Particle Physics & Astrophysics

The Reconfigurable Cluster Element

An I/O System On Chip

Cluster Element (CE) User Guide

Document Version:	0.9
Document Issue:	3
Document Edition:	English
Document Status:	First release to reviewers
Document ID:	XXX-TD-xxxxx
Document Date:	December 18, 2008

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

Abstract

To be written.

Hardware compatibility

This document assumes the following hardware revisions: TBD.

Intended audience

TBW.

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

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

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	The Reconfigurable Cluster Element Cluster Element (CE) User Guide		
	Version:	0.9		
	Issue:	3		
	Edition:	English		
	ID:	XXX-TD-xxxxx		
	Status:	First release to reviewers		
	Created:	February 9, 2002		
	Date:	December 18, 2008		
	Access:	Z:\Private\DTK\RCE\UG\V5\frontmatter.fm		
	Keywords:	CSC ROD		
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, SLAC		
	Written by:	Michael Huffer		
	Reviewed by:	N/A		
	Approved by:	N/A		

Document Status Sheet

Table 2 Document Status Sheet

Title:	The Reconfigurable Cluster Element Cluster Element (CE) User Guide		
ID:	XXX-TD-xxxxx		
Version	**Issue**	**Date**	**Reason for change**
0.1	0	4/16/2009	Initial draft
0.9	3	5/28/2014	Corrected typos and figures 5 and 35. Corrected Table 9.

Table of Contents

List of Figures

List of Tables

Chapter 1 Overview

1.1 Introduction

For the purposes of this document a *Protocol-Plug-In* is defined as simply an arbitrary set of application specific logic, coresident with an RCE's FPGA fabric, but which, requires the exchange of information between it and its corresponding RCE. In turn, that requirement implies the necessity for a common, abstract communication mechanism between plug-in and RCE. That abstract model is the *Plug* and *Socket*. The *Plug* is a set of *Core* IP, provided by the system to the user which wraps a common communication interface around their specific plug-in. Once wrapped, that plug-in can subsequently, independent of implementation or specific function, be "plugged into" one of the *four* (4) predefined *Sockets* contained on an RCE, thus enabling communication between it and its RCE. The relationship between the RCE, its sockets and plug-ins and their corresponding plugs is illustrated in Figure 1:

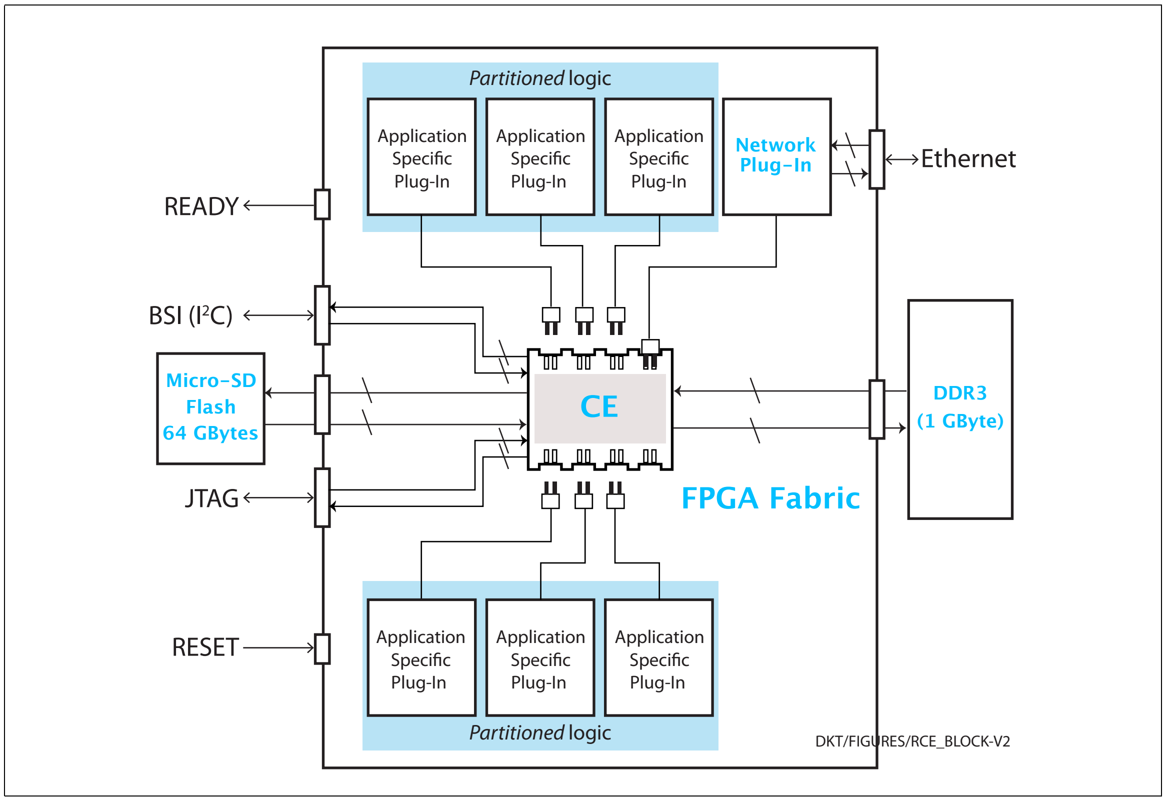
Partitioned logic
Application Specific Plug-In
Application Specific Plug-In
Application Specific Plug-In
Network Plug-In
Ethernet
READY
BSI (I²C)
Micro-SD Flash 64 GBytes
JTAG
RESET
CE
FPGA Fabric
DDR3 (1 GByte)
Application Specific Plug-In
Application Specific Plug-In
Application Specific Plug-In
Partitioned logic
DKT/FIGURES/RCE_BLOCK-V2

Figure 1 RCE block diagram

TBD

1.2 The CE (Cluster Element)

TBD. The relationship between the CE, its sockets and its plug-ins is illustrated in Figure 2:

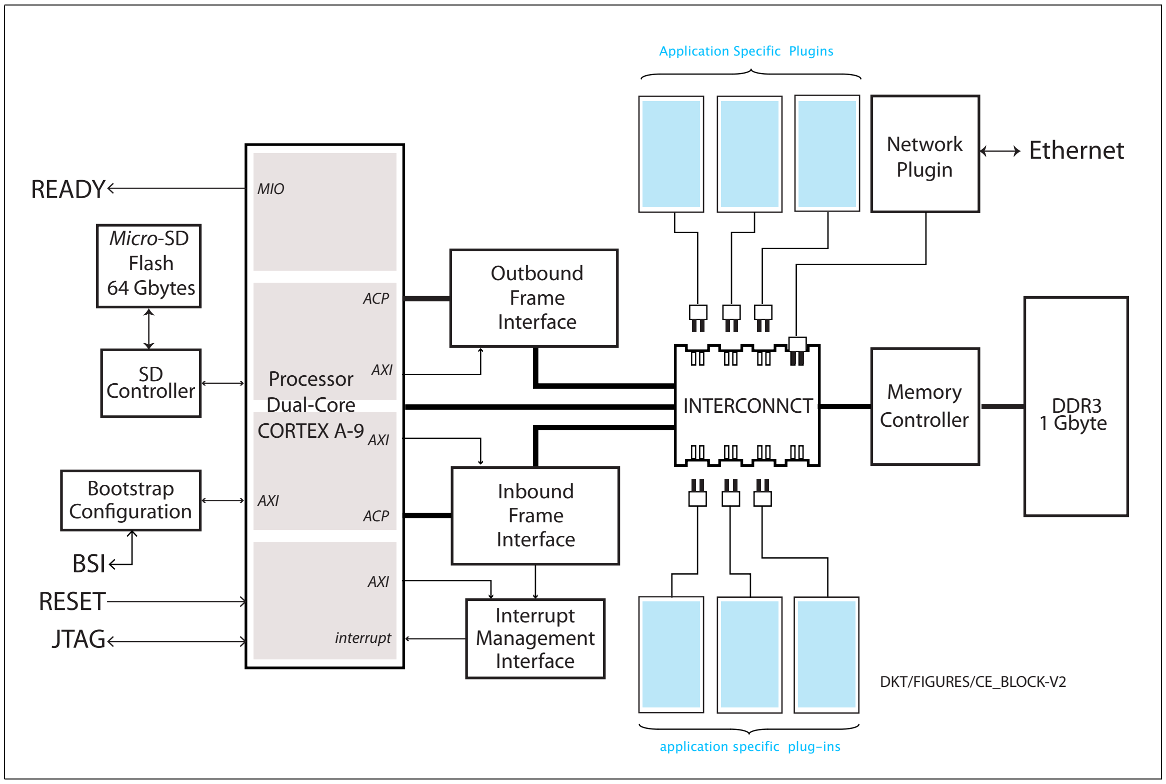

Figure 2 CE block diagram

TBD

1.3 Frames

Data are exchanged between CE and plug-in as indivisible units of *Frames*. A frame is simply a set of arbitrary data characterized by content, size and type. Frames sent by the CE to a plug-in for *transmission* are *outbound* frames. Frames *received* by a plug-in and sent to the CE are *inbound* frames. Outbound frames are described in Section 2.2 and inbound frames in Section 2.3.

