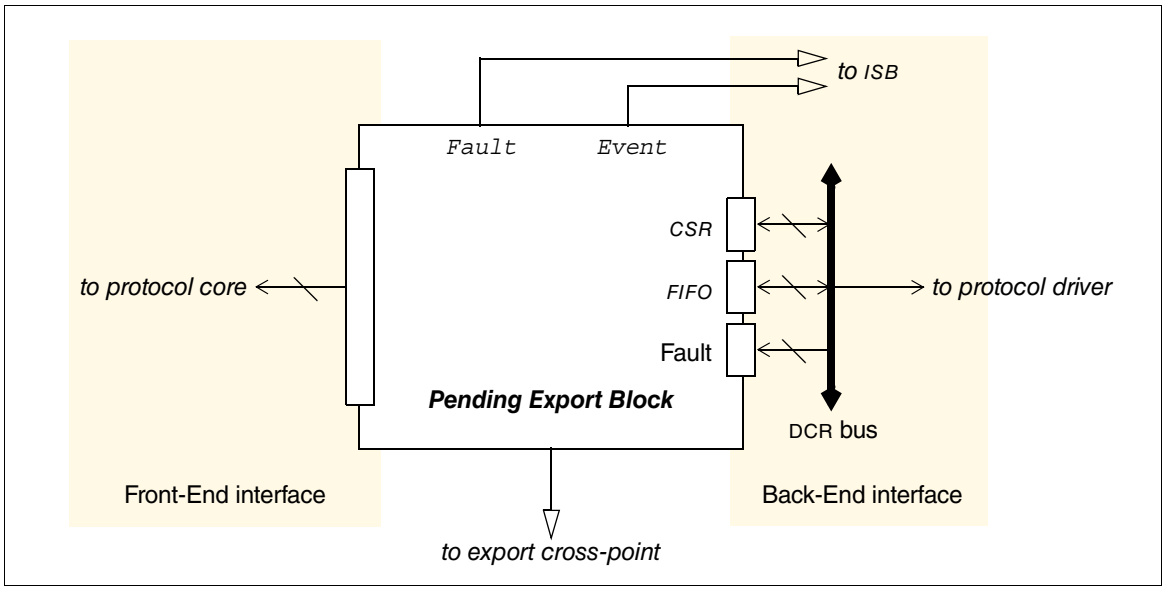

Figure 19 Block diagram of the PEB

The PEB's Front-End Interface is intended to interface with a protocol specific core. That is, a core's *Back-End* transmit interface "plugs" into the block's *Front-End* interface.

The block's Back-End Interface consists principally of an interface to the PEB's transaction FIFO. This FIFO buffers protocol driver originated TDEs. The description for the TDE associated with the PEB is found in Section 4.7.2. The FIFO's *write* port is connected to the processor's DCR

bus. A protocol driver writes to this port in order to queue transactions. The PEB controls access to the FIFO's corresponding *read* port and uses the information gained from this port to DMA both transfer descriptor and packet data. As these data arrive the Front-End Interface signals the protocol core that data is available.

The current condition of the transaction FIFO is used to drive the block's `Event` signal (see Section 4.5). When a block of this type is instantiated this signal as well as the `Fault` signal are automatically connected by the PIC to the ISB (see Section 4.11) and in this fashion FIFO conditions and/or the presence of a fault can used to trigger processor interrupts. The set of conditions which may assert the `Event` signal are configurable through the block's CSR register (see Section 6.1.1). This register is accessed through the DCR bus. A discussion of FIFO configuration is found in Section 4.11.

The specification for the Front-End Interface is found in Section 5.2. The specification for the Back-End Interface is found in Section 6.1.

The maximum number of PEBs which can be instantiated (consistent with the resources of the FPGA) can be no greater then sixteen (16).

4.7.1 Parameters

When a block of this type is instantiated (see Section 4.1.4) various block attributes must be assigned. These include:

Block number This is a small enumeration from *zero* (0) to 15 (decimal) which is used to identify the instantiated block. If more then one block of this type is instantiated, each block must be assigned a unique number. This value is used to establish:

- The relative offset into DCR space for the registers of its Front-End Interface. See Section 1.3.1 for a discussion of how this identifier maps to an absolute address on the DCR bus.
- The relative register number and bit offset in both the event and fault source registers of the ISB (see Section 6.5).

Almost-Full threshold: This parameter specifies the threshold for the ***Almost-Full*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Almost-Empty threshold: This parameter specifies the threshold for the ***Almost-Empty*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Header length: The length (expressed in bytes) of the header of any packet posted to the block for export. This parameter is referred to as The *Maximum header Length* or MHL (see Section 4.1.1). The value of this parameter must be at least *one* (1). Its maximum value must be no larger then the physical address of the CE. To ensure transfer efficiency, when possible, the value of this parameter should be an even number of *cache-lines* (32 bytes).

4.7.2 The PEB's Transaction FIFO

The PEB's transaction FIFO is also called the *pending export* FIFO. This is an *asynchronous* FIFO with the capacity to store up to 512 (decimal) entries. As implied by its name this FIFO buffers pending export transactions. An export transaction is initiated by a protocol driver, processed by the protocol core and completed by the PIB. The protocol driver writes to and the PIB reads from this FIFO. *What* is either inserted or removed from this FIFO is a 32-bit word called a Transfer Descriptor *Entry* (TDE). The structure of a TDE for the PEB is illustrated in Figure 20:

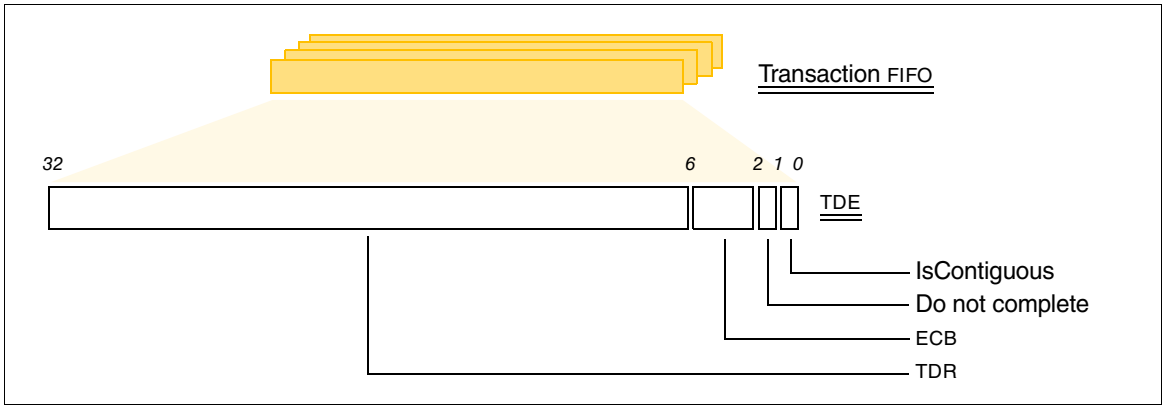

Figure 20 Structure of the TDE as used by the PEB

where:

IsContiguous: This field specifies the in-memory relationship between the end of a packet's header and the beginning of its payload. If this field is true, (*set*) the PEB assumes the packet's header and payload are contiguous with respect to one another. In such a case, the PEB ignores the ***Packet Payload*** field of the corresponding Transfer Descriptor (see below and Section 4.2). If this field is false (*clear*), the PEB assumes the packet is *not* contiguous and that the location of the packet's payload is specified by the ***Packet Payload*** field of the corresponding Transfer Descriptor. This option is useful only when *both* the following constraints are present:

— The in-memory representation of the packet must be contiguous. Note: this is by definition and implication true, when the size of the payload is *zero* (see Section 4.2).

— The packet's payload boundary *cannot* be cache-line aligned as the packet header length is not evenly divisible by a cache-line (32 bytes).

In all other instances, this field should be *false*.

Do not complete: This field specifies, whenever a transaction completes *successfully*, whether the PEB posts a completion message to an ECB (see Section 4.8.2). The target of this completion message is determined by the value of the ***ECB*** field described below. If this field is true (*set*), the PEB will *not* post a completion message. If this field is false (*clear*), the PEB *always* post a completion message. This option is useful when export and import transactions are coupled. For example, consider the case of a command/response type protocol. In such a case, the reception of a response packet could be used to complete not only the export (command), but also its

paired import (response) transaction. In turn, this reduces the traffic through the ECBs of the system and the corresponding interrupts that traffic might generate. *Note*: The usage of this field is *only* applicable to successful transactions. If a transaction fails, the value of this field is ignored and a completion message is *always* posted to the specified ECB.

ECB: A small enumeration which specifies the ECB (see Section 4.8) used to complete the transaction. These values were established when the ECBs of the CE were instantiated (see Section 4.1.4 and Section 4.8.1). Note: As failed transaction are *always* posted to an ECB, this field must contain a legitimate value independent of the value of the ***Do not complete*** field described above.

TDR: A reference to the corresponding Transfer Descriptor (see Section 4.2). In actuality, the 26 most significant bits of the Transfer Descriptor's address as all Transfer Descriptors must be aligned on a *64-byte* boundary and therefore, the low-order six address bits are assumed *zero*.

4.7.3 Faults triggered by a Protocol *Core*

These faults occur in the interface between the protocol *core* and its corresponding PEB. In such a case, the TDE corresponding to the fault is not available to the PIC and therefore the `TDE_Fault_Register` (see Section 6.1.3) is simply *cleared*. See the description of the PEB's Front-End interface (Section 5.2.2) for more information.

4.7.3.1 Data Pipeline *Empty*

The protocol core attempted to retrieve data from the block when the block had no data to deliver. Formally, this corresponds to the protocol core asserting the block's `Advance_Data_Pipeline` signal while its `Data_Available` signal is *not* asserted.

4.7.3.2 Status Pipeline *Full*

The protocol core completes a packet transfer and attempts to communicate completion status to the PEB. However, the PEB does not have sufficient buffering to accept that completion status. Most likely, the PEB is not able to pass off this completion status to the ECB specified in the transaction. Formally, this corresponds to the protocol core asserting the block's `Advance_Status_Pipeline` signal while its `Status_Full` signal is asserted.

4.7.4 Faults triggered by a Protocol Driver

These faults occur in the interface between the protocol *driver* and its corresponding PEB. See the description of the PEB's Back-End interface (Section 6.1) for more information.

4.7.4.1 Invalid Transfer Descriptor

The protocol driver posts a TDE to the block's transaction FIFO. The PEB fails to de-reference the Transaction Descriptor pointed to by that TDE. This fault could occur for a variety of reasons:

— The ECB number specified by the TDE exceeds the last ECB defined for the system.

— The TDR does not correspond to a a valid location in processor memory.

— The TCD address specified in the Transaction descriptor is neither *32-byte* aligned nor corresponds to a valid location in processor memory.

4.7.4.2 No such ECB

The protocol driver posts a TDE whose ***ECB*** field points to an ECB not instantiated in the CE.

4.7.4.3 Export FIFO *Full*

The protocol driver posts a TDE to the block's transaction FIFO. The FIFO is *Full*.

4.8 The Export Complete Block

The Export Complete Block (PEB) is used to communicate completed export transactions from the PIC to a protocol driver. Therefore, the block's Front-end Interface is connected directly to the PIC's EMB (see Section 4.7) and is opaque to the user. As such, there is a close partnership between ECB and PEB (see Section 4.7). When a protocol engine completes the transfer of a packet it notifies its corresponding driver by calling on the services of the ECB's Front-End Interface. A driver "waits" on export completion by using the services of the ECB's Back-End Interface. A detailed discussion of the export transaction is found in Section 4.12. A block diagram of this interface is illustrated in Figure 21:

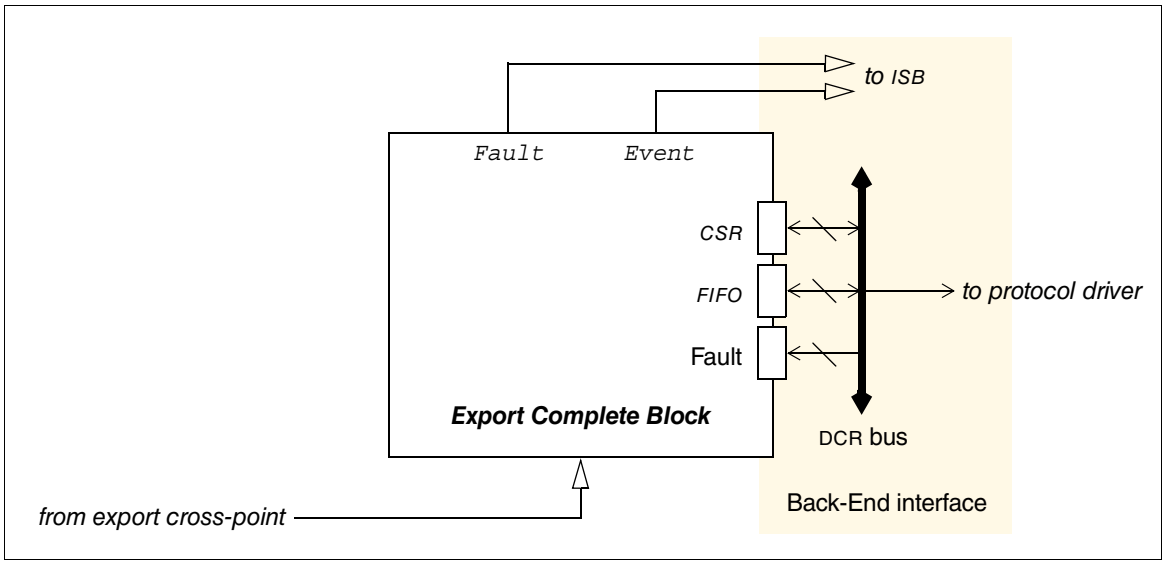

Figure 21 Block diagram of the ECB

The ECB does *not* have a Front-end Interface. Instead, as part of the PIC it is automatically connected to the export cross-point (see Section 4.1.4). This connection allows any of the instantiated PEBs of the CE to draw on the services of any of the instantiated blocks of this type.

The Back-End Interface consists principally of an interface to the block's transaction FIFO. This FIFO buffers completed export requests. The FIFO's *read* port is connected to the processor's DCR bus. A protocol driver reads from this port to process completed transactions. The FIFO's corresponding *write* port is implicitly driven by one or more of the instantiated PEBs through the export cross-point.

The current condition of the transaction FIFO is used to drive the block's `Event` signal (see Section 4.5). When a block of this type is instantiated this signal as well as the `Fault` signal are automatically connected by the PIC to the ISB (see Section 4.11) and in this fashion FIFO conditions and/or the presence of a fault can used to trigger processor interrupts. The set of conditions which may assert the `Event` signal are configurable through the block's CSR register (see Section 6.2.1). This register is accessed through the DCR bus. A discussion of FIFO configuration is found in Section 4.11.

The specification for the Back-End Interface is found in Section 6.2.

The maximum number of ECBs which can be instantiated (consistent with the resources of the FPGA) can be no greater then sixteen (16).

4.8.1 Parameters

When a block of this type is instantiated (see Section 4.1.4) various block attributes must be assigned. These include:

Block number This is a small enumeration from *zero* (0) to 15 (decimal) which is used to identify the instantiated block. If more then one block of this type is instantiated, each block must be assigned a unique number. This value is used to establish:

- The relative offset into DCR space for the registers of its Front-End Interface. See Section 1.3.1 for a discussion of how this identifier maps to an absolute address on the DCR bus.
- The relative register number and bit offset in both the event and fault source registers of the ISB (see Section 6.5).
- The identifier used the protocol driver to specify the completion block to be used by when completing an import transaction (see Section 4.7.2).

Almost-Full threshold: This parameter specifies the threshold for the ***Almost-Full*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Almost-Empty threshold: This parameter specifies the threshold for the ***Almost-Empty*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

4.8.2 The ECB's Transaction FIFO

The *Export complete* FIFO is an *asynchronous* FIFO with the capacity to store up to 512 entries. As implied by its name this FIFO contains *completed* export transactions. An export transaction was initiated by a protocol driver writing and completed by the ECB. The ECB writes to and the protocol driver reads from this FIFO. *What* is either inserted or removed from this FIFO is a 32-bit word called a Transfer Descriptor *Entry* (TDE). The structure of a TDE for the ECB is illustrated in Figure 22:

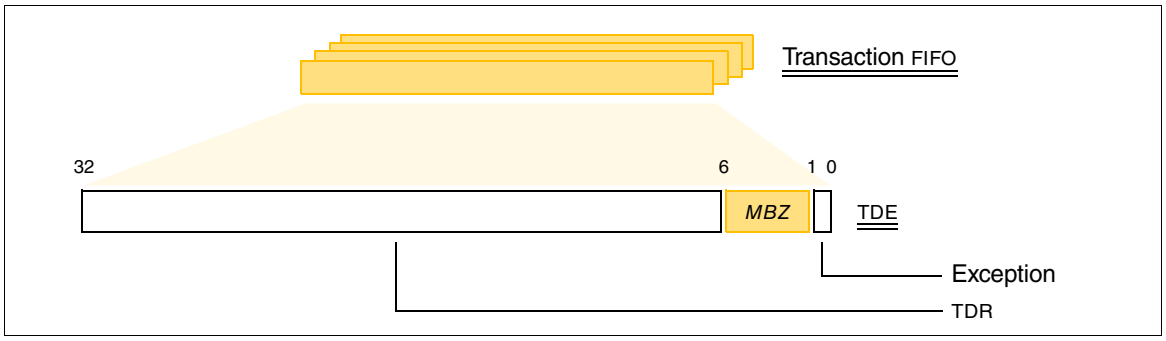

Figure 22 Structure of the TDE as used by the ECB

where:

Exception: This field describes whether or not the transaction completed with error. If the field is true (*set*) the transaction completed with error. In such a case, the TCD corresponding to the Transaction Descriptor will be written as described in Section 4.3. I f the field is false (*clear*) the transaction completed without error. In such a case, the TCD corresponding to the Transaction Descriptor is left unchanged.

TDR: A reference to the Transaction Descriptor (see Section 4.2) corresponding to the completed transaction. In actuality, the 26 most significant bits of the Transaction Descriptor's address as all Transaction Descriptors must be aligned on a *64-byte* boundary and therefore, the low order six address bits are assumed *zero*. The value of this field is the address of the Transaction Descriptor corresponding to the transaction request posted to a PEB by the protocol driver (see Section 4.7).

4.8.3 Faults triggered by a Protocol *Core*

None.

4.8.4 Faults triggered by a Protocol Driver

None.