1.3.1 Frame size

A frame's size is always specified in units of *bytes* (8-bits). Frames are allowed to vary in size from transfer to transfer and/or by specific plug-in. The minimum sized frame, whether inbound or outbound, is *one* (1) *byte* while the maximum sized frame is determined by the specific plug-in.

1.3.2 Frame header and payload

TBD

1.3.3 Frame type

A frame's *Type* is a *small*[1] enumeration assigned to a frame. The enumeration for an *outbound* frame would be assigned (typically) by plug-in specific software in the CE, while for an *inbound* frame that assignment is the responsibility of the plug-in. Enumerations classify a frame's contents and in turn that classification may be used to vector a frame to its appropriate content specific processing. For an inbound frame that processor is within the plug-in, while for an outbound frame that processor is (typically) plug-in specific software in the CE.

For example, assume a plug-in that requires configuration information from the CE before it may transfer frames. To satisfy that requirement the appropriate configuration information could be built and sent to the plug-in packaged as an outbound frame. The plug-in would differentiate configuration frames from transmission frames by assigning a different enumeration to each type. Or, as a further example, assume the same plug-in accumulates state information as it transfers frames, for example, the number of frames it has transmitted or received. The CE wishes to sample those counters. In such a case the CE builds and sends to the plug-in an outbound frame requesting the current value of that counter. The plug-in responds by sampling that counter, saving that sample within an inbound frame and sending that frame back to the CE. Plug-in specific software on the CE would differentiate that frame from frames received by the plug-in through different enumeration values.

1.4 Register Conventions

The firmware interface to the CE consists of *registers* mapped to the processor's GPO AXI bus (see Appendix B, "Register Map"). Registers on this bus are all *thirty-two* (32) bits wide. Registers are accessed through either `load` or `store` instructions whose target is the address of the specified register. `Load` instructions are used to *read* a register and `Store` instructions are used to *write* a register. Although all of `I/O` space is nominally marked read/writable, side effects may occur to a accessed register which is naturally read or write *only*. Specifically for write-only registers `load` instructions will return an indeterminate value and for read-only registers `store` instructions will fail and the value to be written is silently discarded.

1. Contained in a 4-bit field.

1.4.1 FIFO access

Many registers are in fact mapped to FIFOS. Side effects to these types of registers may occur depending on access type and whether the FIFO's *read* or *write* port is mapped to the bus. Specifically, for a register mapped to a FIFO's *write* port:

— The FIFO is treated as a *Write-Only* register.

— `Store` instructions to a *non-empty* FIFO insert one, 32-bit entry onto the FIFO.

— `Store` instructions to a *full* FIFO fail. The state of the FIFO is unchanged and the value to be written is silently discarded.

— `Load` instructions return an indeterminate value. The state of the FIFO is unchanged.

And for a register mapped to a FIFO's *read* port:

— The FIFO is treated as a *Read-Only* register.

— `Load` instructions to a *non-empty* FIFO remove one entry from the FIFO and return that entry as a single, 32-bit value. The *low-order* bit of the returned value is always *zero* (0).

— `Load` instructions to a *empty* FIFO return a single, 32-bit value. The *low-order* bit of the returned value is always *one* (1). The remaining high-order 31 bits are unspecified, but should be *zero* (0). The state of the FIFO is unchanged.

— `Store` instructions fail. The state of the FIFO is unchanged and the value written is silently discarded.

1.4.2 Fields

Many, if not most of the registers described in this document 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 (1) 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 *blued 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 always 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 computed by shifting the field's offset *left* by 16 bits (decimal). An **Enable** field is illustrated *lightly* blued-out. It satisfies the following conventions:

— may only be *set*, clearing the field is ignored

— reads 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* blued-out with their value. R*ead-Only* fields will:

— ignore writes

— reset to *zero*, unless otherwise documented

1.5 Plugs and Sockets

The RCE specifies a common framework for the exchange of information between application specific logic and the CE. That framework is based on a *Plug* and *Socket* model. In that model the *Plug* is a set of common *Core* IP provided by the framework which an application uses to *wrap* their specific logic. Any logic so wrapped is called *Plug-Specific-Logic* (PSL) while the combination of plug wrapper and PSL is called a *Plug-in*.

In turn, the CE contains a set of predefined *Sockets*. A socket has two functions, it:

— Provides a site to connect a plug-in to the CE

— Defines an I/O model for software resident on the CE's processor to send and receive arbitrary data from a connected plug-in.

Once wrapped, all PSLs, independent of their logic, appear and function the same to the CE. That is, they all share a common interface which allows their logic to "plug in" to any one of the CE's four available *sockets*. Once plugged in, that logic is able to transparently exchange information between it and the CE.

The relationship between the CE with its sockets and the plug-in with its plug and application specific logic is illustrated in Figure 3:

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

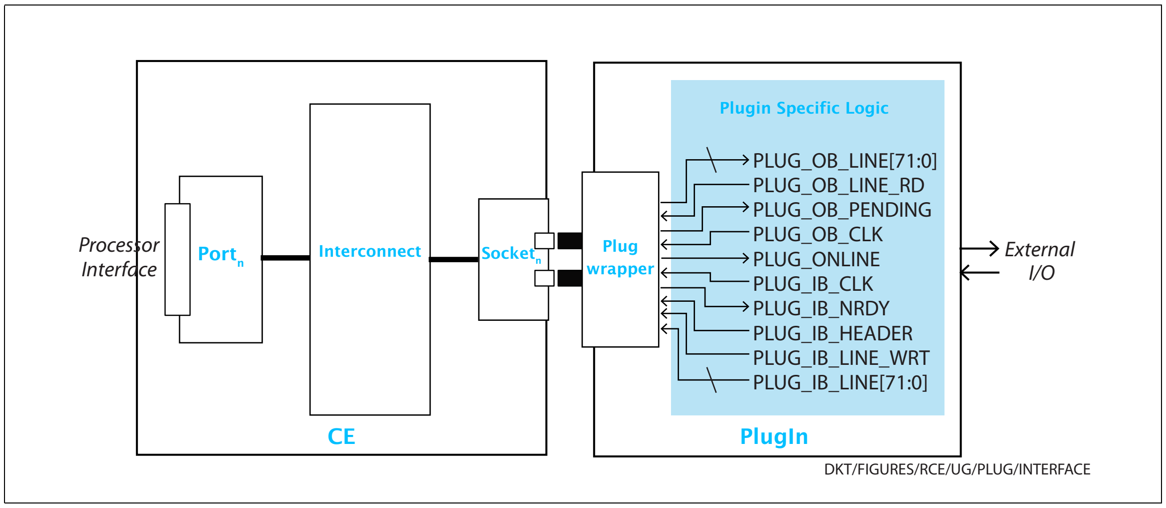

Figure 3 Block diagram of the Plug wrapper

TBD

Chapter 2
The Plug/Socket Interface

2.1 Overview

The socket is the *firmware* interface responsible for mediating frame transfer between processor software and plug-in. Figure 4 below is a block diagram illustrating the relationship between processor code, socket and plug-in:

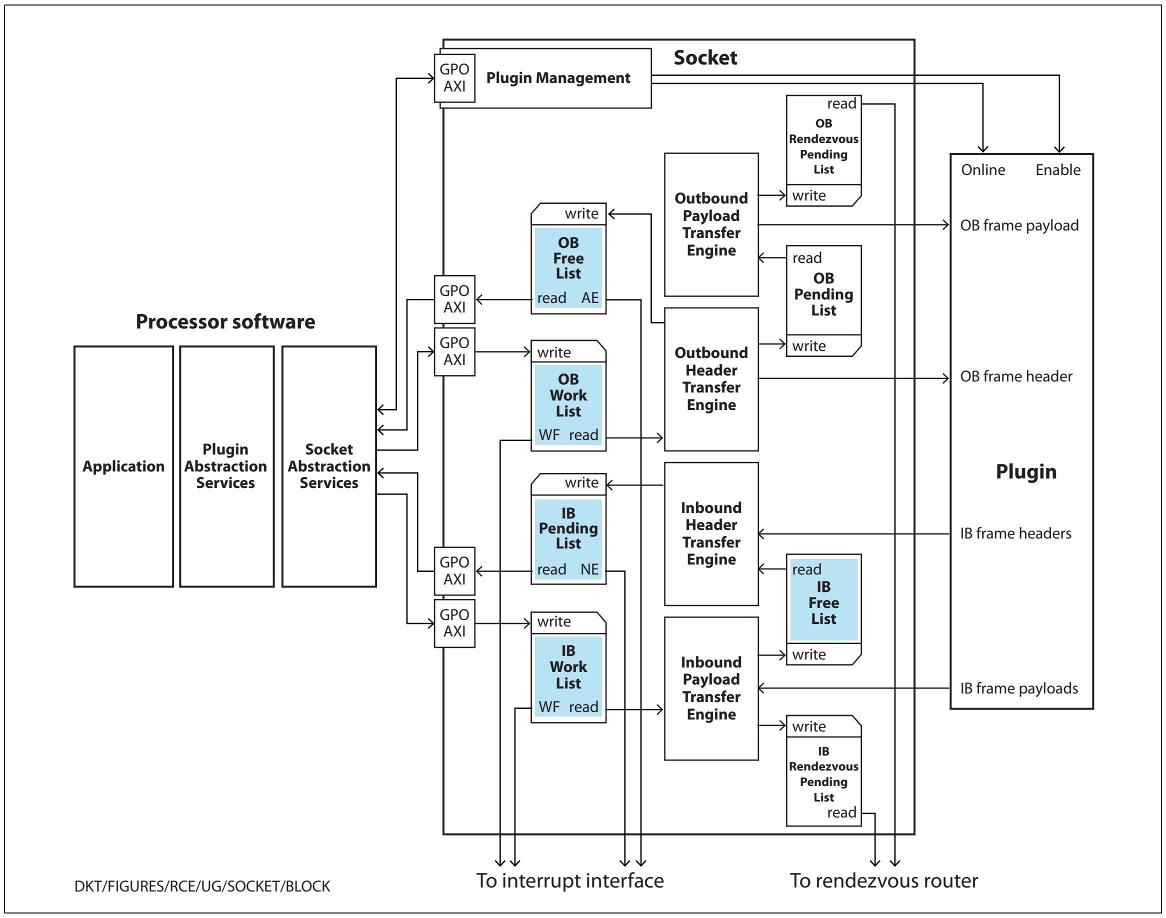

Figure 4 Block diagram of the Socket and its interfaces

Inside the socket are four *engines*, two to manage outbound frame transfers and two to manage inbound transfers. Each engine operates completely independently, allowing for concurrent inbound and outbound transfers. Each of the four engines is briefly summarized below.

2.1.1 Outbound Header Transfer Engine

In conjunction with the outbound payload transfer engine described in Section 2.1.2 below, the outbound *header* transfer engine moves outbound frames from processor to plug-in for transmission. Its principal responsibility is to transfer a frame *header* from descriptor to plug-in.

Both outbound header transfer engine and outbound payload transfer engine coordinate their operation though their socket's *outbound pending-list*. The pending-list is a *512* entry FIFO with its *write* port connected to this engine and its *read* port connected to the payload engine. Entries in this list correspond to the contents of an outbound descriptor's *header* (see Section 2.1.3).

A transfer is initiated by scheduling an *outbound instruction* to the engine. An instruction contains both *opcode* and *operand*. The instruction's opcode determines the specific operation required of the engine. The set of operations the engine supports is enumerated within Table 3 on page 32.

The operand describes the frame to transmit. That description is contained in a small, fixed size buffer called an *outbound frame descriptor* and is described in Section 2.1.3. Descriptors are allocated from the socket's *free-list*. The free-list is a *512* entry FIFO with its *write* port connected to the engine and its *read* port mapped to a GPO AXI register. Reading from this register allocates a descriptor.

Once allocated and initialized outbound instructions are queued to the engine's *work-list* for execution. The work-list is also a *512* entry FIFO. However, unlike the socket's free-list its *write* port is mapped to a GPO AXI register and its *read* port is connected to the engine. Writing this register *inserts* an entry on the FIFO and consequently queues an instruction. Note that as the work-list *is* a FIFO, instructions are always executed in the order they were queued.

The engine executes one instruction at a time. Execution is initiated when three conditions are met:

— The engine is idle.

— The socket's outbound pending-list is not *full*.

— The socket's outbound work-list is not *empty*.

When those conditions are met, the engine removes from its work-list an entry containing the instruction to execute. The instruction is both decoded and executed. Execution involves inserting the *header* of the instruction's descriptor onto the socket's pending-list and copying its *body* to the socket's plug-in. After execution the engine deallocates the descriptor by inserting it onto the socket's free-list and returns to idle.

Note, the free and work lists allow processor software to issue transfer requests concurrently with respect to their processing. The segregation of header and payload provides separation of protocol processing from its content without the need to partition these functions into either separate tasks or processes.

The outbound interface is described in further detail in Section 2.2.

2.1.2 Outbound Payload Transfer Engine

In conjunction with the outbound header engine described in Section 2.1.1 above, the outbound *payload* transfer engine moves outbound frames from processor to plug-in for transmission. Its principal responsibility is to transfer a frame's *payload* from processor memory to plug-in.

Both outbound payload transfer engine and outbound header transfer engine coordinate their operation though their socket's *outbound pending-list*. The pending-list is a *512* entry FIFO with its *read* port connected to this engine and its *write* port connected to the header engine. Entries in this list correspond to the *contents* of an outbound descriptor's header (see Section 2.1.3).

The engine processes one header at a time. Processing is initiated when four conditions are met:

— The engine is idle.

— The socket's outbound pending-list is not *empty*.

— The socket's plug-in is not *full*.

— The socket's outbound rendezvous-pending-list is not *full*.

When those conditions are met, the engine removes an entry from the socket's pending-list. That entry contains the header contents to process. It specifies at a minimum, a reference to a buffer containing a payload to be moved to the plug-in for transmission. First the payload is processed. After payload processing completes, the engine will:

— Based on header, conditionally inserts a rendezvous message on the socket's outbound rendezvous-pending-list. This signals a rendezvous (see Section 3.1.1).

— Based on payload processing status, conditionally inserts a frame transfer-fault message on the socket's outbound rendezvous-pending-list. This signals a frame transfer-fault (see Section 3.1.2).

— Returns to idle.

2.1.3 Outbound Frame Descriptors

A frame's transmission state is maintained by the *outbound frame descriptor*. Its principal function is to specify an outbound frame's header and payload[1]. Descriptors are fixed size buffers containing an integral number of *quadwords*[2]. They must also be quadword aligned and located within the processor's *256* KBytes of *On-Chip-Memory* (OCM). The socket interface communicates descriptors through its interface by *reference*. A reference is expressed as a *byte* offset from the starting location of the OCM. Note that location and quadword alignment dictate that a reference is an *18-bit* quantity whose low-order *four* (4) bits are always *zero* (0).

Each socket manages an ensemble of descriptors whose members are all the same size. However for that ensemble, while their size is fixed, their number varies as a function of socket as determined by software at processor boot time. Descriptor management is orchestrated through a *free-list* held by the socket. Software *allocates* from this free-list and the socket's transfer engine *returns* descriptors to this free-list. See Section 2.5.1 for a discussion of that free-list.