4.9 The Free-List Block

he Free-List Block (FLB) is used implicitly by a protocol core to allocate memory for incoming packets. These packets were transferred from network to memory as part of an *import* transaction. It is the function of the corresponding protocol driver to keep the freelist full using the FLB's Back-End Interface. The FLB is always used in conjunction with a PIB whose function is to actually perform the necessary transfer using the buffer provided by the FLB. (see Section 4.10). A more detailed discussion of the import transaction is found in Section 4.13. A block diagram of this interface is illustrated in Figure 23:

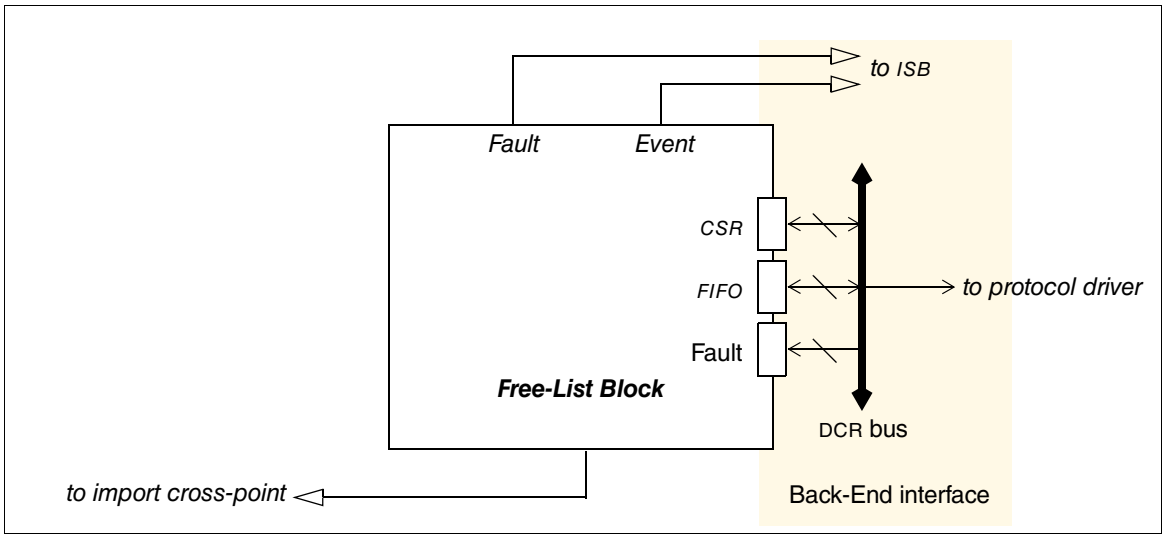

Figure 23 Block diagram of the FLB

The FLB does *not* have a Front-end Interface. Instead, as part of the PIC it is automatically connected to the import cross-point (see Section 4.1.4). This connection allows any of the instantiated ECBs of the CE to draw on the services of any of the instantiated blocks of this type.

The Back-End Interface consists principally of an interface to the block's transaction FIFO. This FIFO contains the set of buffers currently available for allocation by a protocol core. The FIFO's *write* port is connected to the processor's DCR bus. A protocol driver writes to this port to replenish the freelist. The FIFO's corresponding *read* port is implicitly sampled by one or more of the instantiated PIBs through the import cross-point.

The current condition of the transaction FIFO is used to drive the block's `Event` signal (see Section 4.5). When a block of this type is instantiated this signal as well as the `Fault` signal are automatically connected by the PIC to the ISB (see Section 4.11) and in this fashion FIFO conditions and/or the presence of a fault can used to trigger processor interrupts. The set of conditions which may assert the `Event` signal are configurable through the block's CSR register (see Section 6.3.1). This register is accessed through the DCR bus. A discussion of FIFO configuration is found in Section 4.11.

The description for the TDE associated with the FLB is found in Section 4.9.2.

The specification for the Back-End Interface is found in Section 6.3.

The maximum number of FLBs which can be instantiated (consistent with the resources of the FPGA) can be no greater then sixteen (16).

4.9.1 Parameters

When a block of this type is instantiated (see Section 4.1.4) various block attributes must be assigned. These include:

Block number This is a small enumeration from *zero* (0) to 15 (decimal) which is used to identify the instantiated block. If more then one block of this type is instantiated, each block must be assigned a unique number. This value is used to establish:

— The relative offset into DCR space for the registers of its Front-End Interface. See Section 1.3.1 for a discussion of how this identifier maps to an absolute address on the DCR bus.

— The relative register number and bit offset in both the event and fault source registers of the ISB (see Section 6.5).

— The identifier used the protocol core to specify the freelist from which it should allocate a Transaction Descriptor when importing a packet (see Section 5.2).

Almost-Full threshold: This parameter specifies the threshold for the ***Almost-Full*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Almost-Empty threshold: This parameter specifies the threshold for the ***Almost-Empty*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Header length: The length (expressed in bytes) of the header of any packet posted to the block for export. This parameter is referred to as The *Maximum header Length* or MHL (see Section 4.1.1). The value of this parameter must be at least *one* (1). Its maximum value must be no larger then the physical address of the CE. To ensure transfer efficiency, when possible, the value of this parameter should be an even number of *cache-lines* (32 bytes).

4.9.2 The FLB's Transaction FIFO

The *Freelist* FIFO is an *asynchronous* FIFO with the capacity to store up to 512 entries. As implied by its name this FIFO contains pending import transactions. An import transaction is initiated by a driver and completed by the FLB. The driver writes to and the FLB reads from this FIFO. *What* is either inserted or removed from this FIFO is a 32-bit word called a Transfer Descriptor *Entry* (TDE). The structure of a TDE is illustrated in Figure 24:

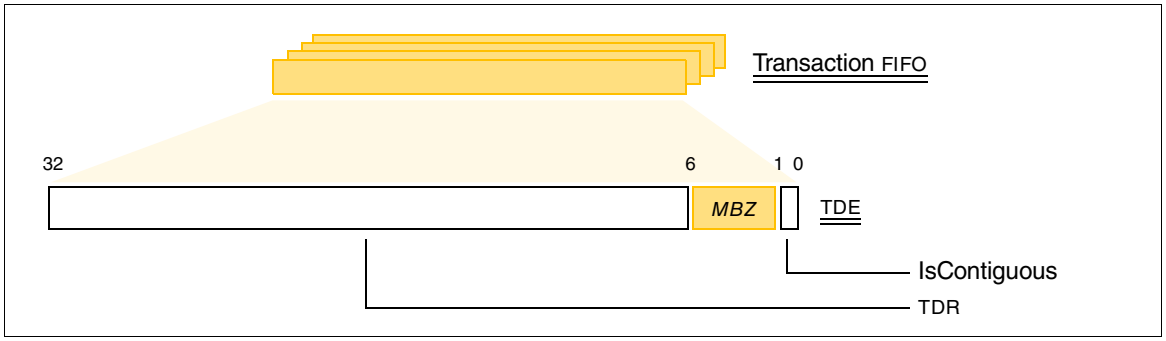

Figure 24 Structure of the TDE as used by the FLB

where:

IsContiguous: This field specifies the in-memory relationship between the end of a packet's header and the beginning of its payload. If this field is true, (*set*) the PIB assumes the packet's header and payload are contiguous with respect to one another. In such a case, the PIB ignores the ***Packet Payload*** field of the corresponding Transfer Descriptor (see below and Section 4.2). If this field is false (*clear*), the PIB assumes the packet is *not* contiguous and that the location of the packet's payload is specified by the ***Packet Payload*** field of the corresponding Transfer Descriptor. This option is useful only when *both* the following constraints are present:

— The in-memory representation of the packet must be contiguous. Note: this is by definition and implication true, when the size of the payload is *zero* (see Section 4.2).

— The packet's payload boundary *cannot* be cache-line aligned as the packet header length is not evenly divisible by a cache-line (32 bytes).

In all other instances, this field should be *false*.

TDR: A reference to the corresponding Transfer Descriptor (see Section 4.2). In actuality, the 26 most significant bits of the Transfer Descriptor's address as all Transfer Descriptors must be aligned on a 64-bit boundary and therefore, the low order six address bits are assumed *zero*.

4.9.3 Faults triggered by a Protocol *Core*

None.

4.9.4 Faults triggered by a Protocol Driver

These faults occur in the interface between the protocol *driver* and its corresponding FLB. See the description of the FLB's Back-End interface (Section 6.3) for more information.

4.9.4.1 Invalid Transfer Descriptor

The protocol driver posts a TDE to the block's freelist. The FLB fails to de-reference the Transaction Descriptor pointed to by that TDE. This fault could occur for a variety of reasons:

— The TDR does not correspond to a a valid location in processor memory.

— The TCD address specified in the Transaction descriptor is neither *32-byte* aligned nor corresponds to a valid location in processor memory.

4.9.4.2 Freelist *Full*

The protocol driver posts a TDE to the block's freelist. The FIFO is *Full*.

4.10 The Pending Import Block

The Pending Import Block (PIB) is used by a protocol core to transfer a packet from network to memory. (an *import* transaction). The location in memory to transfer the packet was determined by a FLB (see Section 4.10). The driver uses the Back-End Interface to "wait" on packet arrivals. A more detailed discussion of the import transaction is found in Section 4.13. A block diagram of this interface is illustrated in Figure 25:

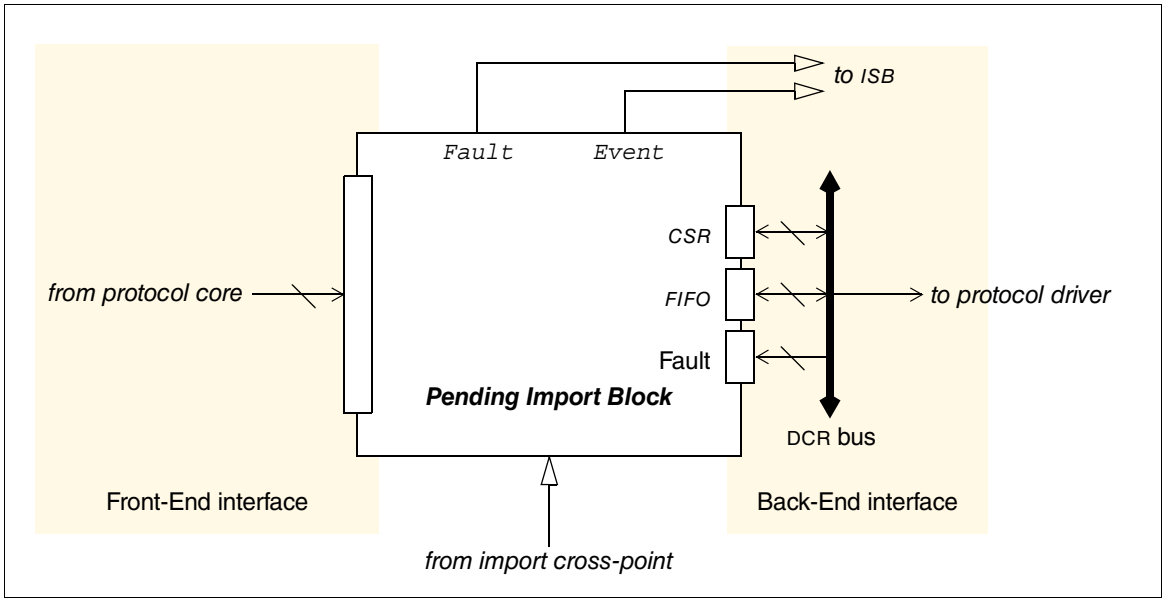

Figure 25 Block diagram of the PIB

The PEB's Front-End Interface is intended to interface with a protocol specific core. That is, a core's *Back-End receive* interface "plugs" into the block's *Front-End* interface.

The block's Back-End Interface consists principally of an interface to the block's transaction FIFO. This FIFO buffers pending, completed import transactions. The transactions are represented as TDEs. The description for the TDE associated with the PIB is found in Section 4.10.2. The FIFO's *read* port is connected to the processor's DCR bus. A protocol driver reads from this port in order to process completed import transactions. The PEB controls access to the FIFO's corresponding *write* port and uses the port to signal transaction completion from the protocol core using its Front-End Interface. This interface is responsible to DMA both transfer descriptor and received packet data.

The current condition of the transaction FIFO is used to drive the block's `Event` signal (see Section 4.5). When a block of this type is instantiated this signal as well as the `Fault` signal are automatically connected by the PIC to the ISB (see Section 4.11) and in this fashion FIFO conditions and/or the presence of a fault can used to trigger processor interrupts. The set of conditions which may assert the `Event` signal are configurable through the block's CSR register (see Section 6.4.1). This register is accessed through the DCR bus. A discussion of FIFO configuration is found in Section 4.11.

The description for the TDE associated with the PIB is found in Section 4.10.2.

The specification for the Front-End Interface is found in Section 5.3. The specification for the Back-End Interface is found in Section 6.4.

The maximum number of PIBs which can be instantiated (consistent with the resources of the FPGA) can be no greater then *sixteen* (16).

4.10.1 Parameters

When a block of this type is instantiated (see Section 4.1.4) various block attributes must be assigned. These include:

Block number This is a small enumeration from *zero* (0) to 15 (decimal) which is used to identify the instantiated block. If more then one block of this type is instantiated, each block must be assigned a unique number. This value is used to establish:

- The relative offset into DCR space for the registers of its Front-End Interface. See Section 1.3.1 for a discussion of how this identifier maps to an absolute address on the DCR bus.
- The relative register number and bit offset in both the event and fault source registers of the ISB (see Section 6.5).

Almost-Full threshold: This parameter specifies the threshold for the ***Almost-Full*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

Almost-Empty threshold: This parameter specifies the threshold for the ***Almost-Empty*** condition to be asserted by the block. See Section 4.4.1 for a discussion of this parameter.

4.10.2 The PIB's Transaction FIFO

The *Import Pending* FIFO is an *asynchronous* FIFO with the capacity to store up to 512 entries. As implied by its name this FIFO contains *completed* import transactions. An import transaction was initiated by a driver and completed by the PIB. The PIB writes to and the driver reads from this FIFO. *What* is either inserted or removed from this FIFO is a 32-bit word called a Transfer Descriptor *Entry* (TDE). The structure of a TDE is illustrated in Figure 26:

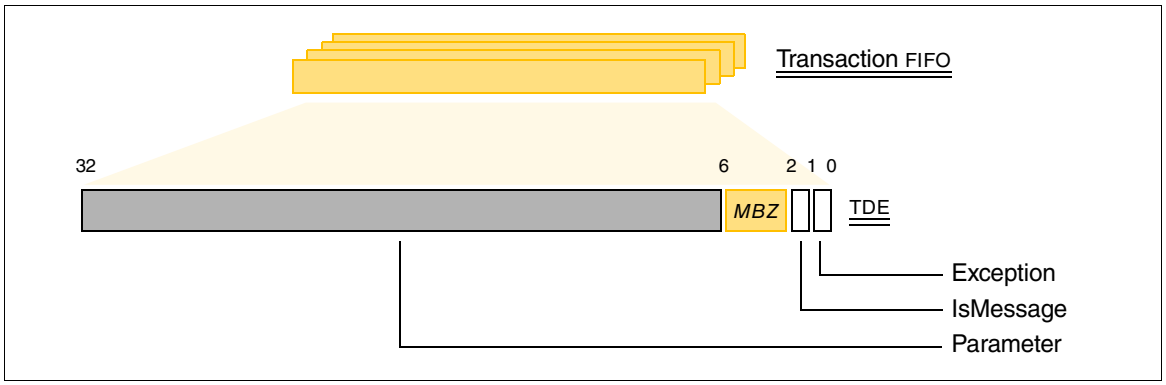

Figure 26 Structure of the TDE as used by the PIB

where:

Exception: This field describes whether or not the transaction completed with error. If the field is true (*set*) the transaction completed with error. In such a case, the corresponding Transfer Descriptor has the structure described in Section 4.12.2. I f the field is false (*clear*) the transaction completed without error. In such a case, the corresponding Transfer Descriptor has the structure described in Section 4.12.1.

IsMessage: This field specifies the interpretation of the ***Parameter*** field (see below). If this field is *clear*, the entry corresponds to a completed import transaction and the ***Parameter*** field is a pointer to a Transfer Descriptor. If this field is *set*, the entry is a message generated by the protocol engine (see Section 4.14). In such a case the meaning of the ***Parameter*** field is protocol dependent.

Parameter: The interpretation of this field depends on the value of the ***IsMessage*** field (see above). If the ***IsMessage*** field is *clear*, this field is a reference to the Transfer Descriptor (see Section 4.2) corresponding to the completed transaction. In actuality, the twenty-six most significant bits of the Transfer Descriptor's address as all Transfer Descriptors must be aligned on a 64-bit boundary and therefore, the low order six address bits are assumed *zero*. The address was extracted from the TDE corresponding to the entry from a FLB's freelist (see Section 4.8.3). If the ***message*** field is *set*, the meaning of this field is protocol dependent.

4.10.3 Faults triggered by a Protocol *Core*

These faults occur in the interface between the protocol *core* and its corresponding PIB. In such a case, the TDE corresponding to the fault is not available to the PIC and therefore the `TDE_Fault_Register` (see Section 6.1.3) is simply *cleared*. See the description of the PIB's Front-End interface (Section 5.3.1) for more information.

4.10.3.1 No such FLB

The protocol core requires a transfer descriptor from a Freelist Block (see Section 4.9) not instantiated in the CE.

4.10.3.2 Data Pipeline Full

The protocol core attempted to send data to the block when the block was full. Formally, this corresponds to the protocol core asserting the block's `Advance_Data_Pipeline` signal while its `Data_Available` signal is asserted.

4.10.4 Faults triggered by a Protocol Driver

None.

4.11 The Interrupt Summary Block

The Interrupt Summary Block (ISB) aggregates the `Event` and `Fault` signals from the four types of transfers blocks (See respectively sections 4.7, 4.8, 4.9 and 4.10 for a description of these blocks). The connections between ISB and instantiated block are created automatically by the PIC (see Section 4.1.4). Unused ports appear as *not* asserted. The logical OR of the `Event` signals drives the processor's *non-critica*l interrupt and in an analogous fashion, the logical OR of the `Fault` signals drives the processor's *critical* interrupt. The current state of all these signals is reflected in four different DCR registers. A block diagram of this interface is illustrated in Figure 27:

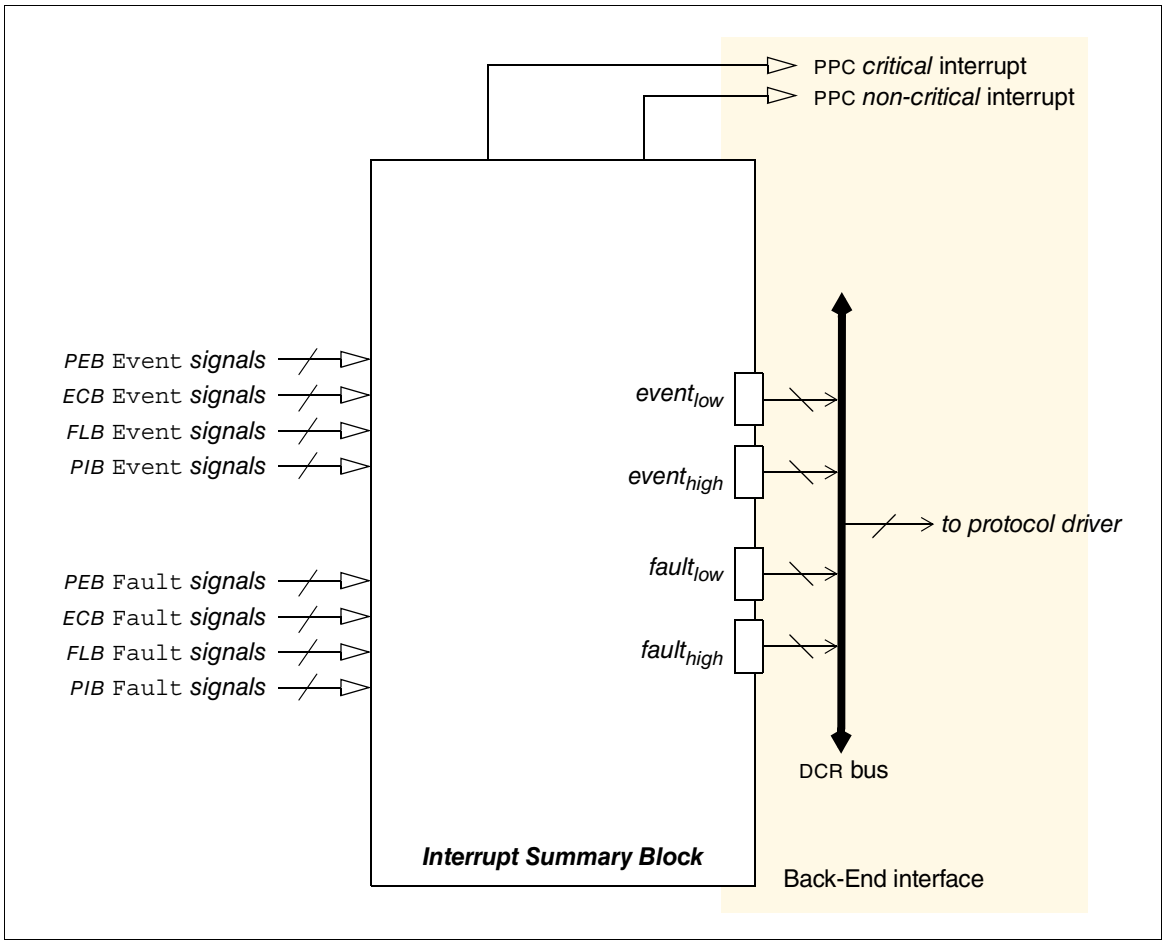

Figure 27 Block diagram of the ISB

The specification for the Back-End Interface is found in Section 6.5. For each CE there will be *one* and *only* one, ISB instantiated.

4.11.1 Parameters

For an ISB:

— DCR address offset for its register block

4.12 The Export transaction

Figure 28 illustrates the typical set of operations involved in any one export transaction. The appropriate protocol driver allocates and initializes both the packet to be transmitted and its

associated Transfer Descriptor and then posts this descriptor to the appropriate PEB. The causes the block's transaction FIFO to go *Not-Empty*, which in turn wakes up the block's Front-End Interface, which removes the descriptor from the FIFO, fetches the descriptor and signals the protocol driver which transmits the packet. When transmission is complete, the protocol driver signals completion which inserts the descriptor on the transaction FIFO of the appropriate ECB. In the meanwhile the protocol driver has been waiting on this FIFO to become *Not-Empty*. When this FIFO goes *Not-Empty*, the protocol driver removes the descriptor and processes the result. Once processing is complete, the protocol driver returns the packet to its free store.

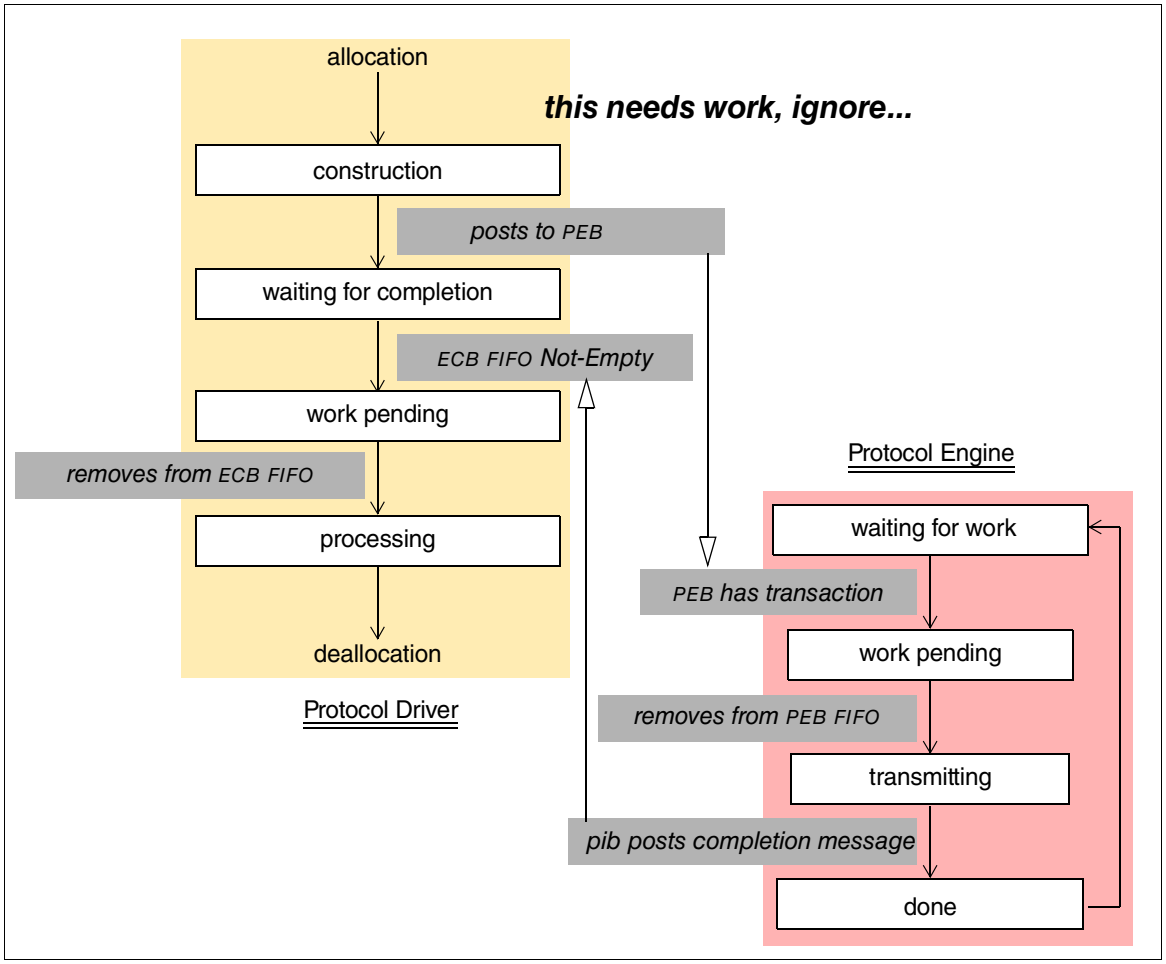

Figure 28 Typical lifetime of a Transfer Descriptor used to transmit a packet

4.12.1 Data structures for a successful export transaction

When an export transaction completes a TDE has been posted to the appropriate ECB and subsequently removed by the protocol driver. The low-order (***Exception***) field of this TDE is *false*, indicating the transaction completed without error. The value of its TDR field is identical to the TDR of the TDE which originated the transaction. Thus the TDR in the completion TDE points to the original Transaction Descriptor. Because the transaction was successful, the TCD

pointed to by the TCD field within the descriptor was not accessed by the PIC and its contents are irrelevant to the transaction. All these relationships are illustrated within Figure 29:

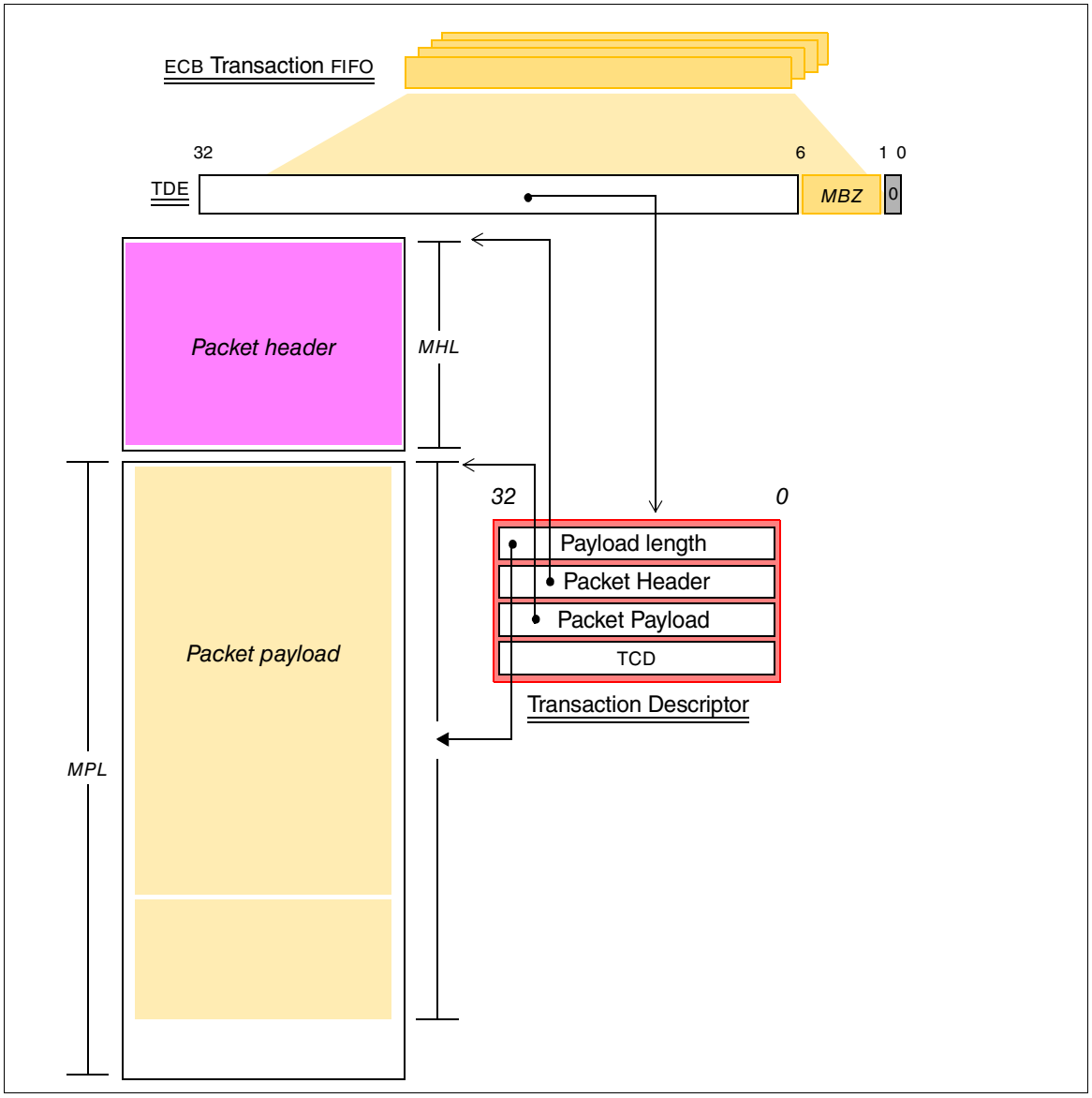

Figure 29 Data structures involved in a successful export transaction

4.12.2 Data structures for a failed export transaction

As was the case of a successful transaction, when an export transaction completes a TDE has been posted to the appropriate ECB and subsequently removed by the protocol driver. However, this time the low-order (***Exception***) field of this TDE is *true*, indicating the transaction failed. The value of its TDR field is identical to the TDR of the TDE which originated

the transaction. Thus the TDR in the completion TDE points to the original Transaction Descriptor. Because the transaction failed, the PIC has written the TCD pointed to within the descriptor. Therefore, the ***Transferred*** field of the TCD specifies how many bytes were successfully transmitted[1]. The EDW has its low-order bit (***wasError***) *set*. The ***Reason*** and ***Parameter*** fields have been written with values which describe the specific error (see Section 4.3). All these relationships are illustrated within Figure 30:

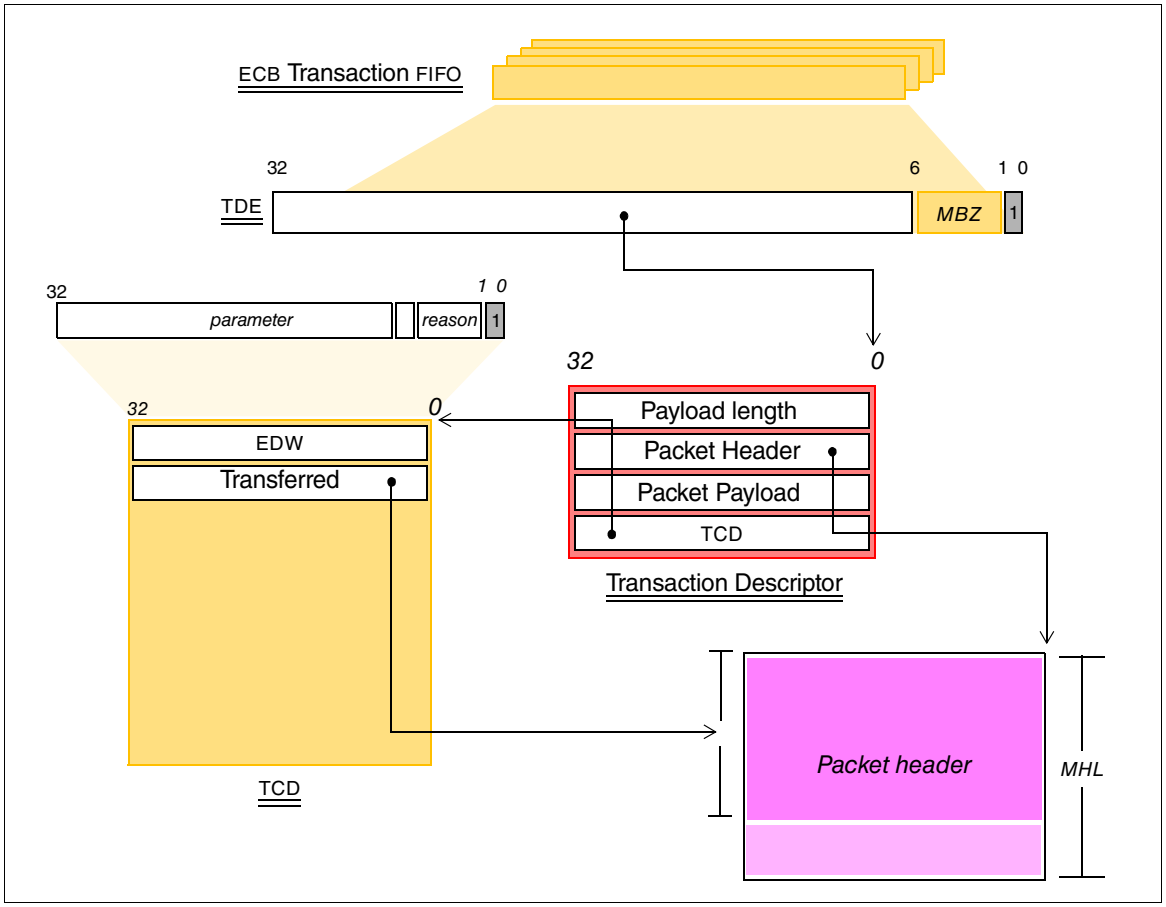

Figure 30 Data structures involved in an unsuccessful export transaction

4.13 The Import transaction

Figure 31 illustrates the typical set of operations used in a import transaction. The example assumes the protocol driver has *a-priori* allocated receive buffers (and their associated Transfer Descriptors) and inserted these descriptors on the appropriate FLB's *freelist*. When a packet arrives off the fabric the protocol engine wakes up, locates the appropriate FLB's freelist and removes a descriptor. This descriptor specifies where in memory the protocol engine should locate the received packet. When reception is complete, the protocol engine inserts the

1. In this example, the transaction failed somewhere within transmitting the header.

descriptor on the appropriate PIB's FIFO. In the meanwhile the protocol driver has been waiting on this FIFO. When this FIFO goes not empty, the protocol driver removes the descriptor and processes the transfer result and its corresponding packet. Once processing is complete, the driver re-inserts the descriptor back on the appropriate *freelist*, where it can be re-used for subsequent transfers.