Descriptors contain a quadword *header* followed by a fixed size *body*. Body size is equal to the MOH (*Maximum-Outbound-Header*) of a socket's plug-in (see Section 1.3.2). A frame's header is always contained by *value* in a descriptor's body and its payload, if present by *reference* in its

1. When present.
2. Sixteen (16) *bytes* or 128 *bits*

header. Figure 5 on page 27 illustrates a hypothetical descriptor as first allocated and then subsequently initialized and deallocated:

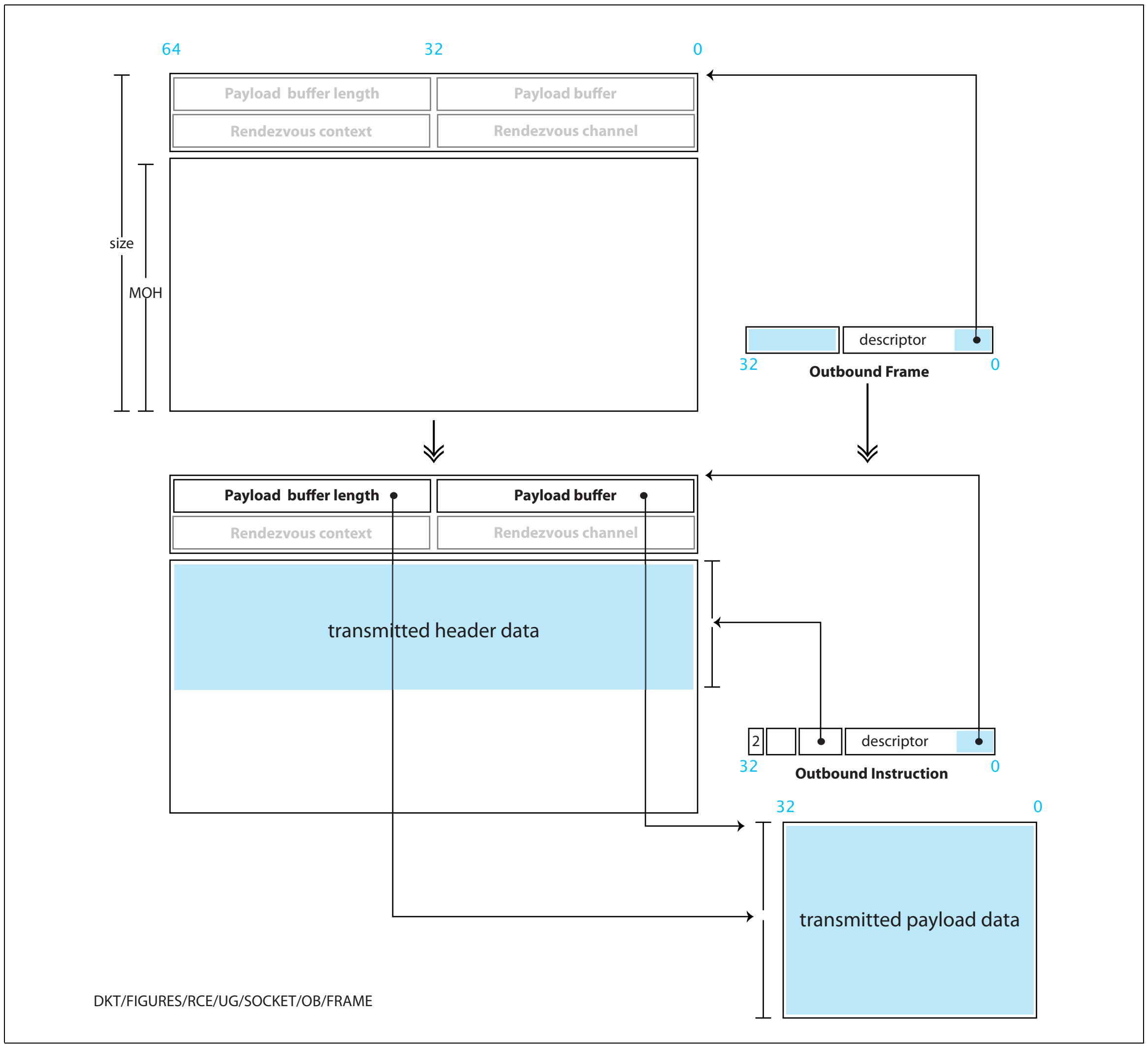

Figure 5 Outbound frame descriptors.

The figure shows that descriptor size is the sum of header and body and its body size is equal to a plug-in's MOH. Each allocated descriptor corresponds to a frame to be transmitted. After allocation the descriptor is initialized. The header data for a frame is copied to the descriptor's body and its header contains the reference to its payload. Once initialized, the descriptor is used to build and queue an instruction to the transfer engine for execution.

2.1.4 Inbound Header Transfer Engine

In conjunction with the inbound payload engine described in Section 2.1.5 below, the inbound *header* transfer engine transfers received inbound frames from plug-in to processor. Its principal responsibility is to transfer a frame's *header* from plug-in to descriptor.

Both inbound header transfer engine and inbound payload transfer engine coordinate their operation though their socket's *inbound pending-list*. The free-list is a *512* entry FIFO with its *read* port connected to this engine and its *write* port connected to the payload engine. Entries in this list are a *reference* to an inbound frame descriptor. The descriptor is a small, fixed size buffer described in Section 2.1.6. Removing from this list *allocates* a descriptor and inserting onto this list *frees* a descriptor.

The engine processes frames *one* at a time. Processing is initiated when four conditions are met:

— The engine is idle.

— The socket's free-list is not *empty*.

— The engine's pending-list is not *full* (see below).

— The socket's plug-in has at least one frame to transfer.

When those conditions are met the engine allocates a descriptor from the socket's free-list. Once allocated, the engine moves the frame's header from plug-in to the descriptor's body. and concludes by inserting a reference to the descriptor on the socket's *pending-list*. The pending-list is also a *512* entry FIFO however, its *write* port is connected to the engine and its *read* port is mapped to a GPO AXI register. Note that as the pending-list *is* a FIFO, descriptors are always received in the order they arrive at the plug-in.

To initiate processing of an inbound frame processor software reads the socket's pending-list register. Reading this register removes a descriptor reference from the socket's pending-list. Note that the pending-list FIFO has its *Not-Empty* flag connected to a processor interrupt.

The inbound interface is described in further detail in Section 2.3.

2.1.5 Inbound Payload Transfer Engine

In conjunction with the inbound header engine described in Section 2.1.4 above, the inbound *payload* transfer engine transfers received inbound frames from plug-in to processor. Its principal responsibility is to move a frame's *payload*.

Both inbound payload transfer engine and inbound header transfer engine coordinate their operation though their socket's *inbound free-list*. The free-list is a *512* entry FIFO with its *write* port connected to this engine and its *read* port connected to the header engine. Entries in this list are a *reference* to an inbound frame descriptor. The descriptor is a small, fixed size buffer described in Section 2.1.6. Removing from this list *allocates* a descriptor and inserting onto this list *frees* a descriptor.

A transfer is initiated by processor software by scheduling an *inbound instruction* to the engine. An instruction contains both *opcode* and *operand*. Its opcode specifies the operation required of the engine. The set of operations supported by an inbound engine is enumerated in Table 4 on page 39.

The operand describes where to place in memory a received payload. That description is contained in a small, fixed size structure called an *inbound frame descriptor* that is described in Section 2.1.6. Descriptors were returned by reading from the inbound pending-list (see Section 2.1.4).

Once allocated and initialized work instructions are queued to the engine's *work-list* for execution. The work-list is also a *512* entry FIFO. However, unlike the socket's free-list its *write* port is mapped to a GPO AXI register and its *read* port is connected to the engine. Writing this register *inserts* an entry on the FIFO and consequently schedules an instruction. Note that as the work-list *is* a FIFO, instructions are always executed in the order they were queued.

The engine executes one instructions at a time. Execution is initiated when four conditions are met:

— The engine is idle.

— The socket's inbound free-list is not *full*.

— The socket's inbound rendezvous-pending-list is not *full*.

— The socket's inbound work-list is not *empty*.

When those conditions are met, the engine removes from its work-list an entry containing the instruction to execute. The instruction is decoded and executed. Execution first involves, conditionally processing the plug-in's payload by either transfer to memory or discard. After payload processing the engine:

— Based on instruction, conditionally inserts a rendezvous message on the socket's inbound rendezvous-pending-list. This signals a rendezvous (see Section 3.1.1).