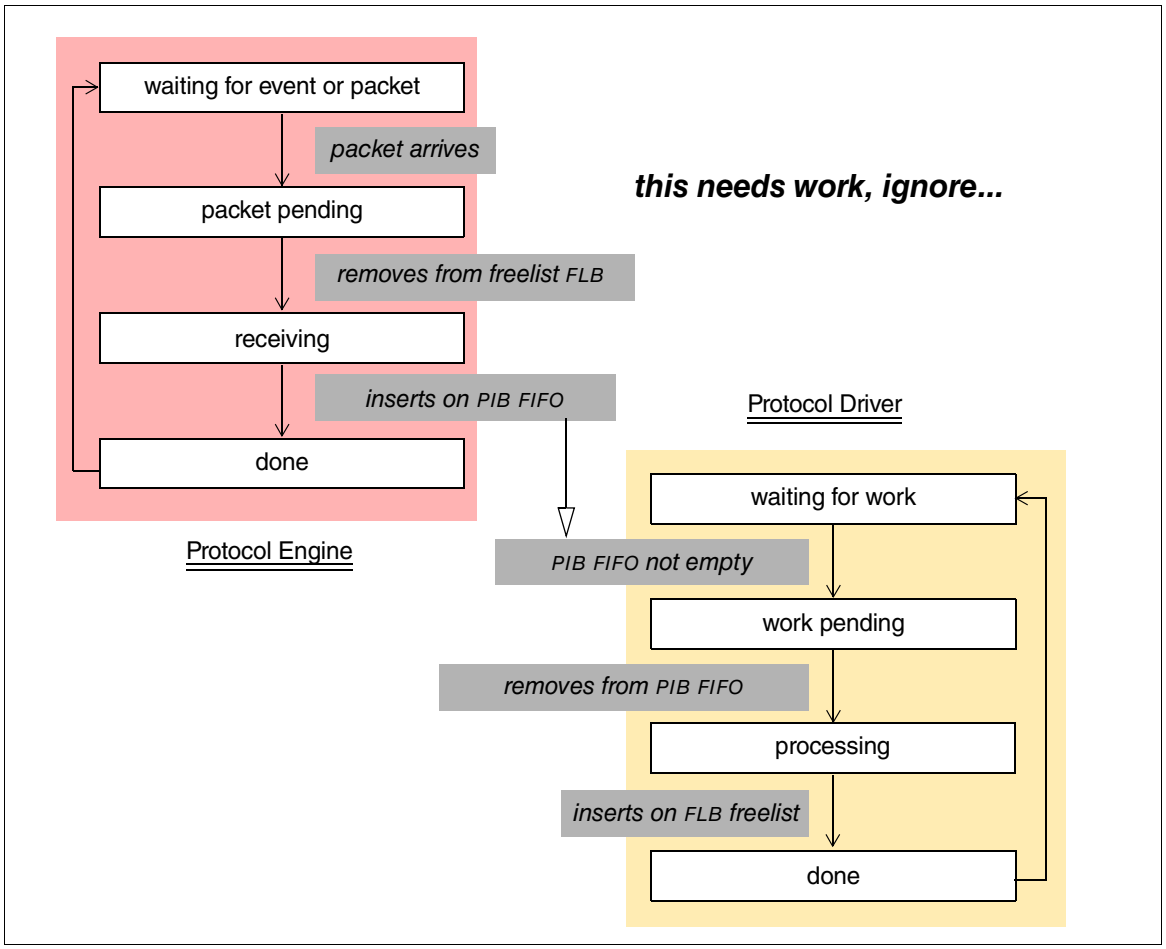

Figure 31 Typical lifetime of a Transfer Descriptor used to receive a packet

4.13.1 Data structures for a successful import transaction

When an import transaction completes a TDE has been posted to the appropriate PIB and subsequently removed by the protocol driver. The low-order (***Exception***) field of this TDE is *clear*, indicating the transaction completed without error. The value of its TDR field is identical to the TDR of the FLB which provided the buffer for the transaction. Thus, the TDR in the completion TDE points back to the original Transaction Descriptor. However, unlike an import transaction, the PIC *always*, independent of completion status, writes the TCD pointed to by the Transaction Descriptor. The EDW has its low-order bit (***wasError***) *clear*, indicating the

transaction was successful. The remaining fields of the EDW are “don’t care”, however, they have been initialized to *zero* (0). The ***Transferred*** field contains the length (in bytes) of the received packet. All these relationships are illustrated in Figure 32:

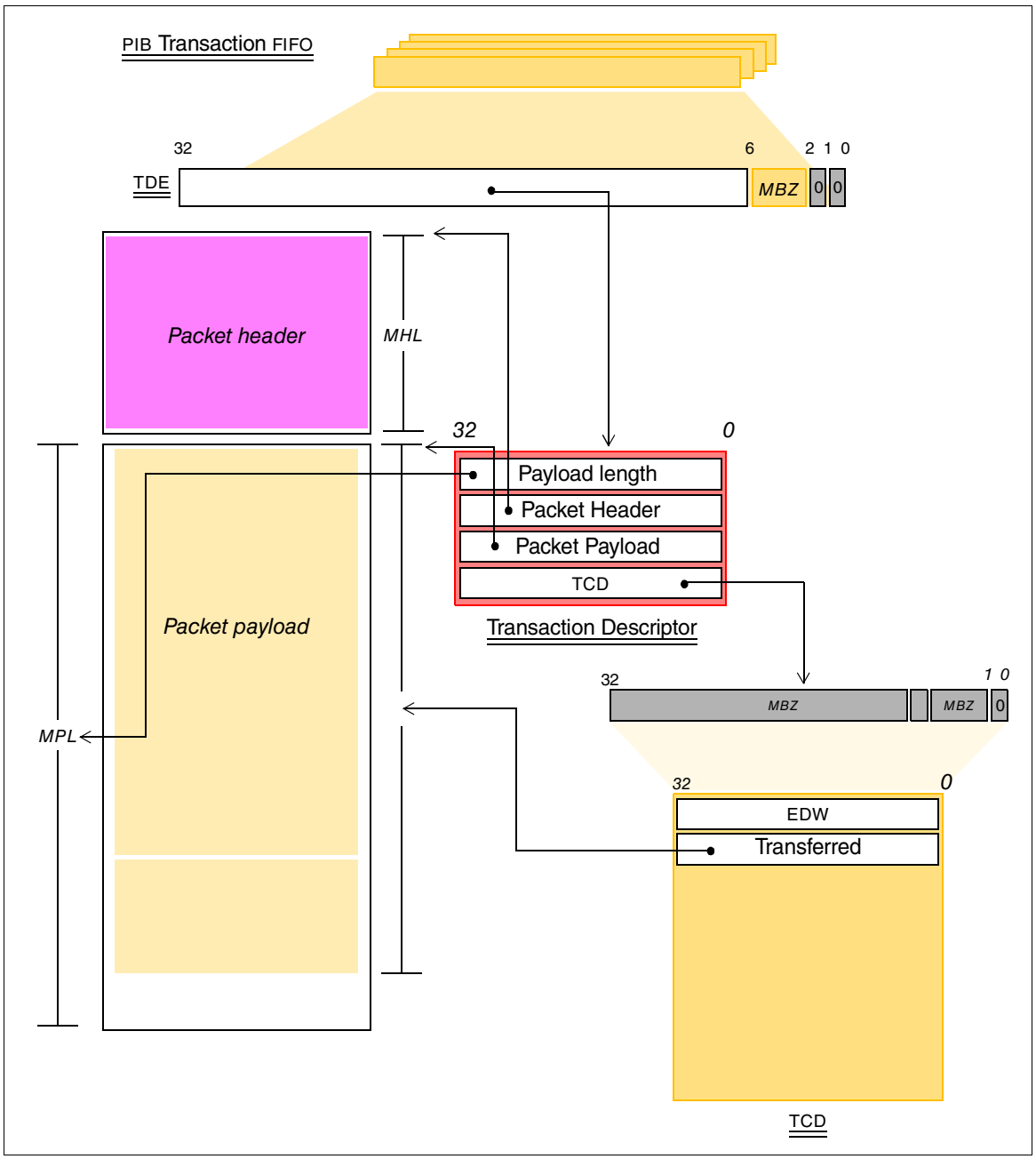

Figure 32 Data structures involved in a successful import transaction

4.13.2 Data structures for a failed import transaction

As was the case of a successful import transaction, when a failed import transaction completes a TDE has been posted to the appropriate PIB and subsequently removed by the protocol driver. However, this time the low-order (***Exception***) field of this TDE is *set*, indicating the transaction failed. The value of its TDR field is identical to the TDR of the FLB which provided the buffer for the transaction. Thus the TDR in the completion TDE points back to the original Transaction Descriptor. Because the transaction failed, the EDW has its low-order bit (***wasError***) *set*. The ***Reason*** and ***Parameter*** fields have been written with values which describe the specific error (see Section 4.3). Finally, the ***Transferred*** field of the TCD specifies how many bytes were successfully received. All these relationships are illustrated in Figure 33:

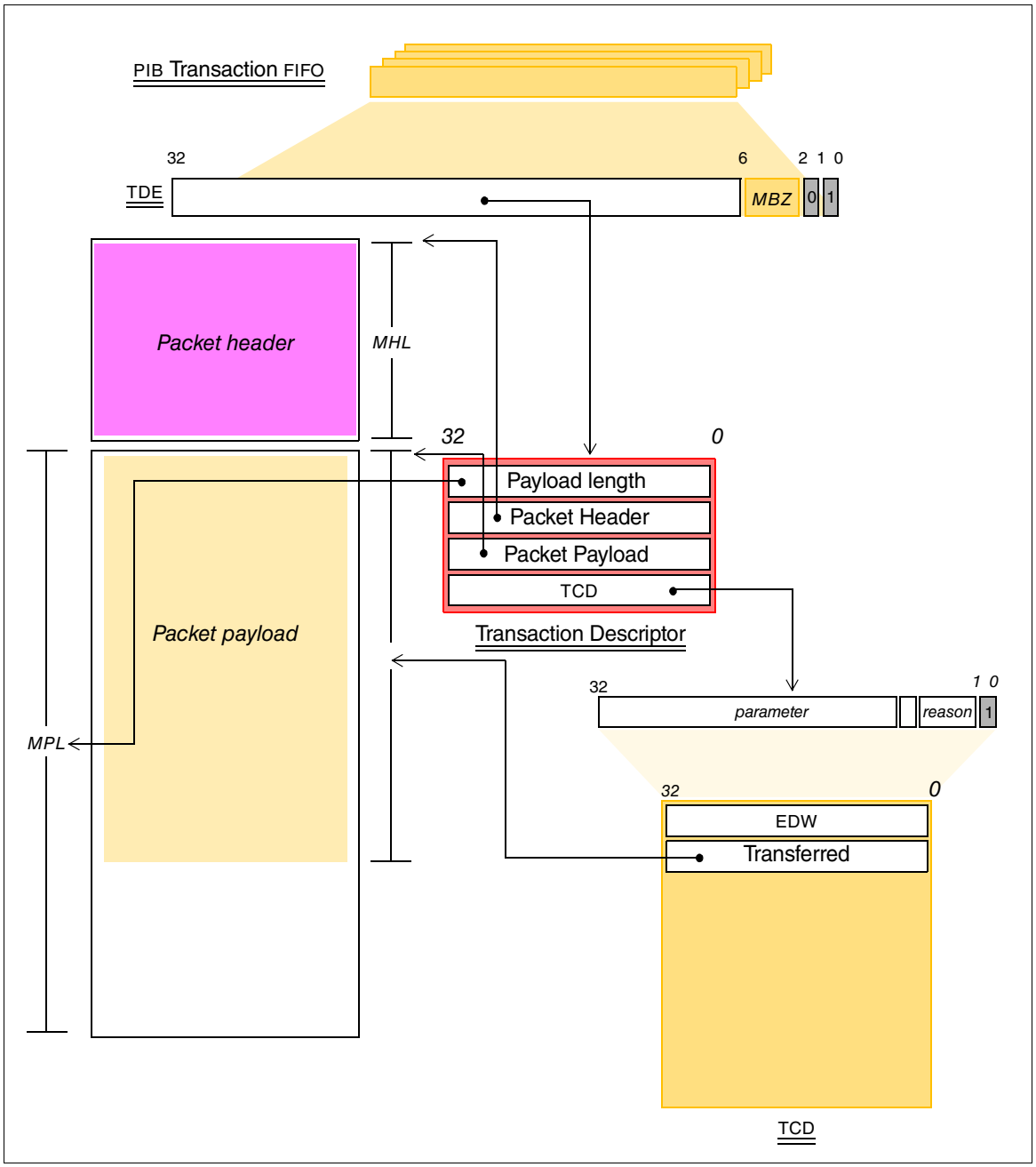

Figure 33 Data structures involved in an unsuccessful import transaction

4.14 The protocol engine message transaction

The protocol engine may find the necessity to communicate to its corresponding driver messages which do not necessarily announce packet arrival, but must be in-band with respect to the flow of such traffic. For example, in the implementation of an ACK/NACK protocol the

protocol engine might maintain timers to time out acknowledgments. When these timers fire, they would inform the driver of the need to either "age" or perhaps re-transmit packets pending acknowledgment. To allow for this possibility, the PIB provides the protocol core with the option of posting a TDE which does not contain a reference to a Transfer Descriptor (see Section 5.3). Such an exchange looks very much like an import operation (see Section 4.13) and is illustrated in Figure 34.

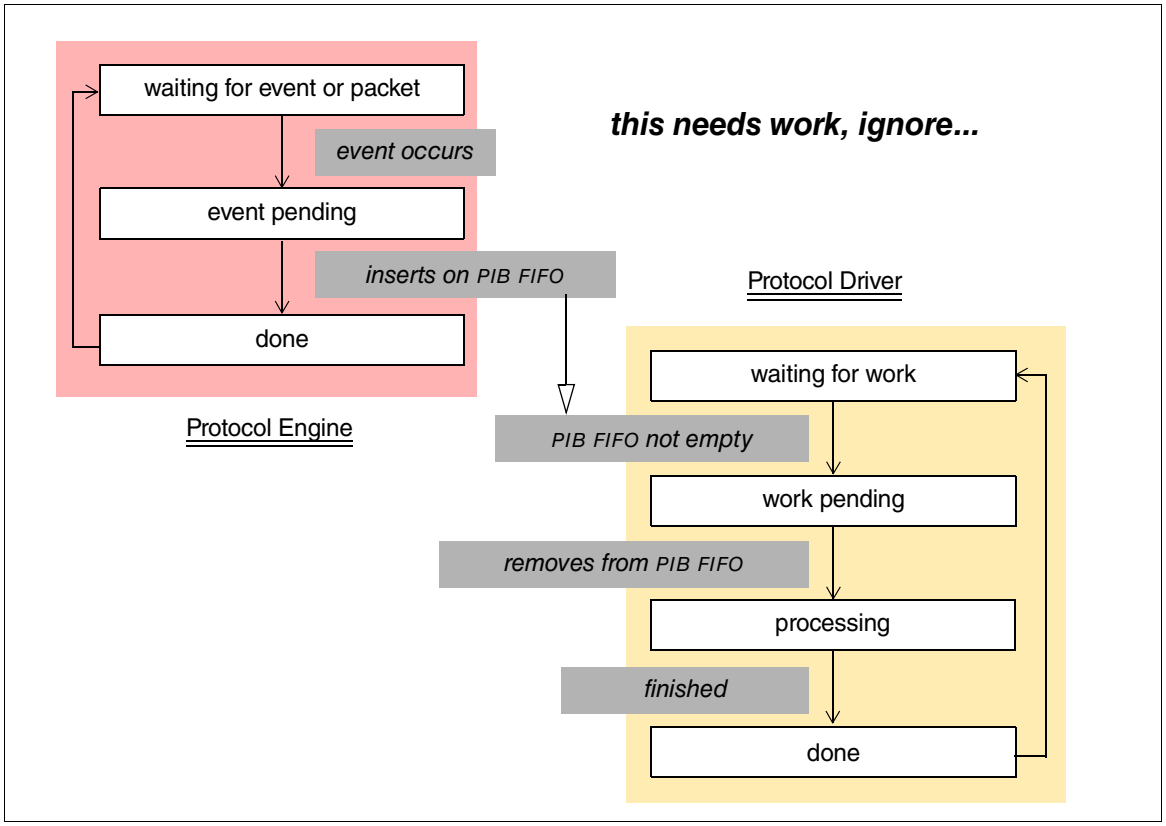

Figure 34 Information exchange for a protocol engine message transaction

4.14.1 Data structures for a message transaction

As was the case for any import transaction, when a message transaction completes a TDE has been posted to the appropriate PIB and subsequently removed by the protocol driver. However, this time the low-order (***Exception***) field of the TDE is *false*, while the ***Message*** field is *true*. This specifies the TDE contains a message from the protocol core for the protocol driver. The message is located in the high-order (***Parameter***) field and is 24 bits long. The interpretation of this message is protocol dependent. These relationships are illustrated in Figure 35:

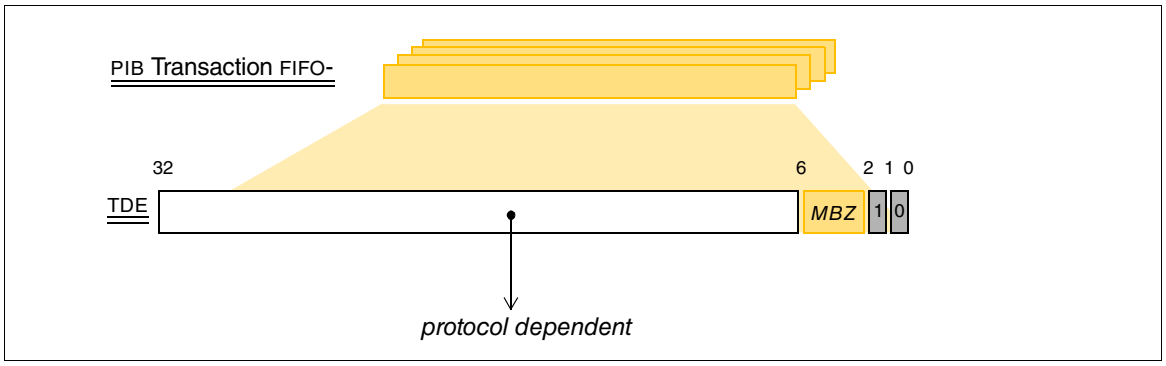

Figure 35 Data structures involved in a protocol engine message transaction

Chapter 5
The PIC Front-End Interface

5.1 Introduction

5.1.1 Terminology

The implementation body of a data transfer requires clocking information either to or from a data *pipeline*. The entrance or exit of this pipeline is called its *data port*. The width of a data port is 64-bits, which is either 8 *bytes* or *1/4* of a processor cache-line. Information is given to or taken from the pipeline in units of *lines*, where a line represents from one to seven bytes of information. One line of data may contain header information, payload information, or both. A line which presents only header information will be identified as H_n: where *n* starts at *one* (1) and represents the n^{th} clocked line.

A line which presents only payload information is identified as D_n. A line which presents both header and payload information is identified as $H_M D_M$: where *M* represents the last clocked line with any header information.

It is the protocol's responsibility to advance a pipeline. In order to do so both the PEB and PIB define a signal which is to be asserted by the protocol core:

— For the PEB this signal is `Export_Advance_Data_Pipeline` (see Section 5.2.2.4)

— For the PIB this signal is `Import_Advance_Data_Pipeline` (see Section 5.3.1.3)

In either case, a single assertion of this signal will be identified as a_n: where *n* starts at *zero* (0) for an *export* transaction and *one* (1) for an *import* transaction.

5.1.2 Post-processing and completion status

Independent of packet transmission or reception, once a transaction is complete, a protocol core will undergo a post-processing phase. This phase must include at a minimum signalling

transaction completion to the appropriate block (PEB or PIB). While the specific guidelines for post-processing depend on whether the core is importing or exporting, the generic guidelines are as follows:

— Completion must be signalled *once* and *only* once per transaction.

— At a minimum, completion is signalled by the protocol core passing a 32-bit structure called the EDW (see Section 4.3).

— The value of the EDW must mirror transaction status. In particular, if the transaction succeeds, the low-order (***wasError***) field must be *clear*, if the transaction fails, the field must be *set*.

— On failure, the value of the EDW must describe the nature of that failure.

The specific guidelines for exporting are described in Section 5.2.5. The specific guides for importing are described in Section 5.3.4

Note: Failure of a protocol core to follow both these guidelines and the specific rules governing import and export could result in the fatal operation of a block and subsequent hanging of any I/O requests processed through the PIC.

5.1.3 What to do if a data or status pipeline is full?

Under exceptional circumstances the block may not be able to consume data given by the protocol core. In the case of the PEB this circumstance is signified by the assertion of the `Status_Pipeline_Full` signal while the protocol core asserts `Pipeline_Advance` and for the PIB this circumstance is signified by the `Data_Pipeline_Full` signal while the protocol core assert `Pipeline_Advance`.

tbd...

5.2 Exporting

The PEB's Front-End-Interface is used by a protocol core to *export* packets from memory. Export requests originated with the protocol's corresponding driver. This driver communicates these requests by posting to the PEB's Back-End Interface. After the protocol core completes a transfer it signals completion status back to the block. Conditionally, depending on transaction and completion status, the PEB forwards this completion information (through the export cross-point) to the appropriate ECB. The ECB (see Section 5.2) would then post this information to the protocol driver which completes the transaction. The PEB's Front-End signals are enumerated in Table 6. A detailed discussion of the function of the PEB is found in Section 4.7. Schematically, from the perspective of the protocol core, any one transfer involves the following steps:

— The block gains the attention of the protocol core by asserting its `Data_Available` signal. This signal remains asserted while the block has at least one packet waiting to transmit.

— The pipeline is primed by asserting `Data_Pipeline_Advance` once before clocking data (see Section 5.2.1 below).

— While `Data_Available` is asserted, the protocol core reads the packet data from block's `Data` port. The protocol core requests data from that port by asserting `Data_Pipeline_Advance`. Each assertion of this signal delivers one line and each line may contain from to one eight bytes of data. When the last line is made available by the block, it asserts `Data_Last_Line`. The last line of data may contain from one to eight bytes. Odd packet sizes are communicated using the `Data_Last_Valid_Byte` port, which specifies how many bytes on the last line are valid. The structure of the data delivered through this port is discussed in more detail in Section 5.2.3.

— After the protocol core transfers the packet it must signal transaction completion to the block. This is accomplished by writing to the block's `Status` port. The value(s) written to that port correspond to the values which are potentially written to the TCD (see Section 4.3) for the transaction. This process is described in detail within Section 5.2.5.

5.2.1 Advancing the export pipeline

Note that the signals and ports associated with an export pipeline (see, for example, `Data`, `Data_Last_Value`, and `Data_Last_Valid_Byte`) are in reality the contents of registers which are loaded from the outputs of the corresponding bits of the FIFO memory array at the conclusion of an `Clock` tick in which `Advance_Data_Pipeline` is asserted. Thus, the protocol core control logic must pull each successive data word out of the FIFO memory array into these output registers prior to absorbing the corresponding data into the protocol core data pipelines. This arrangement reduces the clock-to-output propagation delay for valid signals to that of the output registers themselves, instead of having a clock advance the FIFO memory array address and then having to wait a memory access time for the data to become valid. This arrangement requires the advance signal to be asserted one clock *before* its corresponding line of data is clocked. Therefore, the n^{th} line of data is present at the $n + 1$ advance, or at A_{n+1}.

5.2.2 PEB Signal Descriptions

Table 6 Signal definitions for the PEB

Signal name	Direction[1]	Description found in:
Export_Clock	Input	Section 5.2.2.1
Export_Data_Available	Output	Section 5.2.2.2
Export_Data_Start	Output	Section 5.2.2.3
Export_Advance_Data_Pipeline	Input	Section 5.2.2.4
Export_Data_Last_Line	Output	Section 5.2.2.5
Export_Data_Last_Valid_Byte[0:2]	Output	Section 5.2.2.6
Export_Data[0:63]	Output	Section 5.2.2.7
Export_Advance_Status_Pipeline	Input	Section 5.2.2.8
Export_Status[0:31]	Input	Section 5.2.2.9
Export_Status_Full	Output	Section 5.2.2.10
Export_Core_Reset	Output	Section 5.2.2.11

1. From the perspective of the block.

5.2.2.1 Export-Clock

This signal is the clock provided by the protocol core to the block. All other inputs to, and outputs from the protocol core must be synchronous with respect to this signal. The signal is assumed to be the output of a *Xilinx* BUFG global clock buffer. The identical clock buffer output may be used for multiple instantiations of PIC export logic when the corresponding instantiations of a class of protocol core permit this – as in the case , for example, of multiple independent PGP lanes. The clock frequency must be at least 50% of the system (bus) clock frequency – i.e. the clock period must be less than two periods of the system (bus) clock.

5.2.2.2 Export-Data Available

This signal indicates that a packet is available for the protocol core to transmit. Data for this packet are read from the Data port described below and are clocked out using the Advance_Data_Pipeline signal, also described below. This signal is first asserted when the first line of the packet becomes available. Once asserted, it *should* remain asserted until the last line of the packet has been read from the block (and possibly longer, if another packet is already present in the block). However, there is no guarantee that this condition can always be met. Such a case is called a transmit data *under-run*. The responsibilities of the protocol core in both detecting and handling such a case are enumerated in Section 5.2.5.

5.2.2.3 Export-Data Start

this section (and all sections which should describe its use) needs work...

This signal indicates that a packet is available for the protocol core to transmit. Data for this packet are read from the `Data` port described below and are clocked out using the `Advance_Data_Pipeline` signal, also described below. This signal is first asserted when the first line of the packet becomes available. Once asserted, it *should* remain asserted until the last line of the packet has been read from the block (and possibly longer, if another packet is already present in the block).

5.2.2.4 Export-Advance Data Pipeline

This signal is the request from the protocol for the next line of data. The line is obtained by reading the `Data` port described below. This signal must be asserted only while the `Data_Available` signal remains asserted. If this signal is asserted while `Data_Available` is *not* asserted the block will generate a ***Data Pipeline Empty*** fault (see Section 4.7.3.1)

5.2.2.5 Export-Data Last Line

This signal indicates that the current line read from the block is the last line of the packet. This signal will be asserted *once* and *only* once per transaction.

5.2.2.6 Export-Data Last Valid Byte

This field indicates the byte offset of the last valid byte in the last transferred line. The contents of this field are only valid when the `Data_Last_Line` signal is asserted. A value of `000` (binary) in this field indicates that the only valid byte is located within byte offset 0; while a value of `111` indicates that the last valid byte is located within byte offset 7 and thus the entire last line is valid. *Note*: Byte offsets are numbered in little-endian order.

5.2.2.7 Export-Data

The data port for the block. Packet data are read from the block through this port coincident with the `Adavance_Data_Pipeline` signal while `Data_Available` is active. Section 5.2.3 describes the structure of data read from this port. All eight bytes returned by a single advance are valid except when `Data_Last_Line` is asserted. In which case the location, relative to the port of the last valid byte, is signified by the simultaneously asserted value of the `Data_Last_Valid_Byte` port described above.

5.2.2.8 Export-Advance Status Pipeline

This signal is the request from the protocol core to load either the completion status or byte count to the block's status FIFO. The value is presented on the `Status` port described below. This signal must be asserted only while the `Status_Full` signal is not asserted. If this signal is asserted while `Status_Full` is asserted the block will generate a ***Status Pipeline Full*** fault (see Section 4.7.3.2)

5.2.2.9 Export-Status

The transfer completion status port for the block. Transferred completion status and/or completion byte counts are clocked by the protocol core to the block through this port coincident with the `Adavance_Status_Pipeline` signal while the `Status_Full signal` is *not* asserted. Data must be written to this port at least *once* and no more then *twice* for each transaction. See Section 5.2.5 for a description of that data.

5.2.2.10 Export-Status Full

This signal is the back-pressure line from the block to the protocol core. In the normal course of operation this signal should never be asserted. It signifies that the block does not have enough capacity to absorb either EDW or transfer count. The protocol core must wait until this signal is deasserted before loading either EDW or transfer count into the `Status` port. If `Advance_Status_Pipeline` is asserted while this signal is also asserted the block will generate a ***Status Pipeline Full*** fault (see Section 4.7.3.2) See Section 5.1.3 for guidelines concerning the use of this signal.

5.2.2.11 Export-Core Reset

This signal is a request to *reset* the transmit side of the protocol core.This is a pulse which started life as a pulse in the system clock domain with a minimum width of two clock ticks. It then passes through a two-stage synchronizer moving it into the block's `Clock` domain. As long as the `Clock` signal has a frequency which is at least 50% of the system clock frequency, this pulse will have a minimum duration of one tick of the `Clock` signal. The pulse can have one of three origins:

— The protocol driver through the block's CSR (see Section 6.1.1).

— The Global reset signal.

5.2.3 Transfer data structure

The structure of the data read through the block's `Data` port consists of two components:

Header: The packet header. The header's length is fixed and was determined by an instantiation parameter for the block. However, note that the size of the header is not necessarily an even number of lines. The header is emitted on the *first* advance.

Payload: The packet payload. Unlike the header, the payload's length is not fixed and will vary from one transfer to the next. The payload follows immediately after the last byte of the header and continues until the `Last_Word` signal is asserted. If the header was not an even number of lines, the advance which fetches the end of the header will also include some fraction of the beginning payload bytes. As was the case with the header, the payload may not be an even number of lines. In such a case, the `Last_Valid_Bytes` port is used to determine the fractional remainder.

As an example, consider a packet which is composed of a header 18 bytes long and a payload 52 bytes long. Assuming the first advance (A_0) has previously been asserted, the data for this packet from the `Data` port as illustrated in Figure 36:

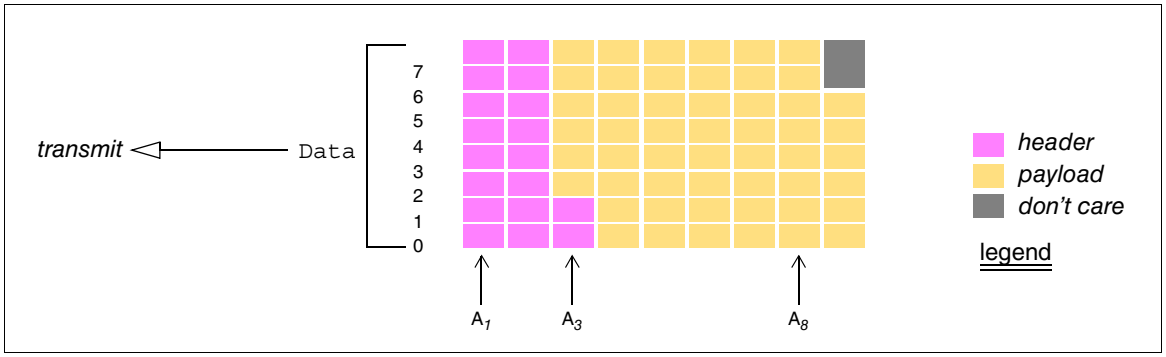

Figure 36 Transfer structure for the PEB.

Rounding the header up to an integral number of lines, three advances are required to transfer the header. The first two advances read the first *sixteen* bytes, while the last advance reads the last *two* bytes. The first bye of the packet payload immediately follows the header and therefore the third advance also returns the first *six* bytes of payload. Six additional advances are required to clock the remainder of the packet. As the advance signal is one clock ahead of its corresponding line, the last advance reads the last line and this line is partially filled with *six* bytes.

5.2.4 Timing examples

A hypothetical timing diagram for the packet illustrated in Figure 36 is shown in Figure 37. Recall that this packet is contained in nine lines.

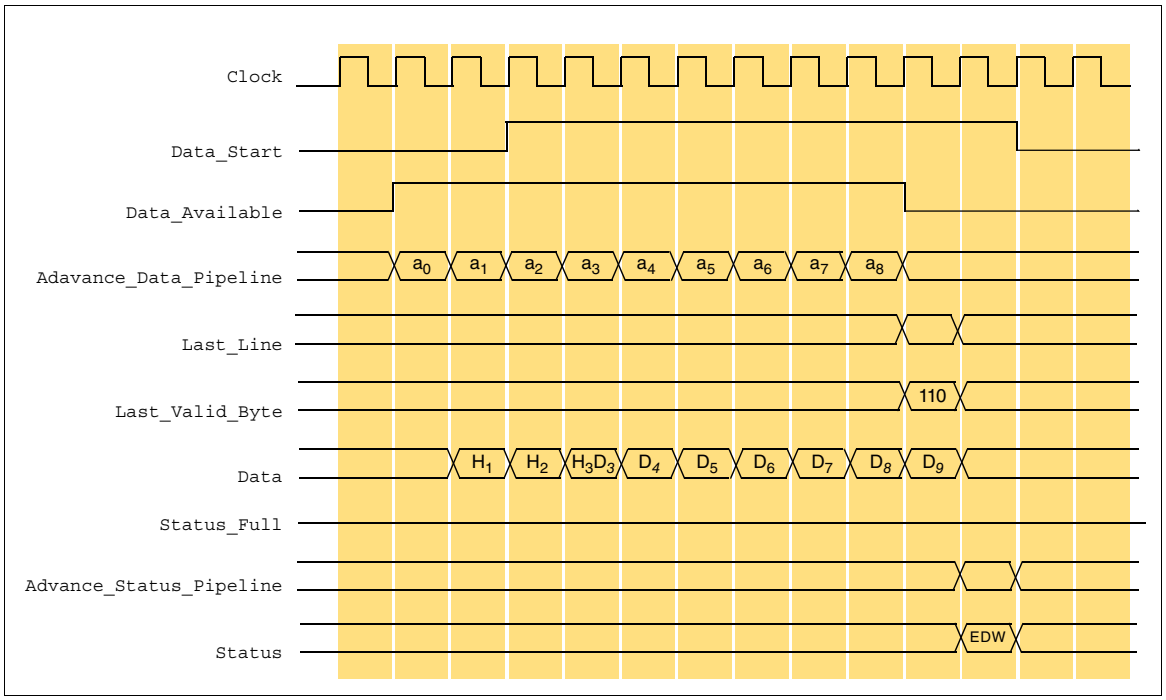

Figure 37 Timing diagram (one transfer) for the PEB

Immediately after detecting the block has a transaction to process (`Data_Available` is asserted), the protocol core asserts `Advance_Data_Pipeline`. This initial advance primes the data pipeline. Data can now begin to be clocked through the data port. Eight additional advances are required to obtain the entire packet. At the *ninth* line the block asserts `Last_Line`. Coincident with that assertion, sampling the `Last_Valid_Byte` port returns a value of *six* indicating that the last line contains only six valid bytes. Once `Last_Line` is deasserted the entire packet has been transferred. Immediately following this deassertion the protocol core completes the transaction by writing the appropriate EDW (see Section 5.2.5) to the `Status` port.

As another example, assume the driver has queued three packet transfers requests and each packet consists of only a sixteen byte header with no payload. The timing diagram for this scenario is illustrated in Figure 38:

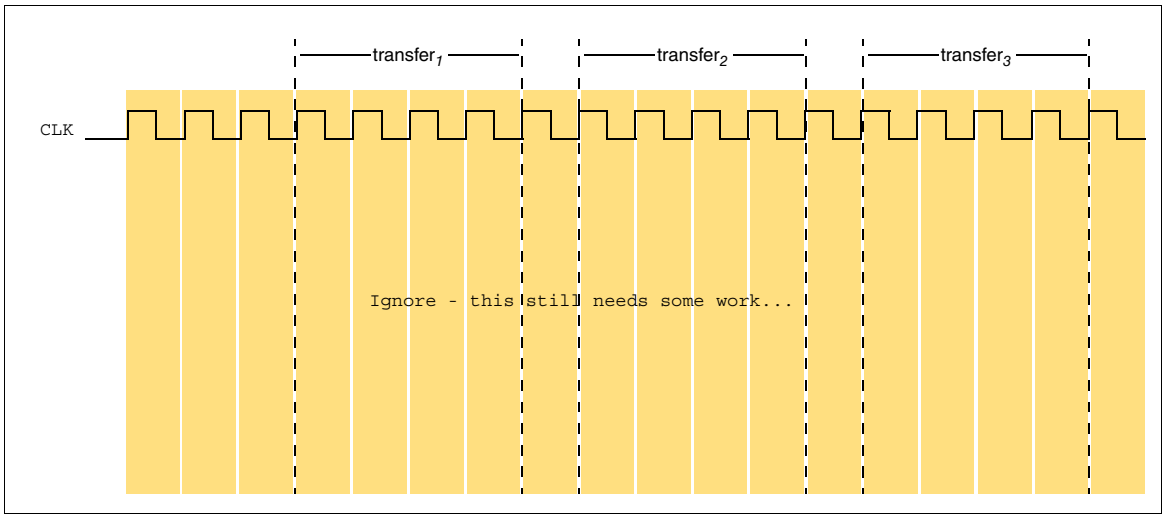

Figure 38 Timing diagram for PEB three packet transfer

5.2.5 Export Post-processing

The protocol core must follow both the generic guidelines for post-processing as defined in Section 5.1.2 and the specific guidelines for exporting as outlined below:

The transfer succeeds: In such a case the protocol core's responsibility is *two*-fold:

- It must assert completion to the block. Completion is asserted by writing *once* to the block's `Status` port.
- The asserted value must correspond to the EDW for the transaction (see Section 4.3) The constraints on the values of its fields in this case are as follows:
  - The ***WasError*** field of the must be *false* (*clear*).
  - The ***Reason*** field is "Don't Care".
  - the ***WasBlock*** field is "Don't Care".
  - The ***Parameter*** field is "Don't Care".

The transfer fails due to a data under-run error: This is an example of the type of error which had its origin in the system, but which can only be detected by the protocol core. The definition of this type of error is as follows:

`Data_Available` is deasserted *before* `Data_Last` during packet transmission.

In such a case the protocol core's responsibility is *four*-fold:

- It must detect this type of error.
- It must assert completion to the block. Completion is asserted by writing *twice* to the block's `Status` port.

— It must advance *all* the packet's data, independent of where within the transmission under-run occurred. Presumably the data following the under-run is simply discarded, but its actual disposition will be protocol specific. For example, the protocol could "poison" the transmitted packet, but continue its transmission to the "wire".

— The value asserted first must correspond to the EDW for the transaction (see Section 4.3). The constraints on the values of its fields in this particular case are as follows:

  — The ***WasError*** field of the must be *true* (*set).*

  — The ***WasBlock*** field must be *true (set).*

  — The ***Reason*** field must have the value DATA_UNDER_RUN (63 decimal).

  — The ***Parameter*** field must be *zero*.

— The second value written must be the number of bytes successfully transmitted. Writing this value must follow (but not necessarily immediately) writing the EDW. The byte count must be written *once* and *only* once per failed transaction.

The transfer fails due to a protocol error: In such a case the protocol core's responsibility is *three*-fold:

— It must assert completion to the block. Completion is asserted by writing *twice* per to the block's Status port.

— It must advance *all* the packet's data, independent of independent of where within the transmission under-run occurred. Presumably the data following the error is simply discarded, but its actual disposition will be protocol specific. For example, the protocol could "poison" a transmitted packet, but continue its transmission to the "wire".

— The value asserted first must correspond to the EDW for the transaction (see Section 4.3). The constraints on the values of its fields in this case are as follows:

  — The ***WasError*** field of the must be *true* (*set).*

  — The ***WasBlock*** field must be *false (clear).*

  — The ***Reason*** field must have a value appropriate to the transfer error. All unique errors should be assigned unique numbers.

  — The ***Parameter*** field should have a value appropriate to the error.

— The second value written must be the number of bytes successfully transferred. Writing this value must follow (but not necessarily immediately) writing the EDW. The byte count must be written *once* and *only* once per failed transaction.