— Based on payload processing status, conditionally inserts a frame transfer-fault message on the socket's inbound rendezvous-pending-list. This signals a frame transfer-fault (see Section 3.1.2).

— Based on instruction, inserts the descriptor it on the socket's inbound free-list. This deallocates the descriptor.

— Returns to idle.

Note, the pending and work lists allow processor software to build transfer requests concurrently with respect to their processing. The segregation of header and payload provides separation of protocol processing from its content without the need to partition these functions into either separate tasks or processes.

The inbound interface is described in further detail in Section 2.3.

2.1.6 Inbound Frame Descriptors

A frame's reception state is maintained by the *inbound frame descriptor*. Its principal function is to specify an inbound frame's header and payload[1]. Descriptors are fixed size buffers containing an integral number of *quadwords*[2]. They must also be quadword aligned and located within the processor's *256* KBytes of *On-Chip-Memory* (OCM). The socket interface communicates descriptors through its interface by *reference*. A reference is expressed as a *byte* offset from the starting location of the OCM. Note that location and quadword alignment dictate that a reference is an *18-bit* quantity whose low-order *four* (4) bits are always *zero* (0).

Each socket manages an ensemble of descriptors whose members are all the same size. However for that ensemble, while their size is fixed, their number varies as a function of socket and is determined by software at processor boot time. Descriptor management is orchestrated through a *free-list* held by the socket. Software *allocates* from this free-list and the socket's transfer engine *returns* descriptors to this free-list. See Section 2.5.1 for a discussion of that free-list.

Descriptors contain a quadword *header* followed by a fixed size *body*. Body size is equal to the MIH (*Maximum-Inbound-Header*) of a socket's plug-in (see Section 1.3.2). A frame's header is always contained by *value* in a descriptor's body and its payload, if present by *reference* in its header. Figure 6 on page 31 illustrates a hypothetical descriptor as retrieved from the socket's pending-list and then subsequently initialized and deallocated:

1. When present.
2. Sixteen (16) *bytes* or 128 *bits*

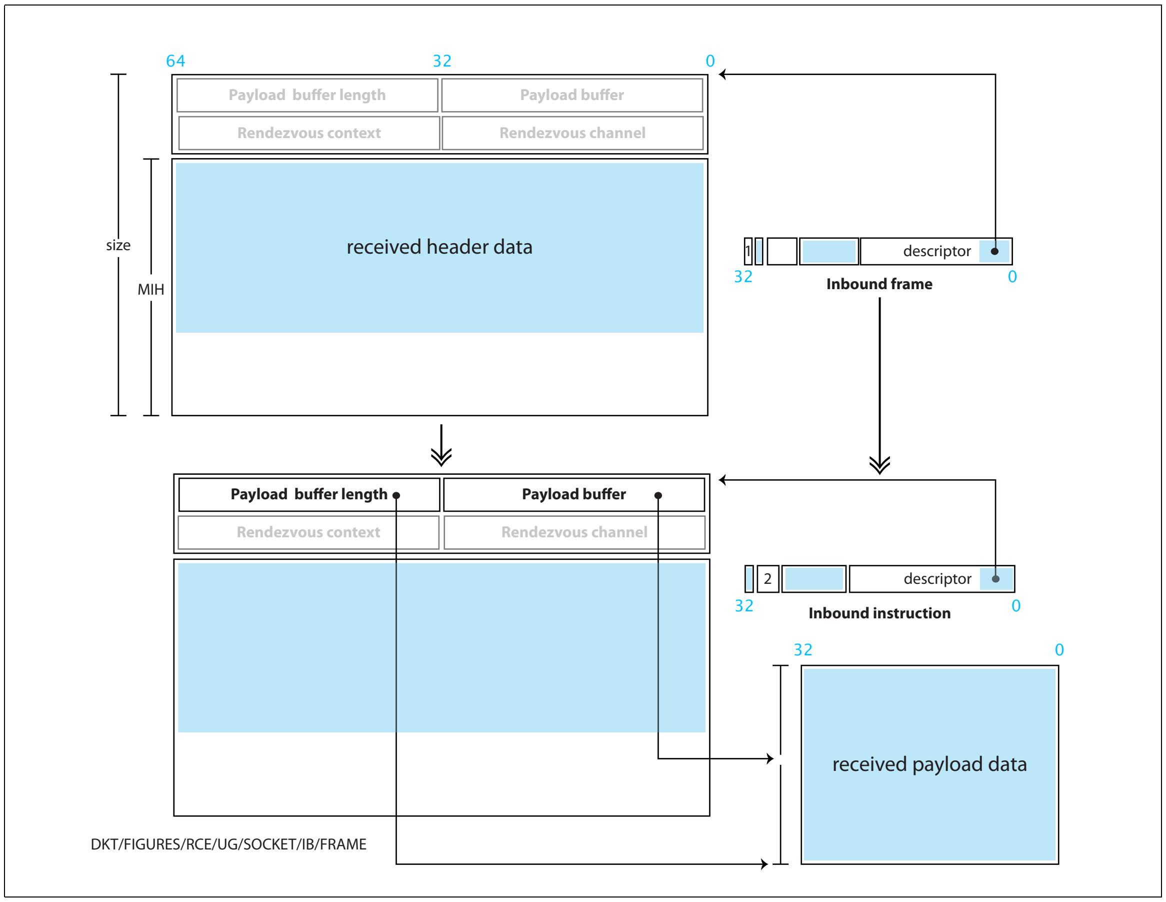

Figure 6 Inbound frame descriptors.

The figure shows that descriptor size is the sum of its header and body and its body size is equal to a plug-in's MIH. Each descriptor returned from the pending-list corresponds to one received inbound frame and its header is contained in the descriptor's body.

In this example the received frame contains a payload. After reception the header is decoded to resolve the placement of the frame's payload and once resolved a reference to the receive buffer is copied to the descriptor's header. The initialized descriptor is used to build the instruction to queue to the transfer engine for execution.

2.2 Outbound Interface

As described in Section 2.1.1 a socket's outbound interface employs its outbound *work* and *free* list FIFOs. Consequently, its corresponding processor interface consists simply of two GPO AXI registers (see Appendix B, "Register Map"). The *read* port of the *free-list* FIFO is mapped to a *read-only* register while the *write* port of the *work-list* FIFO is mapped to a *write-only* register.

Therefore, for each socket the interface provides processor software with two operations: *Removal* from its free-list and *insertion* onto its work-list. Section 1.4.1 describes the access policy for FIFOs mapped to registers.

An entry inserted on a socket's work-list is an *outbound instruction*. An instruction serves two purposes: First, it specifies a potential transmit function. That function is encoded as an *opcode* and *operand*. The opcode specifies the type of transfer required of the engine and the operand is a reference to an *outbound descriptor* (see Section 2.1.3) containing the frame to transfer. Second, the instruction directs the engine, once transfer is complete to return its corresponding descriptor back to the transfer engine's free-list. Table 3 on page 32 enumerates the allowed transfer functions along with their associated encoding, while Section 2.5.2 defines the structure of an instruction.

TABLE 3 Outbound opcodes

Opcode	Description and transfer engine action	See:
`0`	The instruction's descriptor defines *neither* a frame nor rendezvous. The entire contents of the instruction's descriptor are ignored and the engine returns the descriptor to its free-list.	Section 2.2.1.1
`1`	The instruction's descriptor specifies a frame containing *only* a header. The engine transfers the frame to its corresponding plug-in for transmission. On completion the engine returns the descriptor to its free-list.	Section 2.2.1.2
`2`	The instruction's descriptor specifies a frame containing *both* header and payload. The engine gathers the frame and transfers it to its corresponding plug-in for transmission. On completion the engine returns the descriptor to its free-list.	Section 2.2.1.3
`3`	The instruction's descriptor specifies a frame containing *both* header and payload as well as rendezvous. The engine gathers the frame and transfers it to its corresponding plug-in for transmission. On completion, it inserts the specified rendezvous context onto the specified rendezvous channel and returns the descriptor to its outbound free-list.	Section 2.2.1.4

Figure 7 on page 33 expresses the relationship between instruction and descriptor:

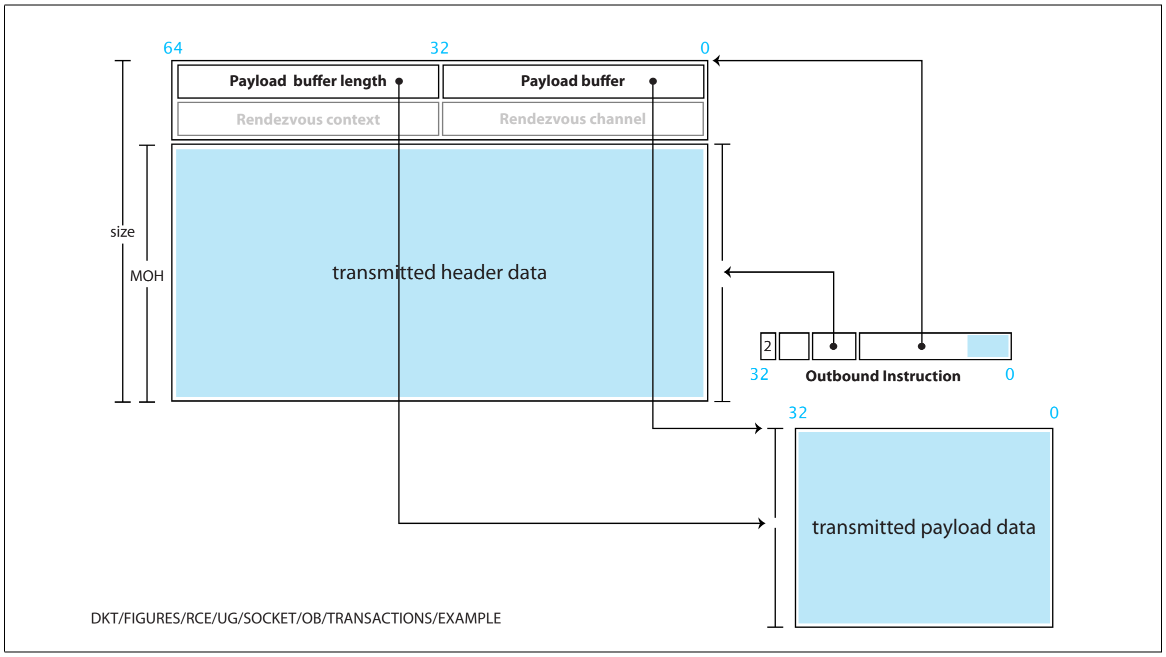

Figure 7 Outbound descriptors and instructions.

This example illustrates an instruction to transmit a frame containing *both* header and payload. This corresponds to an opcode *two* (2) instruction with the instructions's descriptor *containing* the frame's header in its body and *pointing* to its payload in its header. Note, for this example, the length of the frame's header is equal to the plug-in's MOH.

A socket's outbound free-list is initially populated by posting opcode *zero* (0) instructions to its *work-list*. Each instruction adds one descriptor to the socket's free-list. Once populated, frame transmission is a straightforward three step process:

— *Allocate* a descriptor by removing from the socket's outbound free-list. The returned value is a reference to a descriptor used to specify the frame transmitted. The structure of a value returned from a free-list is described in Section 2.5.1.

— Convert the descriptor reference to an address. Initialize that descriptor with the frame to be transmitted. Note that a descriptor always contains, if present, the frame's *header* by *value* and if present, its *payload* by *reference*. Section 2.2.1 specifies proper construction of a descriptor's header for each type of instruction.

— Build an outbound instruction using the initialized descriptor. Post the instruction to the transfer engine by inserting it on the engine's outbound work-list (see Section 2.5.2).

Note that although the socket's outbound transfer engine processes only one outbound instruction at a time and always in the order requested this does not imply the processor is necessarily blocked waiting on the transfer engine to complete. Instead, the socket's pending FIFO allows for an asynchronous I/O model in which instructions are *queued* while the engine is busy processing a transaction. The maximum number of outstanding transactions permitted determined by the initial population of the socket's free-list.

2.2.1 Transactions

2.2.1.1 Free descriptor

Figure 8 illustrates an instruction which does *not* transmit a frame and *only* frees a previously allocated descriptor buffer. The instruction contains a reference to a valid *outbound* descriptor (see Section 2.1.3) and an `opcode` (see Table 3 on page 32) with a value of *zero* (0). The descriptor reference was acquired from a socket's free-list as described in Section 2.5.1.

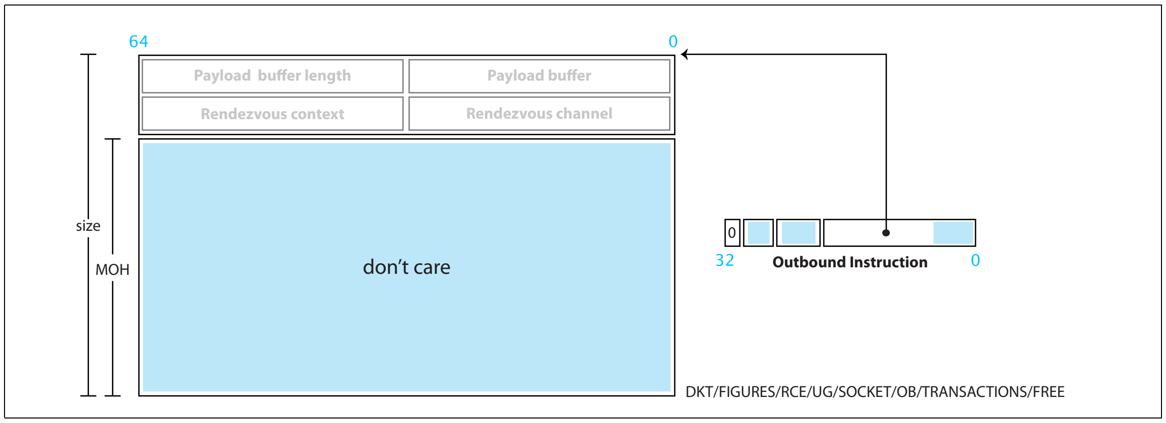

Figure 8 Free outbound descriptor (opcode = 0)

Note that for this type of instruction both **Length** and **Frame Type** fields are ignored by the transfer engine, but should be *zero* (0). Note also that the contents of the descriptor's header and body are ignored by the transfer engine. This instruction serves two purposes:

— To initially populate a socket's outbound free-list after a reset (see Section 2.2).

— To handle an error in descriptor construction which must abort transmission.

2.2.1.2 Transmit frame with only header

Figure 9 illustrates an instruction which transmits an outbound frame. However that frame is wholly contained in the instruction's descriptor body. The instruction contains a reference to a valid *outbound* descriptor (see Section 2.1.3) and an `opcode` (see Table 3 on page 32) with a value of *one* (1). The descriptor reference was acquired from a socket's free-list as described in Section 2.5.1.

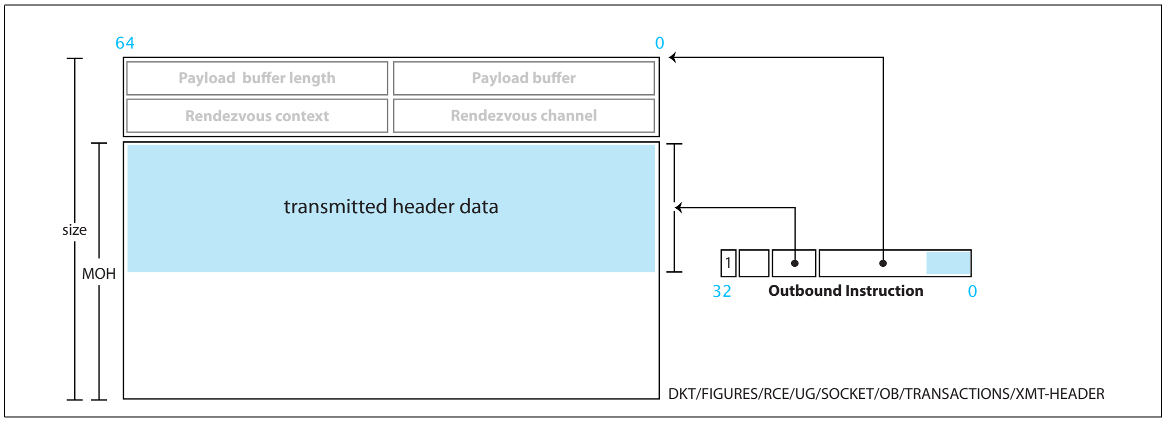

Figure 9 Transmit outbound frame with only header (opcode = 1)

For this type of instruction both **Length** and **Frame Type** fields must contain valid values. The **Length** field must:

— be expressed in units of *quadwords.*

— be equal to the size of the transmitted frame's header.