In short the protocol core must adhere to the following rules:

— Independent of success or failure, transfer completion must be signalled by writing at least *once* and at most *twice* to the block's Status port. Completion may be signalled at any point within or without the actual transfer, but it must be signalled in the same order that transactions were processed and retired.

— Completion is signalled at a minimum by writing an EDW appropriate to the transaction. The appropriate value for the EDW is determined by whether or not the transaction failed and if it did fail, the nature of that failure.

— If the transaction does fail, the protocol core must write a second value (after the EDW) to the block's `Status` port. This value must contain the number of bytes successfully transmitted or received before the failure was detected.

5.3 Importing

The PIB's Front-End-Interface is used by a protocol core to *import* packets to memory. Once a packet has been transferred the PIB signals the corresponding protocol driver that it has one or more packets pending to process. The PIB's Front-End signals are enumerated in Table 7. A detailed discussion of the function of the PIB is found in Section 4.8. Schematically, from the perspective of the protocol core, any one packet transfer involves the following steps:

— When the protocol core has a packet ready to import it gains the attention of the block by asserting its `Data_Pipeline_Advance` signal and its first line of packet data.

— Coincident with this assertion, the protocol core specifies the freelist from which the memory to contain the packet will be allocated.

— The protocol core delivers the packet's data to the block through its `Data` port. The protocol core writes data to that port by asserting `Data_Pipeline_Advance`. Each assertion of this signal delivers one line and each line contains from to one eight bytes of data. When the last line is made available to the block, the protocol core asserts `Data_Last_Line`. The last line of data may contain from one to eight bytes. Odd packet sizes are communicated using the `Data_Last_Valid_Byte` port, which specifies how many bytes on the last line are valid. The structure of the data delivered to this port is discussed in more detail in Section 5.3.2.

— After the protocol core transfers the packet it must signal transaction completion to the block. This is accomplished by again writing to the block's `Data` port. The value written to that port correspond to the values which are written to the TCD (see Section 4.3) for the transaction. This process is described in detail within Section 5.3.4.

5.3.1 PIB Signal Descriptions

Table 7 Signal definitions for the PIB

Signal name	Direction[1]	Description found in:
Import_Clock	Input	Section 5.3.1.1
Import_Free_List[0:3]	Input	Section 5.3.1.2
Import_Advance_Data_Pipeline	Input	Section 5.3.1.3
Import_Data_Last_Line	Input	Section 5.3.1.4
Import_Data_Last_Valid_Byte[0:2]	Input	Section 5.3.1.5
Import_Data[0:63]	Input	Section 5.3.1.6
Import_Data_Pipeline_Full	Output	Section 5.3.1.7
Import_Core_Reset	Output	Section 5.3.1.8

1. From the perspective of the block.

5.3.1.1 Import-Clock

This signal is the clock provided by the protocol core to the block. All other inputs to, and outputs from the protocol core must be synchronous with respect to this signal. The signal is assumed to be the output of a *Xilinx* BUFG global clock buffer. The identical clock buffer output may be used for multiple instantiations of PIC export logic when the corresponding instantiations of a class of protocol core permit this – as in the case , for example, of multiple independent PGP lanes. The clock frequency must be at least 50% of the system (bus) clock frequency – i.e. the clock period must be less than two periods of the system (bus) clock.

5.3.1.2 Import-Freelist

This field specifies the identity of the freelist block (FLB) from which the TDE and related descriptor will be removed in order to process an import data packet. This field is only valid when loading the first data word in an import packet into the block. Specifying an illegal value of this field when loading the first word of an import data packet will result in a ***No such FLB*** fault (see Section 4.10.3.1) fault generated by the block when that word is attempted to be DMA'd to memory. In such a case, the remainder of the packet will be discarded by the block.

5.3.1.3 Import-Advance Data Pipeline

This signal is the request from the protocol core to write the next line to the block. The line is written to the Data port described below. This signal must be asserted only while the FULL

signal is *not* asserted. If this signal is asserted while FULL is also asserted the block will generate a ***Data Pipeline Full*** fault (see Section 4.10.3.2)

5.3.1.4 Import-Data Last Line

This signal indicates that the current line written to the block is the last line of the packet. This signal must be asserted once and only once per transaction.

5.3.1.5 Import-Data Last Valid Byte

This field indicates the byte offset of the last valid byte in the last transferred line. The contents of this field are only valid when the Data_Last_Line signal is asserted. A value of 000 (binary) in this field indicates that the only valid byte is located within byte offset 0; while a value of 111 indicates that the last valid byte is located within byte offset 7 and thus the entire last line is valid. *Note*: Byte offsets are numbered in little-endian order.

5.3.1.6 Import-Data

The data port for the block. Both packet data and completion information are written to the block through this port coincident with the Adavance_Data_Pipeline signal while Full is inactive. Section 5.2.3 describes the structure of data written to this port. All eight bytes written by a single advance are valid except when Data_Last_Line is asserted. In which case the location, relative to the port of the last valid byte, is signified by the simultaneously asserted value of the Data_Last_Valid_Byte port described above.

5.3.1.7 Import-Data Pipeline Full

This signal is the back-pressure line from the block to the protocol core. In the normal course of operation this signal should never be asserted. It signifies that the block does not have enough capacity to absorb either packet data or EDW. The protocol core must wait until this signal is deasserted before loading either packet data or EDW into the Data port. If Advance_Data_Pipeline is asserted while this signal is also asserted the block will generate a ***Data Pipeline Full*** fault (see Section 4.10.3.2) See Section 5.1.3 for guidelines concerning the use of this signal.

5.3.1.8 Import-Core Reset

This signal is a request to *reset* the receive side of the protocol core.This is a pulse which started life as a pulse in the system clock domain with a minimum width of two clock ticks. It then passes through a two-stage synchronizer moving it into the block's Clock domain. As long as the Clock signal has a frequency which is at least 50% of the system clock frequency, this pulse will have a minimum duration of one tick of the Clock signal. The pulse can have one of three origins:

— Front-End interface

— Global reset

5.3.2 Transfer data structure

The structure of the data written to the block's `Data` port consists of three components:

Header: The packet header. The header's length is fixed and was determined by an instantiation parameter for the block. However, note that the size of the header is not necessarily an even number of lines. The header is emitted on the *first* advance.

Payload: The packet payload. Unlike the header, the payload's length is not fixed and will vary from one transfer to the next. The payload follows immediately after the last byte of the header and continues until the `Last_Word` signal is asserted. If the header was not an even number of lines, the advance which fetches the end of the header will also include some fraction of the beginning payload bytes. As was the case with the header, the payload may not be an even number of lines. In such a case, the `Last_Valid_Bytes` port is used to determine the fractional remainder.

EDW: The transaction completion status. See Section 4.3 for a discussion of the EDW and its format. See Section 5.3.4 for a discussion on how completion is signalled back to the block.

As an example, consider a packet which is composed of a header 18 bytes long and a payload 52 bytes long. The data for this packet flows into the `Data` port as illustrated in Figure 39:

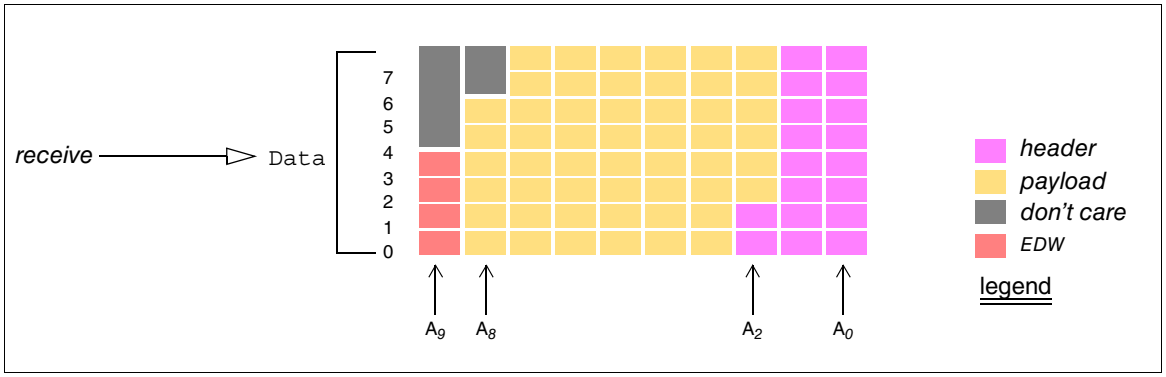

Figure 39 Transfer structure for the PEB.

Rounding the header up to an integral number of lines, three advances are required to write the header. The first two advances write the first *sixteen* bytes, while the last advance writes the last *two* bytes. The first bye of the packet payload immediately follows the header and therefore the third advance also writes the first *six* bytes of payload. Six additional advances are required to clock the remainder of the packet. The last of these six advances writes the last line of the packet and this line is partially filled with *six* bytes. The 10^{th} and last advance will contain the EDW for the transaction.

5.3.3 Timing examples

A hypothetical timing diagram for the packet illustrated in Figure 39 is shown in Figure 40. Recall that this packet is contained in nine lines.

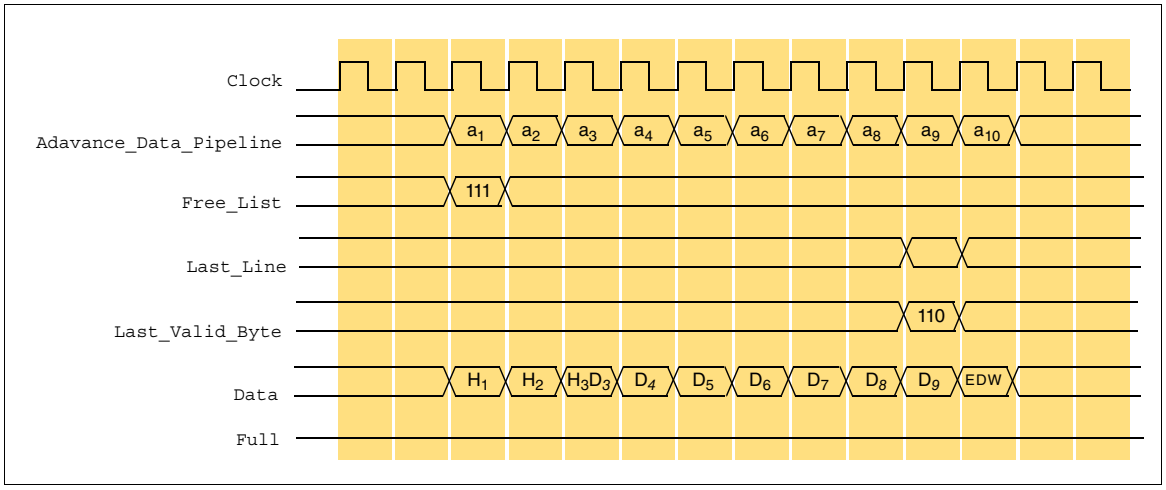

Figure 40 Timing diagram (one transfer) for the PIB

The protocol core asserts `Advance_Data_Pipeline`. This signals the block that the protocol core has a packet to import. Coincident with that advance the protocol core asserts both a freelist number and the first line of packet data. The value asserted to the `Free_List` port specifies the FLB (see Section 4.9) from which the memory for the packet will be allocated. Eight additional advances are then asserted to write the remainder of the packet. On the *ninth* advance the protocol core asserts `Last_Line`. Coincident with that assertion, the protocol core also loads the `Last_Valid_Byte` port with a value of *six* indicating that the last line contains only six valid bytes. Once `Last_Line` is deasserted the entire packet has been transferred. Immediately following this deassertion the protocol core completes the transaction by writing the appropriate EDW (see Section 5.3.4) to the `Data` port.

As another example, assume the protocol core receives three packets back-to-back and each packet consists of only a sixteen byte header with no payload. The timing diagram for this scenario is illustrated in Figure 41:

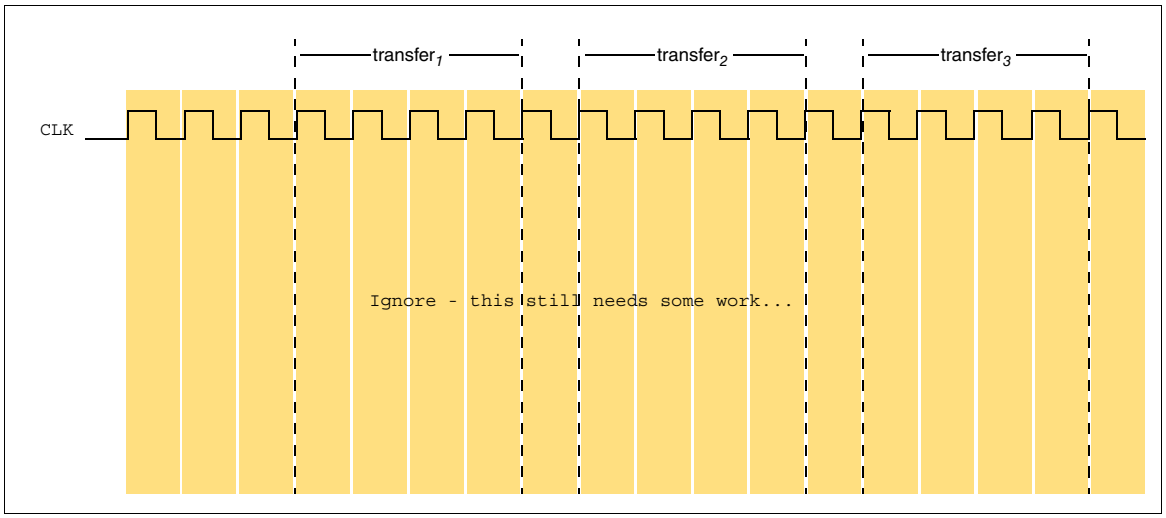

Figure 41 Timing diagram for PIB three packet transfer

5.3.4 Import Post-processing

The protocol core must follow both the generic guidelines for post-processing as defined in Section 5.1.2. and the specific guidelines for importing as outlined below:

The transfer succeeds: In such a case the protocol core's responsibility is *two*-fold:

- It must assert completion to the block. Completion is asserted by writing *once* to the block's `Data` port. The value must be asserted *after* `Last_line`.
- The asserted value must correspond to the EDW for the transaction (see Section 4.3). The constraints on the values of its fields in this case are as follows:
  - The ***WasError*** field of the must be *false* (*clear*).
  - The ***Reason*** field is "Don't Care".
  - the ***WasBlock*** field is "Don't Care".
  - The ***Parameter*** field is "Don't Care".

The transfer fails due to a data over-run error: This is an example of the type of error which had its origin in the system, but which can only be detected by the protocol core. The definition of this type of error is as follows:

`Data_Available` is deasserted *before* `Data_Last` during packet transmission.

In such a case the protocol core's responsibility is *three*-fold:

- It must detect this type of error.
- It must assert completion to the block. Completion is asserted by writing *once* to the block's `Data` port. The value must be asserted *after* `Last_line`.
- The asserted value must correspond to the EDW for the transaction (see Section 4.3). The constraints on the values of its fields in this particular case are as follows:
  - The ***WasError*** field of the must be *true* (*set).*

— The ***WasBlock*** field must be *true (set).*

— The ***Reason*** field must have the value `DATA_OVER_RUN` (62 decimal).

— The ***Parameter*** field must be *zero*.

The transfer fails due to a protocol error: In such a case the protocol core's responsibility is *two*-fold:

— It must assert completion to the block. Completion is asserted by writing *once* to the block's `Data` port. The value must be asserted *after* `Last_line`.

— The asserted value must correspond to the EDW for the transaction (see Section 4.3). The constraints on the values of its fields in this case are as follows:

— The ***WasError*** field of the must be *true* (*set).*

— The ***WasBlock*** field must be *false (clear).*

— The ***Reason*** field must have a value appropriate to the transfer error. All unique errors should be assigned unique numbers.

— The ***Parameter*** field should have a value appropriate to the error.

In short the protocol core must adhere to the following rules:

— Independent of success or failure, transfer completion must be signalled by writing *once* and *only* once per transaction. Completion must be signalled *after* `Last_line` has been asserted.

— Completion is signalled by writing an EDW appropriate to the transaction. The EDW is written to the block's `Data` port. The appropriate value for the EDW is determined by whether or not the transaction failed and if it did fail, the nature of that failure.

Chapter 6 The PIC DCR Interface

6.1 The PEB (Pending Export Block)

The Back-End interface to the PEB consists principally of four registers[1], one of which is reserved for future use. One of the four registers is used for setup and control of the PEB, one register is an interface to the *write* port of the PEB's transaction FIFO and one register contains the potential fault parameter. The location of these registers on the DCR bus relative to one another is enumerated in Table 8. The absolute location of these registers is specified as a block instantiation parameter and is discussed in Section 1.3. For a detailed discussion of the functionality of the PEB see Section 4.7.

Table 8 Register offsets for the PEB

Offset[1]	Name	Description
0	PEB_CSR	Control and Status Register. See Section 6.1.1
1	EXPORT_PENDING	*Write* port of the block's transaction FIFO. See Section 6.1.2
2	EXPORT_FAULT	Contains the TDE associated with a fault. See Section 6.1.3
3	PEB_RESERVED	Must be *zero*.

1. In words

6.1.1 Control and Status Register (CSR)

This register manages the configuration and setup of the block. Principally, this implies establishing the conditions under which the block will assert its Event signal. This signal is connected to the ISB (see Section 6.5) where it may be used to trigger a processor interrupt.

1. See also the interrupt interface described in Section 6.5.1.

Note: this register provides for selective *set* and *clear* of its fields. The low order 16 bits constitute the fields which can be changed and the high order 16 bits are the corresponding field enables. See Section 1.3.2 for a discussion of these types of fields.

The structure of this register is illustrated in Figure 42:

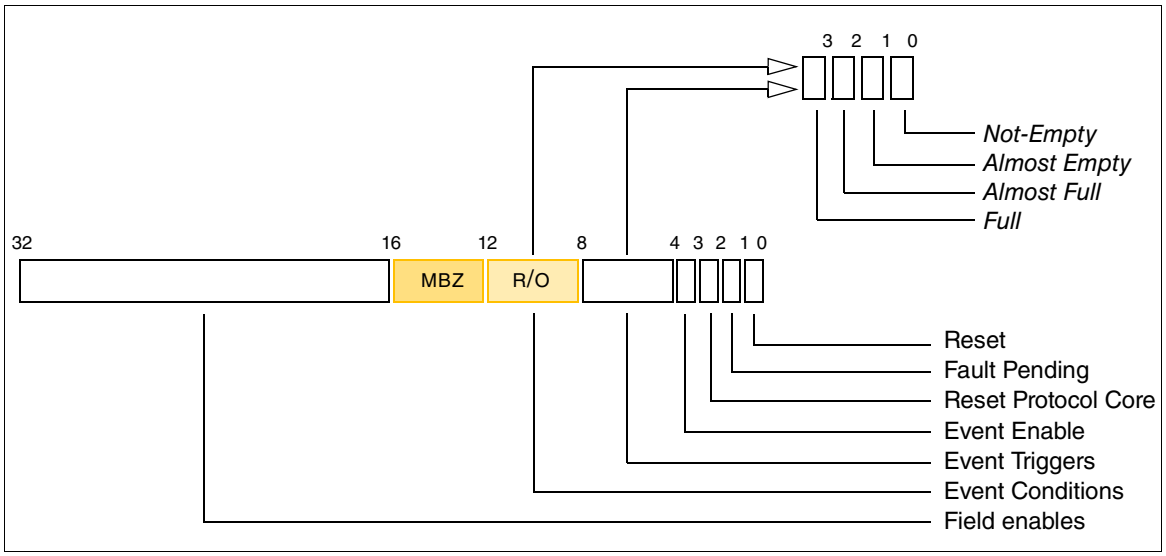

Figure 42 PEB CSR register

Where:

Reset: This field is used to reset the corresponding block. This field may only be *set*. Any access which attempts to clear this field will be ignored. Asserting (*setting*) this field triggers a reset of the block. While the reset is in progress the field remains *set*. The value of any other field of the register during this time is indeterminate. After the reset is complete, the field is *cleared*. While a reset is in progress any requests to modify the block's state are ignored. This includes writing fields of this register or any other register, as well as any operation which implicitly or explicitly writes to the block's FIFOs. The result of reading a a block's transaction FIFO while a reset is in progress are indeterminate. A reset flushes all entries buffered in any of the block's FIFOs, resets all its DCR registers, and returns all the block's internal *State Machines* to their initial state. Therefore, after reset is complete:

- all FIFOs are *empty*
- the `Event` signal is masked
- a pending fault (if any) is flushed
- the ***Event Triggers*** and ***Event Conditions*** fields are set to *zero*

Fault Pending: This field indicates the presence of a fault. If this field is *set*, a fault is pending. To dismiss a pending fault this field must be *cleared*. See Section 4.6 for the description of a fault.

Reset Protocol core: This field is used to reset the protocol core. In actuality, setting this field asserts a pulse on the `Core_Reset` signal of the PEB's Front-end-Interface (see Section 5.2.2.11). This field may only be *set*. Any access which attempts to clear this field will be ignored.

Event Enable: This field allows the block to assert its `Event`signal whenever the conditions specified by the Event Triggers field (see below) are true. When this field is *set*, the `Event` signal is *enabled*. When this field is *clear*, the signal is masked (*disabled*). See Section 4.5 for the description of an event.

Event Triggers: This field enumerates which combination of the four possible conditions of the block's transaction FIFO will assert the block's `Event` signal (Assuming events are enabled, see above). In turn, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set* and the corresponding condition in the transaction FIFO is present, the `Event` signal will be asserted. If a field is *clear*, independent of the state of the corresponding condition in the transaction FIFO, the `Event` signal will *not* be asserted.See Section 4.5 for the description of an event and its corresponding signal.

Event Conditions: This field enumerates which combination of the four possible transaction FIFO conditions are currently asserted. As with the ***Event Triggers*** field described above, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set*, the corresponding condition is present. If a field is *clear*, the condition is *not* present. The values of this field are independent of the state of the ***Event Enable*** field (see above). Note: This field is *Read-Only*. See Section 4.5 for the description of an event and its corresponding signal.

Field enables: The low-order 16 bits of this register are *Selective Set and Clear*. This field forms the *write* enables for those 16 bits. See Section 1.3.2 for an explanation of how this field is to be used.

6.1.2 Export Pending Register

This register is an interface to the *write* port of the block's transaction FIFO. When written, the corresponding value is inserted at the FIFO's tail. If, when written, the FIFO is *full*, the written value is discarded and the block generates a ***Export FIFO Full*** fault (see Section 4.7.4.3). The structure of any value written to the FIFO follows the conventions specified for a Transfer Descriptor Entry (TDE). The structure of the TDE for this particular type of block is described in Section 4.7.2. When read, the returned value is always *zero*. The structure of this register is illustrated in Figure 43:

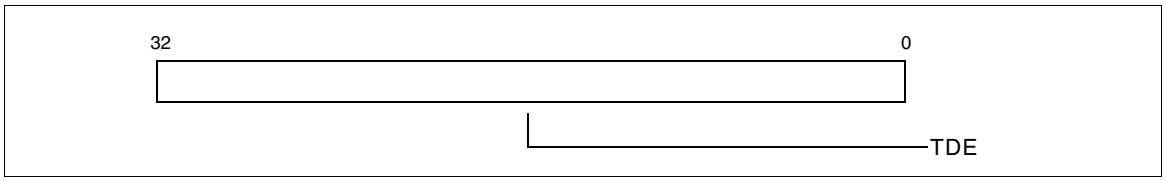

Figure 43 Export Pending Register

6.1.3 Export Fault Register

In the eventuality that usage of the block asserts an fault, this register contains the offending TDE. The structure of a TDE for this particular type of block is described in Section 4.7.2. The contents of this register are only valid while the ***Fault Pending*** field of the CSR register is asserted. See Sections 4.6, 4.7.3 and 4.7.4 for a discussion of fault processing. The structure of this register is illustrated in Figure 44:

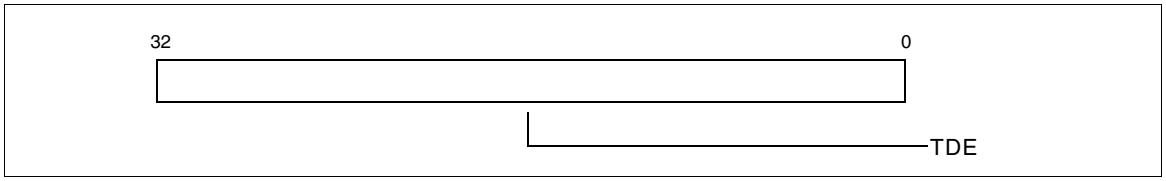

Figure 44 Export Fault Register

6.2 The ECB (Export Complete Block)

The Back-End interface to the ECB consists principally of four registers, one of which is reserved for future use. One of the four registers is used for setup and control of the PEB, one register is an interface to the *read* port of the ECB's transaction FIFO and one register contains the potential fault parameter. The location of these registers on the DCR bus relative to one another is enumerated in Table 8. The absolute location of these registers is specified as a block instantiation parameter and is discussed in Section 1.3. For a detailed discussion of the functionality of the ECB see Section 4.8.

Table 9 Register offsets for the ECB

Offset[1]	Name	Description
0	ECB_CSR	Control and Status Register. See Section 6.2.1
1	EXPORT_COMPLETE	*Read* port of the transaction FIFO. See Section 6.2.2
2	EXPORT_COMPLETE_FAULT	Contains the TDE associated with an fault. See Section 6.2.3
3	ECB_RESERVED	Must be *zero*.

1. In bytes

6.2.1 Control and Status Register (CSR)

This register manages the configuration and setup of the block. Principally, this implies establishing the conditions under which the block will assert its `Event` signal. This signal is connected to the ISB (see Section 6.5) where it may be used to trigger a processor interrupt.

Note: this register provides for selective *set* and *clear* of its fields. The low order 16 bits constitute the fields which can be changed and the high order 16 bits are the corresponding field enables. See Section 1.3.2 for a discussion of these types of fields.

The structure of this register is illustrated in Figure 45:

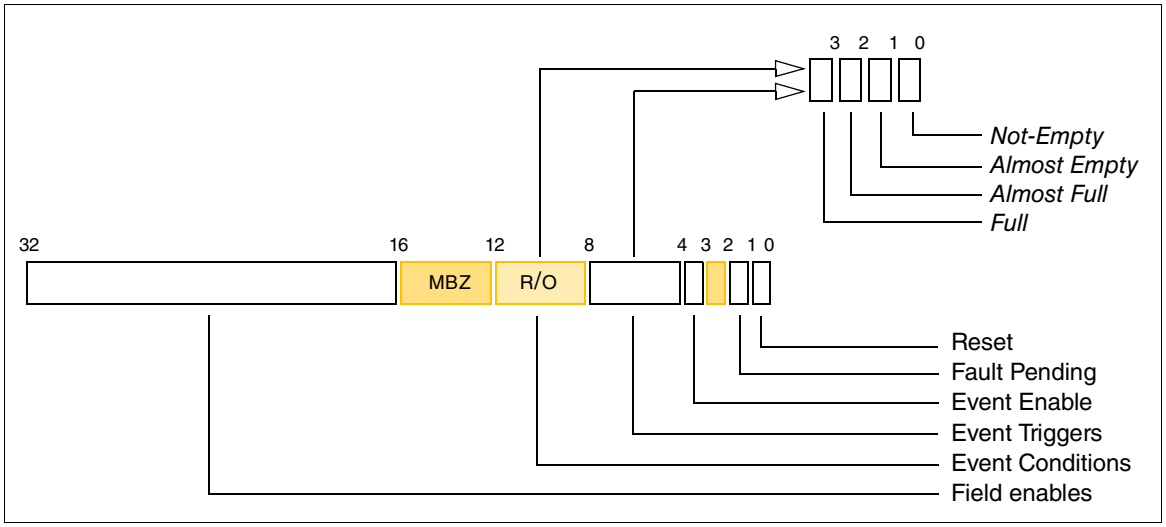

Figure 45 ECB CSR register

Where:

Reset: This field is used to reset the corresponding block. This field may only be *set*. Any access which attempts to clear this field will be ignored. Asserting (*setting*) this field triggers a reset of the block. While the reset is in progress the field remains *set*. The value of any other field of the register during this time is indeterminate. After the reset is complete, the field is *cleared*. While a reset is in progress any requests to modify the block's state are ignored. This includes writing fields of this register or any other register, as well as any operation which implicitly or explicitly writes to the block's FIFOs. The result of reading a a block's transaction FIFO while a reset is in progress are indeterminate. A reset flushes all entries buffered in any of the block's FIFOs, resets all its DCR registers, and returns all the block's internal *State Machines* to their initial state. Therefore, after reset is complete:

— all FIFOs are *empty*

— the `Event` signal is masked

— a pending fault (if any) is flushed

— the ***Event Triggers*** and ***Event Conditions*** fields are set to *zero*

Fault Pending: This field indicates the presence of a fault. If this field is *set*, a fault is pending. To dismiss a pending fault this field must be *cleared*. See Section 4.6 for the description of a fault.

Event Enable: This field allows the block to assert its `Event`signal whenever the conditions specified by the Event Triggers field (see below) are true. When this field is *set*, the `Event` signal is *enabled*. When this field is *clear*, the signal is masked (*disabled*). See Section 4.5 for the description of an event.

Event Triggers: This field enumerates which combination of the four possible conditions of the block's transaction FIFO will assert the block's `Event` signal (Assuming events are enabled, see above). In turn, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set* and the corresponding condition in the transaction FIFO is present, the `Event` signal will be asserted. If a field is *clear*, independent of the state of the corresponding condition in the transaction FIFO, the `Event` signal will *not* be asserted.See Section 4.5 for the description of an event and its corresponding signal.

Event Conditions: This field enumerates which combination of the four possible transaction FIFO conditions are currently asserted. As with the ***Event Triggers*** field described above, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set*, the corresponding condition is present. If a field is *clear*, the condition is *not* present. The values of this field are independent of the state of the ***Event Enable*** field (see above). Note: This field is *Read-Only.* See Section 4.5 for the description of an event and its corresponding signal.

Field enables: The low-order 16 bits of this register are *Selective Set and Clear.* This field forms the *write* enables for those 16 bits. See Section 1.3.2 for an explanation of how this field is to be used.

6.2.2 Export Complete Register

This register is an interface to the *read* port of the block's transaction FIFO. When read, the returned value corresponds to the entry at the FIFO's head. The structure of any returned entry follows the conventions specified for a Transfer Descriptor Entry (TDE). The structure of the TDE for this particular register is described in Section 4.8.2. If the returned value is *four* (4), the FIFO was *empty.* The structure of this register is illustrated in Figure 46:

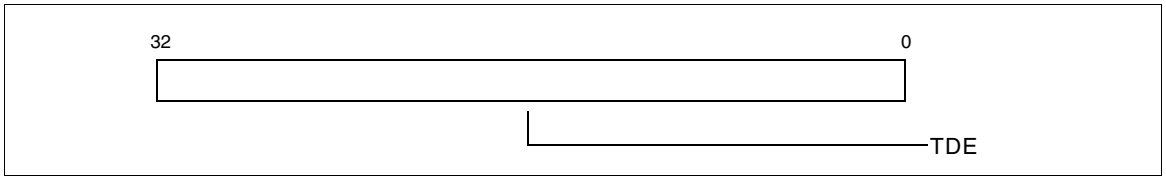

Figure 46 Export Complete Register

6.2.3 Export Complete Fault Register

In the eventuality that usage of the block asserts an fault, this register contains the offending TDE. The structure of a TDE for this particular type of block is described in Section 4.8.2. The contents of this register are only valid while the ***Fault Pending*** field of the CSR register is asserted. See Sections 4.6, 4.7.3 and 4.7.4 for a discussion of fault processing. The structure of this register is illustrated in Figure 47:

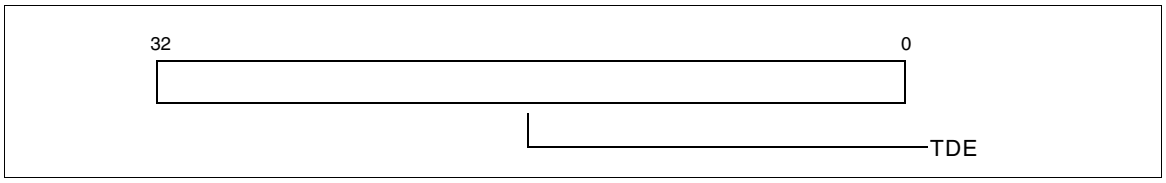

Figure 47 Export Complete Fault Register

6.3 The FLB (Freelist Block)

The Back-End interface to the ECB consists principally of four registers[1], one of which is reserved for future use. One of the four registers is used for setup and control of the FLB, one register is an interface to the *write* port of the FLB's transaction FIFO and one register contains the potential fault parameter. The location of these registers on the DCR bus relative to one another is enumerated in Table 10. The absolute location of these registers is specified as a block instantiation parameter and is discussed in Section 1.3. For a detailed discussion of the functionality of the ECB see Section 4.8.3.

Table 10 Register offsets for the FLB

Offset[1]	Name	Description
0	FLB_CSR	Control and Status Register. See Section 6.3.1
1	FREELIST	*Write* port of the block's transaction FIFO. See Section 6.3.2
2	FREELIST_FAULT	Contains the TDE associated with an fault. See Section 6.3.3
3	FLB_RESERVED	Must be *zero*.

1. In bytes

1. See also the interrupt interface described in Section 6.5.1.

6.3.1 Control and Status Register (CSR)

This register manages the configuration and setup of the block. Principally, this implies establishing the conditions under which the block will assert its `Event` signal. This signal is connected to the ISB (see Section 6.5) where it may be used to trigger a processor interrupt.

Note: this register provides for selective *set* and *clear* of its fields. The low order 16 bits constitute the fields which can be changed and the high order 16 bits are the corresponding field enables. See Section 1.3.2 for a discussion of these types of fields.

The structure of this register is illustrated in Figure 48:

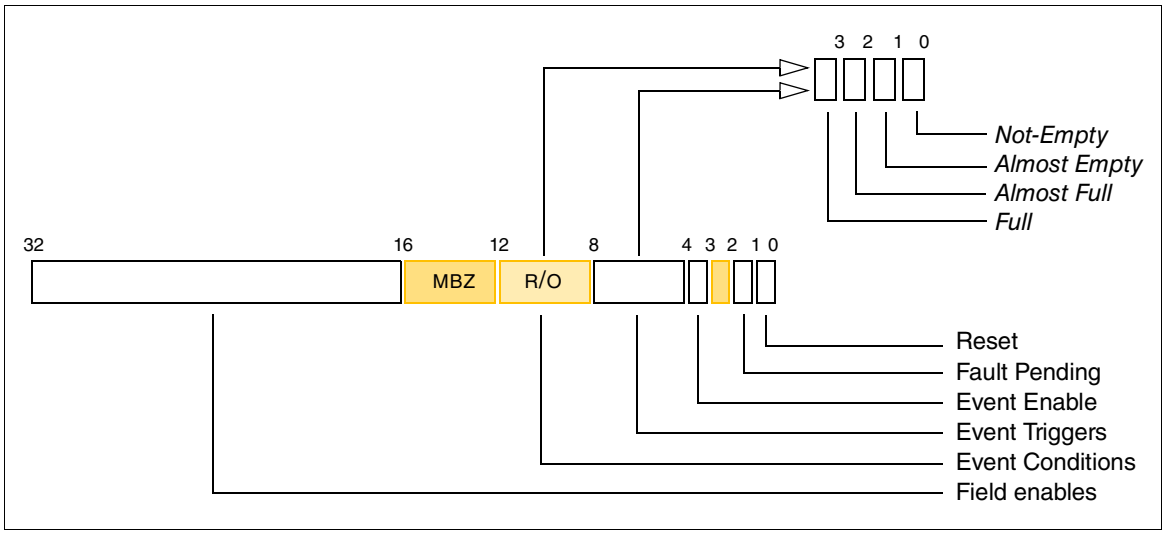

Figure 48 FLB CSR register

Where:

Reset: This field is used to reset the corresponding block. This field may only be *set*. Any access which attempts to clear this field will be ignored. Asserting (*setting*) this field triggers a reset of the block. While the reset is in progress the field remains *set*. The value of any other field of the register during this time is indeterminate. After the reset is complete, the field is *cleared*. While a reset is in progress any requests to modify the block's state are ignored. This includes writing fields of this register or any other register, as well as any operation which implicitly or explicitly writes to the block's FIFOs. The result of reading a a block's transaction FIFO while a reset is in progress are indeterminate. A reset flushes all entries buffered in any of the block's FIFOs, resets all its DCR registers, and returns all the block's internal *State Machines* to their initial state. Therefore, after reset is complete:

- all FIFOs are *empty*
- the `Event` signal is masked
- a pending fault (if any) is flushed
- the ***Event Triggers*** and ***Event Conditions*** fields are set to *zero*

Fault Pending: This field indicates the presence of a fault. If this field is *set*, a fault is pending. To dismiss a pending fault this field must be *cleared*. See Section 4.6 for the description of a fault.

Event Enable: This field allows the block to assert its `Event`signal whenever the conditions specified by the Event Triggers field (see below) are true. When this field is *set*, the `Event` signal is *enabled*. When this field is *clear*, the signal is masked (*disabled*). See Section 4.5 for the description of an event.

Event Triggers: This field enumerates which combination of the four possible conditions of the block's transaction FIFO will assert the block's `Event` signal (Assuming events are enabled, see above). In turn, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set* and the corresponding condition in the transaction FIFO is present, the `Event` signal will be asserted. If a field is *clear*, independent of the state of the corresponding condition in the transaction FIFO, the `Event` signal will *not* be asserted.See Section 4.5 for the description of an event and its corresponding signal.