— not be *zero* (0).

— not exceed the plug-in's MOH (see Section 2.2).

If any of these constraints are violated the socket's resulting behaviour is undefined.

The behaviour of a plug-in processing a frame whose type it does not support is unspecified.

2.2.1.3 Transmit frame

Figure 10 illustrates an instruction which transmits an outbound frame. That frame contains both header and payload. The instruction contains a reference to a valid *outbound* descriptor (see Section 2.1.3) and an `opcode` (see Table 3 on page 32) with a value of *two* (2). The descriptor reference was acquired from a socket's free-list as described in Section 2.5.1.

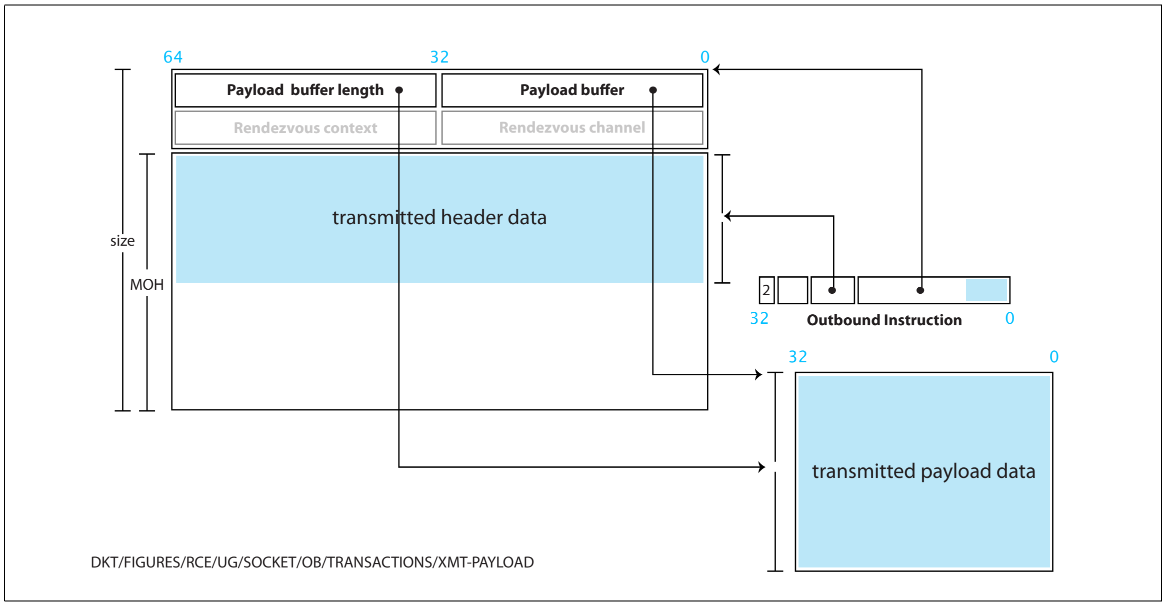

Figure 10 Transmit outbound frame (opcode = 2)

For this type of instruction both **Length** and **Frame Type** fields must contain valid values. The **Length** field must:

— be expressed in units of *quadwords.*

— be equal to the size of the transmitted frame's header.

— not exceed the plug-in's MOH (see Section 2.2).

If any of these constraints are violated the socket's resulting behaviour is *undefined*.

The behaviour of a plug-in processing a frame whose type it does not support is *unspecified*.

As is the case for the transmission of any outbound frame, header data is contained within the body of the instruction's descriptor. However, unlike an `opcode 1` instruction (see Section 2.2.1.2), *zero* is a valid length. If its **Length** field has a *zero* (0) value the transmitted frame contains *only* payload.

The descriptor's header contains a valid *payload reference*. That reference occupies the header's first quadword and contains two 32-bit (*word*) fields whose interpretation is as follows:

Payload buffer: This field contains a *physical* address pointing to a buffer containing the frame's payload. No restrictions are placed on its alignment. The transfer engine assumes the buffer has either previously been flushed or is located in uncached memory. Transfer behaviour is undefined if neither constraint is met. A buffer's *length* is determined by the field below.

Payload length: This field contains the length of the payload whose data are pointed to by the field above. Lengths are expressed in units of *bytes* (8-bits) and range from a value of *zero* (0) to the MOF of the plug-in associated with the socket (see Section 1.3.2). A value of zero is allowed, but will *not* incur a transfer. The behaviour of a transfer which exceeds MOF is undefined, as is the behaviour of a transfer exceeding the processor's physical address space.

The contents of the rendezvous reference in the descriptor's header are ignored by the transfer engine and need not be initialized to any known value.

2.2.1.4 Transmit and rendezvous frame

Figure 11 illustrates an instruction which transmits an outbound frame. That frame contains both header and payload. However, once its transfer is complete the transfer engine arranges and starts a rendezvous (see Chapter 3). The instruction contains a reference to a valid *outbound* descriptor (see Section 2.1.3) and an `opcode` (see Table 3 on page 32) with a value of *three* (3).

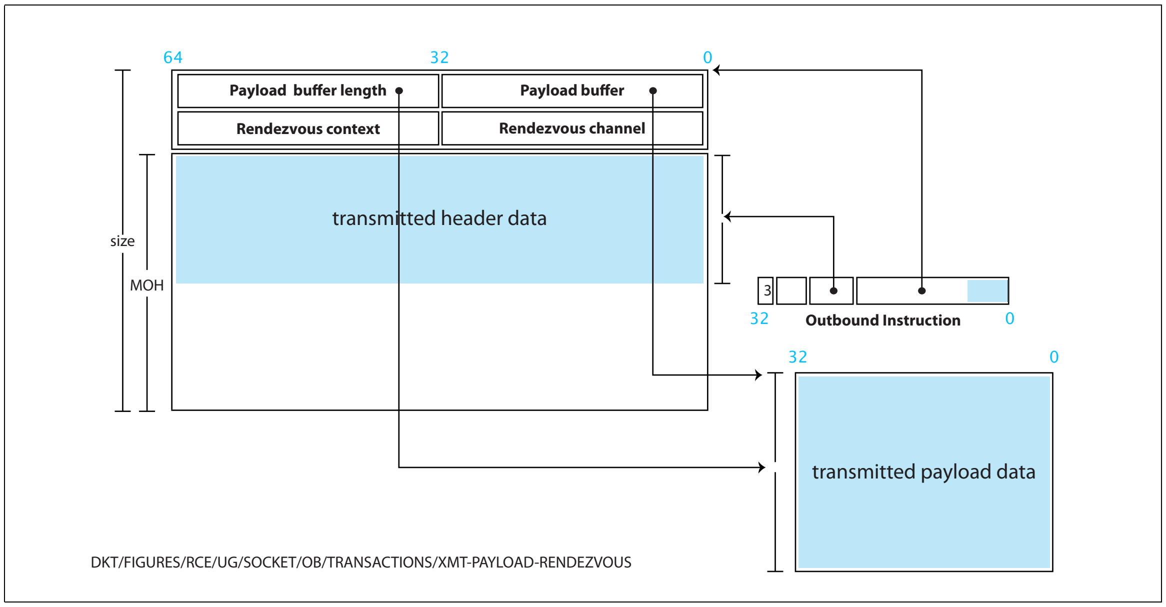

Figure 11 Transmit and rendezvous outbound frame (opcode = 3)

For this type of instruction both **Length** and **Frame Type** fields must contain valid values. The **Length** field must:

— be expressed in units of *quadwords.*

— be equal to the size of the transmitted frame's header.

— not exceed the plug-in's MOH (see Section 2.2).

If any of these constraints are violated the socket's resulting behaviour is *undefined*.

The behaviour of a plug-in processing a frame whose type it does not support is *unspecified*.

As is the case for the transmission of any outbound frame, header data is contained within the body of the instruction's descriptor. However, unlike an `opcode 1` instruction (see Section 2.2.1.2), *zero* is a valid length. If its **Length** field has a *zero* (0) value the transmitted frame contains *only* payload.