Event Conditions: This field enumerates which combination of the four possible transaction FIFO conditions are currently asserted. As with the ***Event Triggers*** field described above, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set*, the corresponding condition is present. If a field is *clear*, the condition is *not* present. The values of this field are independent of the state of the ***Event Enable*** field (see above). Note: This field is *Read-Only.* See Section 4.5 for the description of an event and its corresponding signal.

Field enables: The low-order 16 bits of this register are *Selective Set and Clear*. This field forms the *write* enables for those 16 bits. See Section 1.3.2 for an explanation of how this field is to be used.

6.3.2 Freelist Register

This register is an interface to the *write* port of the block's transaction FIFO. When written, the corresponding value is inserted at the FIFO's tail. If, when written, the FIFO is *full*, the written value is discarded and the block generates a ***Freelist Full*** fault (see Section 4.9.4.2). The structure of any value written to the FIFO follows the conventions specified for a Transfer Descriptor Entry (TDE). The structure of the TDE for this particular type of block is described in Section 4.9.2. When read, the returned value is always *zero*. The structure of this register is illustrated in Figure 49:

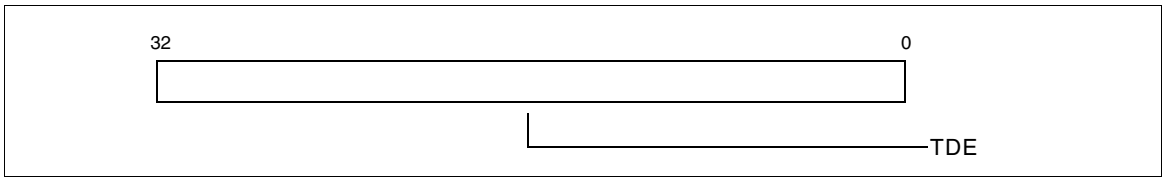

Figure 49 Freelist Register

6.3.3 Freelist Fault Register

n the eventuality that usage of the block asserts an fault, this register contains the offending TDE. The structure of a TDE for this particular type of block is described in Section 4.9.2. The contents of this register are only valid while the ***Fault Pending*** field of the CSR register is asserted. See Sections 4.6, 4.7.3 and 4.7.4 for a discussion of fault processing. The structure of this register is illustrated in Figure 50:

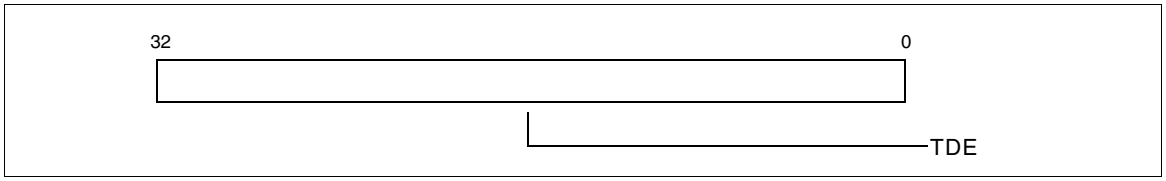

Figure 50 Freelist Fault Register

6.4 The (PIB) Pending Import Block

The Back-End interface to the PIB consists principally of four registers[1], one of which is reserved for future use. One of the four registers is used for setup and control of the PIB, one register is an interface to the *read* port of the PIB's transaction FIFO and one register contains the potential fault parameter. The location of these registers on the DCR bus relative to one another is enumerated in Table 11. The absolute location of these registers is specified as a block instantiation parameter and is discussed in Section 1.3. For a detailed discussion of the functionality of the PIB see Section 4.10.

Table 11 Register offsets for the PIB

Offset[1]	Name	Description
0	PIB_CSR	Control and Status Register. See Section 6.4.1
1	IMPORT_PENDING	*Read* port of the block's transaction FIFO. See Section 6.4.2
2	IMPORT_FAULT	Contains the TDE associated with an fault. See Section 6.4.3
3	PIB_RESERVED	Must be *zero*.

1. See also the interrupt interface described in Section 6.5.1.

1. In bytes

6.4.1 Control and Status Register (CSR)

This register manages the configuration and setup of the block. Principally, this implies establishing the conditions under which the block will assert its `Event` signal. This signal is nominally connected to the ISB (see Section 6.5) where it may used to trigger a processor interrupt. Note: this register provides for selective *set* and *clear* of its fields. The low order 16 bits constitute the fields which can be changed and the high order 16 bits are the corresponding field enables. See Section 1.3.2 for a discussion of these types of fields. The structure of this register is illustrated in Figure 51:

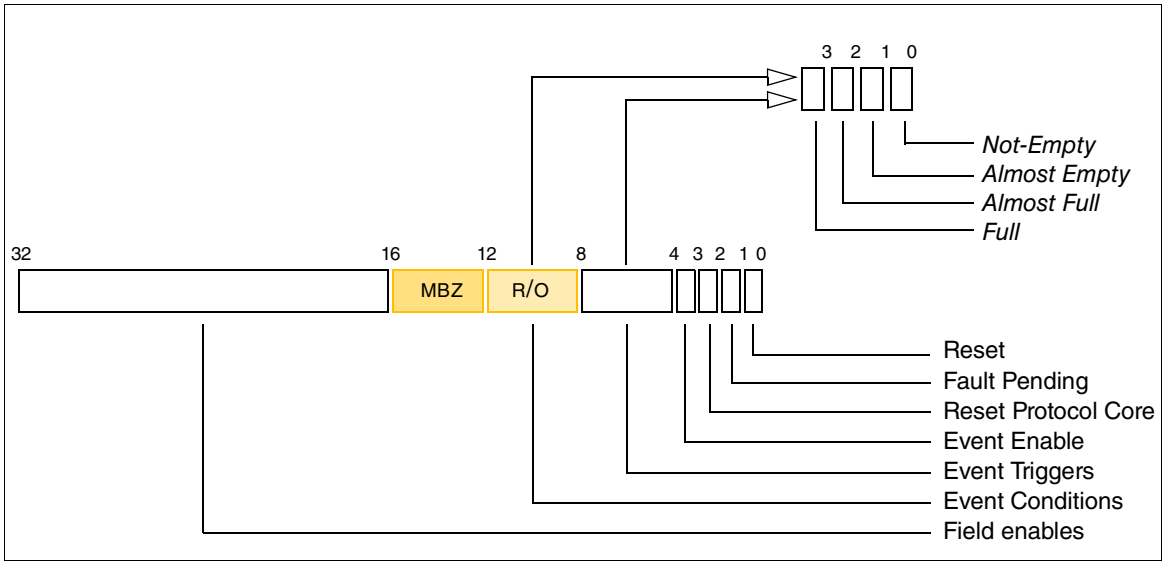

Figure 51 PIB CSR register

Where:

Reset: This field is used to reset the corresponding block. This field may only be *set*. Any access which attempts to clear this field will be ignored. Asserting (*setting*) this field triggers a reset of the block. While the reset is in progress the field remains *set*. The value of any other field of the register during this time is indeterminate. After the reset is complete, the field is *cleared*. While a reset is in progress any requests to modify the block's state are ignored. This includes writing fields of this register or any other register, as well as any operation which implicitly or explicitly writes to the block's FIFOs. The result of reading a a block's transaction

FIFO while a reset is in progress are indeterminate. A reset flushes all entries buffered in any of the block's FIFOs, resets all its DCR registers, and returns all the block's internal *State Machines* to their initial state. Therefore, after reset is complete:

— all FIFOs are *empty*

— the `Event` signal is masked

— a pending fault (if any) is flushed

— the ***Event Triggers*** and ***Event Conditions*** fields are set to *zero*

Fault Pending: This field indicates the presence of a fault. If this field is *set*, a fault is pending. To dismiss a pending fault this field must be *cleared*. See Section 4.6 for the description of a fault.

Reset Protocol core: This field is used to reset the protocol core. In actuality, setting this field asserts a pulse on the `Core_Reset` signal of the PEB's Front-end-Interface (see Section 5.2.2.11). This field may only be *set*. Any access which attempts to clear this field will be ignored.

Event Enable: This field allows the block to assert its `Event`signal whenever the conditions specified by the Event Triggers field (see below) are true. When this field is *set*, the `Event` signal is *enabled*. When this field is *clear*, the signal is masked (*disabled*). See Section 4.5 for the description of an event.

Event Triggers: This field enumerates which combination of the four possible conditions of the block's transaction FIFO will assert the block's `Event` signal (Assuming events are enabled, see above). In turn, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set* and the corresponding condition in the transaction FIFO is present, the `Event` signal will be asserted. If a field is *clear*, independent of the state of the corresponding condition in the transaction FIFO, the `Event` signal will *not* be asserted.See Section 4.5 for the description of an event and its corresponding signal.

Event Conditions: This field enumerates which combination of the four possible transaction FIFO conditions are currently asserted. As with the ***Event Triggers*** field described above, this field is divided into four sub-fields with the offset of each field corresponding to a specific condition. If a sub-field is *set*, the corresponding condition is present. If a field is *clear*, the condition is *not* present. The values of this field are independent of the state of the ***Event Enable*** field (see above). Note: This field is *Read-Only.* See Section 4.5 for the description of an event and its corresponding signal.

Field enables: The low-order 16 bits of this register are *Selective Set and Clear.* This field forms the *write* enables for those 16 bits. See Section 1.3.2 for an explanation of how this field is to be used.

6.4.2 Import Pending Register

This register is an interface to the *read* port of the block's transaction FIFO. When read, the returned value corresponds to the entry at the FIFO's head. The structure of any returned entry follows the conventions specified for a Transfer Descriptor Entry (TDE). The structure of the TDE for this particular register is described in Section 4.10.2. If the returned value is *four* (4), the FIFO was *empty*. The structure of this register is illustrated in Figure 52:

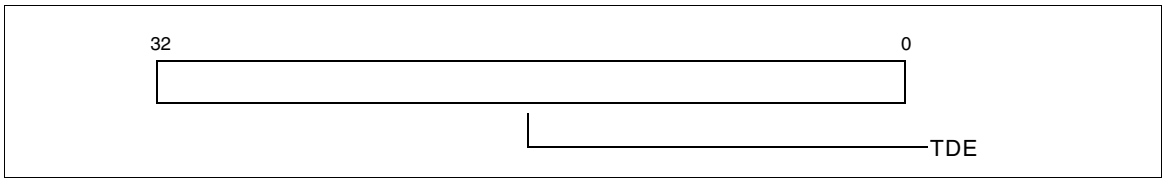

Figure 52 Import Pending Register

6.4.3 Import Fault Register

In the eventuality that usage of the block asserts an fault, this register contains the offending TDE. The structure of a TDE for this particular type of block is described in Section 4.10.2. The contents of this register are only valid while the ***Fault Pending*** field of the CSR register is asserted. See Sections 4.6, 4.7.3 and 4.7.4 for a discussion of fault processing. The structure of this register is illustrated in Figure 53:

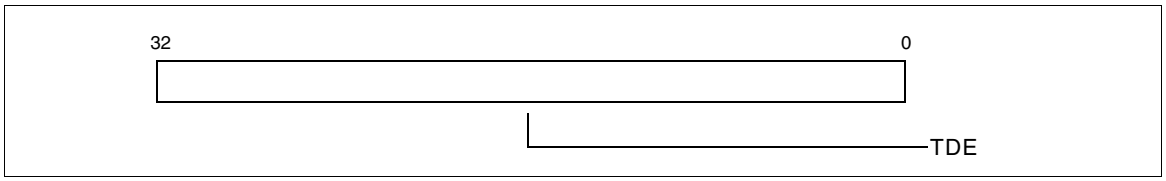

Figure 53 Import Fault Register

6.5 The ISB (Interrupt Summary Block)

The Back-End interface to the ISB consists of four registers. These registers are *Read-Only* and simply reflect the current state of the up to 128 `Event` and `Fault` signals generated by the transfer blocks of a system. The location of these registers on the DCR bus relative to one another is enumerated in Table 12. The absolute location of these registers is specified as a block instantiation parameter. For a detailed discussion of the functions of the ISB see Section 4.11.

Table 12 Register offsets for the ISB

Offset[1]	Name	Description
0	EVENT_SRCS_LOW	State of Event lines for PEB and ECB
1	EVENT_SRCS_HIGH	State of Event lines for FLB and PIB
2	FAULT_SRCS_LOW	State of Fault lines for PEB and ECB
3	FAULT_SRCS_HIGH	State of Fault lines for FLB and PIB

1. In decimal

6.5.1 Event Sources (Low) Register

This register specifies the current state of the Event signals for both PEB and ECB. The register contains two sixteen bit fields. The low-order field corresponds to the set of Event signals for the PEB, while the high-order field corresponds to the set of Event signals for the ECB. Each field is a bit-list, where the value at any given offset corresponds to the state of the Event signal for the corresponding block. If the value pointed to by the offset is *set*, the Event signal for the corresponding block is asserted. If the value pointed to by the offset is *clear*, the signal is *not* asserted. The structure of this register is illustrated in Figure 54:

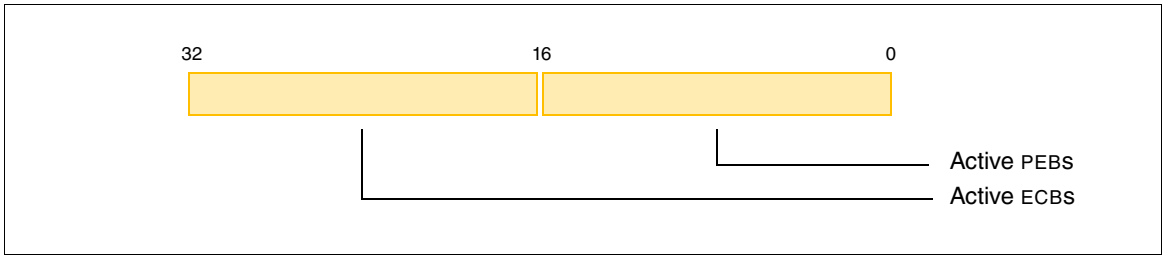

Figure 54 ISB event register (Low)

6.5.2 Event Sources (High) Register

This register specifies the current state of the Event signals for both FLB and PIB. The register contains two sixteen bit fields. The low-order field corresponds to the set of Event signals for the FLB, while the high-order field corresponds to the set of Event signals for the PIB. Each field is a bit-list, where the value at any given offset corresponds to the state of the Event signal for the corresponding block. If the value pointed to by the offset is *set*, the Event signal for the corresponding block is asserted. If the value pointed to by the offset is *clear*, the signal is *not* asserted. The structure of this register is illustrated in Figure 55:

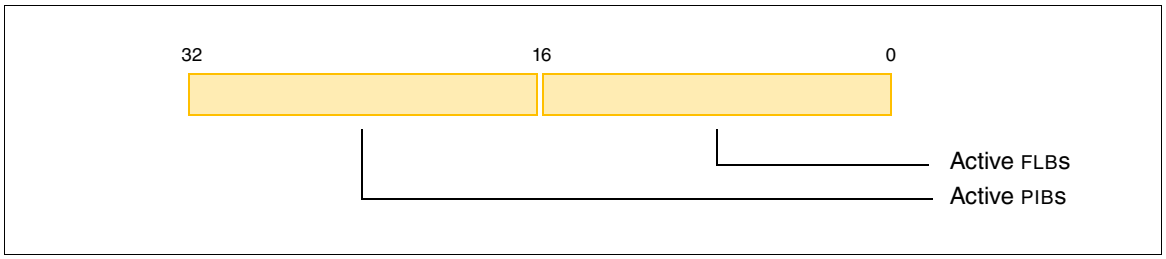

Figure 55 ISB event register (High)

6.5.3 Fault Sources (Low) Register

This register specifies the current state of the `Fault` signals for both PEB and ECB. The register contains two sixteen bit fields. The low-order field corresponds to the set of `Fault` signals for the PEB, while the high-order field corresponds to the set of `Fault` signals for the ECB. Each field is a bit-list, where the value at any given offset corresponds to the state of the `Fault` signal for the corresponding block. If the value pointed to by the offset is *set*, the `Fault` signal for the corresponding block is asserted. If the value pointed to by the offset is *clear*, the signal is *not* asserted. The structure of this register is illustrated in Figure 56:

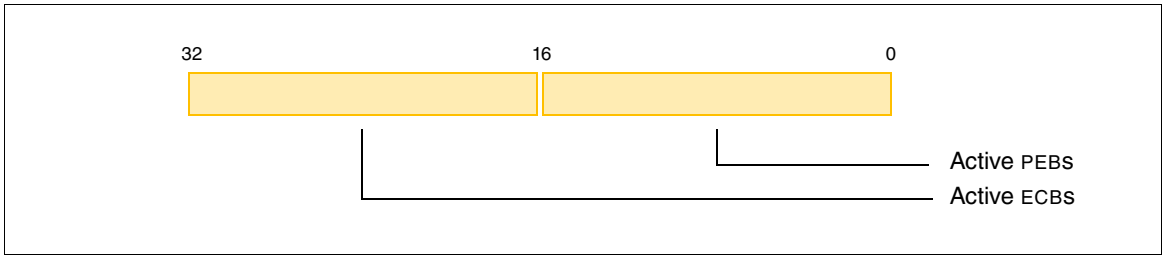

Figure 56 ISB fault register (Low)

6.5.4 Fault Sources (High) Register

This register specifies the current state of the `Fault` signals for both FLB and PIB. The register contains two sixteen bit fields. The low-order field corresponds to the set of `Fault` signals for the FLB, while the high-order field corresponds to the set of `Fault` signals for the PIB. Each field is a bit-list, where the value at any given offset corresponds to the state of the `Event` signal for the corresponding block. If the value pointed to by the offset is *set*, the `Fault` signal for the corresponding block is asserted. If the value pointed to by the offset is *clear*, the signal is *not* asserted. The structure of this register is illustrated in Figure 57:

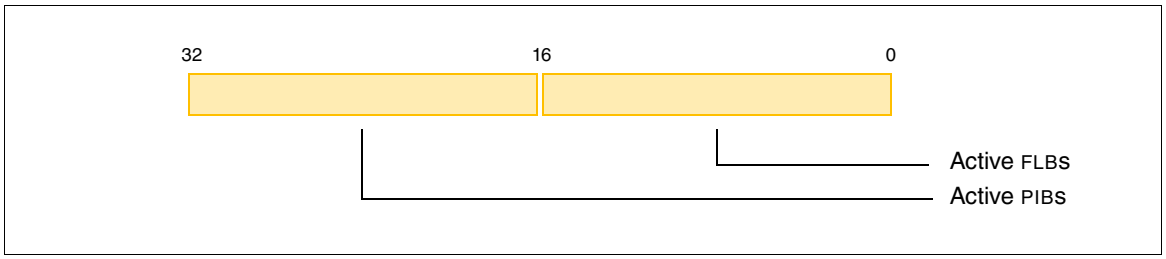

Figure 57 ISB fault register (High)