The descriptor's header contains a valid *payload reference*. That reference occupies the header's first quadword and contains two 32-bit (*word*) fields whose interpretation is as follows:

Payload buffer: This field contains a *physical* address pointing to a buffer containing the frame's payload. No restrictions are placed on its alignment. The transfer engine assumes the buffer has either previously been flushed or is located in uncached memory. Transfer behaviour is undefined if neither constraint is met. A buffer's *length* is determined by the field below.

Payload length: This field contains the length of the payload whose data are pointed to by the field above. Lengths are expressed in units of *bytes* (8-bits) and range from a value of *zero* (0) to the MOF of the plug-in associated with the socket (see Section 1.3.2). A value of zero is allowed, but will *not* incur a transfer. The behaviour of a transfer which exceeds MOF is undefined, as is the behaviour of a transfer exceeding the processor's physical address space.

The descriptor's header also contains a valid *rendezvous reference*. A rendezvous is described in Chapter 3. A rendezvous reference occupies the header's *second* quadword and contains two 32-bit (*word*) fields. Section 3.1.1 describes these two fields as well as their interpretation.

2.3 Inbound interface

As described in Section 2.1 a socket's inbound interface employs its inbound *pending* and *free* list FIFOs. Consequently its corresponding processor interface consists simply of two GPO AXI registers (see Appendix B, "Register Map"). The *read* port of a *pending-list* FIFO is mapped to a *read-only* register while the *write* port of a *free-list* FIFO is mapped to a *write-only* register. Therefore, for each socket the interface provides processor software with two operations: *Removal* from its pending-list and *insertion* onto its work-list. Section 1.4.1 describes the access policy for FIFOs mapped to registers.

For each frame received from a socket's plug-in its corresponding inbound *header* transfer engine allocates a descriptor from the socket's free-list, moves the frame's header to the body of the descriptor and inserts the constructed descriptor onto the engine's pending-list.

Frames are received by removal from the engines's pending-list. Each removed entry corresponds to a single inbound frame. Once removed, the frame's payload must be disposed by the engine and the frame's descriptor returned to the socket's free-list. To dispose a payload the engine can be instructed to either move it or discard it. If moved, its destination in memory is specified as a parameter to the engine.

Payload and descriptor disposal are bound together in a single operation by inserting an *inbound instruction* onto the engine's work-list. That instruction includes not only the descriptor to deallocate but also an `opcode` specifying the engine's disposal instructions. Table 4 on page 39 enumerates the allowed opcodes while Section 2.5.3 defines the structure of an instruction.

TABLE 4 Inbound Opcodes

Opcode	Description and transfer engine action	See:
`0`	The descriptor contains *neither* payload nor rendezvous information. Conditionally return the descriptor to its socket's free-list.	Section 2.3.1.1
`1`	The descriptor specifies *neither* receive buffer nor rendezvous. The engine discards (flushes) the payload of the socket's oldest pending inbound frame. Conditionally, return the descriptor to its socket's free-list.	Section 2.3.1.2
`2`	The descriptor specifies a receive buffer. The engine moves the payload of its oldest pending inbound frame to the specified receive buffer. If that payload is larger than the receive buffer the payload is silently truncated. Conditionally, return the descriptor to its socket's free-list.	Section 2.3.1.3
`3`	*Reserved*. The descriptor contains *neither* payload nor rendezvous information. Conditionally the engine returns the descriptor to the socket's free-list.	N/A
`4`	The descriptor does *not* specify a receive buffer, but does specify a rendezvous. On completion, insert the rendezvous context onto the rendezvous channel. Conditionally return the descriptor to its socket's free-list.	Section 2.3.1.5
`5`	*Reserved*. The descriptor contains *neither* payload nor rendezvous information. Conditionally return the descriptor to its socket's free-list.	N/A
`6`	The descriptor specifies both a receive buffer and rendezvous. The engine moves the payload of its oldest pending inbound frame to the receive buffer. If the payload is larger than the receive buffer the payload is truncated. On completion, insert the rendezvous context onto the rendezvous channel. Conditionally return the descriptor to its socket's free-list.	Section 2.3.1.4
`7`	*Reserved*. The descriptor contains *neither* payload nor rendezvous information. Conditionally, return the descriptor to its socket's free-list.	N/A

Figure 12 on page 40 expresses the relationship between instruction and descriptor:

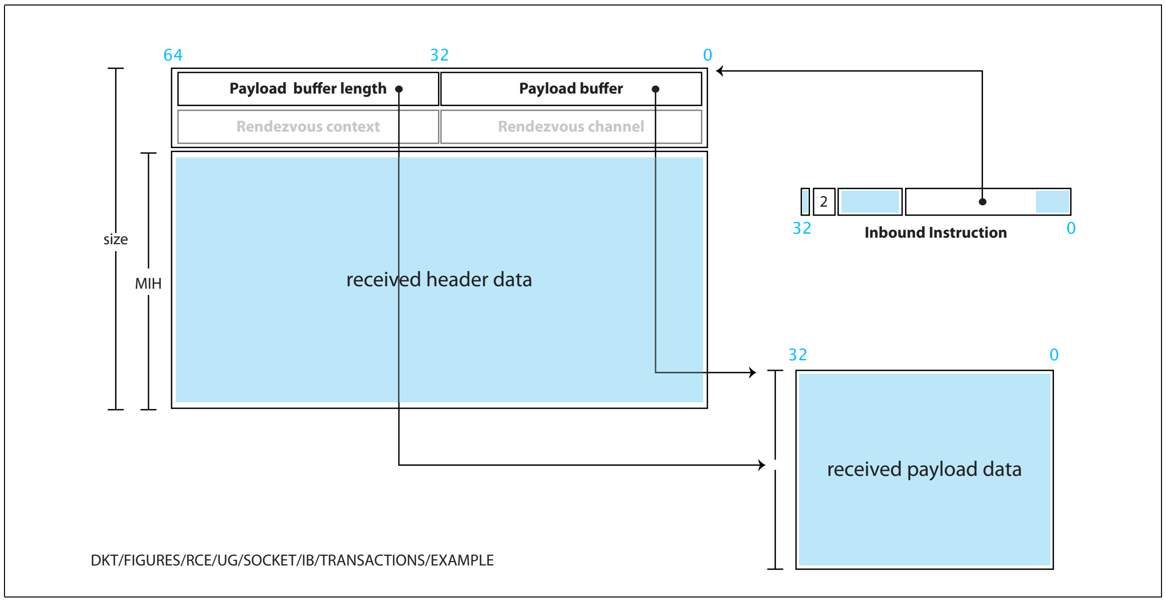

Figure 12 Inbound descriptors and instructions.

This example illustrates the reception of a frame containing *both* header and payload. For this example the length of the frame's header is equal to the plug-in's *Maximum-Inbound-Header* (MIH).

The inbound free-list is initially populated by posting instruction *zero* (0) entries to the socket's inbound *free-list*. Once its free-list is populated frame reception is a straightforward three step process:

— Wait for an inbound frame by removing an entry from a socket's inbound pending-list. The structure of an entry on this list is specified in Section . If the list is not empty the returned value contains a reference to an inbound frame descriptor. It also contains a flag indicating whether or not the frame has a payload. For this example, that flag is assumed *true*.

— Convert the descriptor reference to an address. The descriptor body contains the frame's header. Decode the header to determine where to place the frame's payload. Initialize the descriptor with a reference to a buffer to receive the payload.

— Construct an instruction using the initialized descriptor. Post the instruction to the socket's *payload* transfer engine by inserting the instruction on its inbound free-list (see Section 2.5.4).

Note that although the socket processes only a single inbound instruction at a time and always in the order received this does not imply the processor is necessarily blocked waiting on the transfer engine to complete. Instead, the socket's pending FIFO allows for an asynchronous I/O model in which transfer instructions are *queued* while the engine is busy processing a transaction. The maximum number of outstanding transactions permitted determined by the initial population of the socket's free-list.

2.3.1 Transactions

2.3.1.1 Free descriptor

Figure 13 illustrates an instruction which does *not* consume an inbound payload and *conditionally* frees a previously allocated inbound descriptor. The instruction contains a reference to a valid *inbound* descriptor (see Section 2.1.6) and an `opcode` (see Table 4 on page 39) with a value of *zero* (0). The descriptor reference was acquired from a socket's pending-list as described in Section .

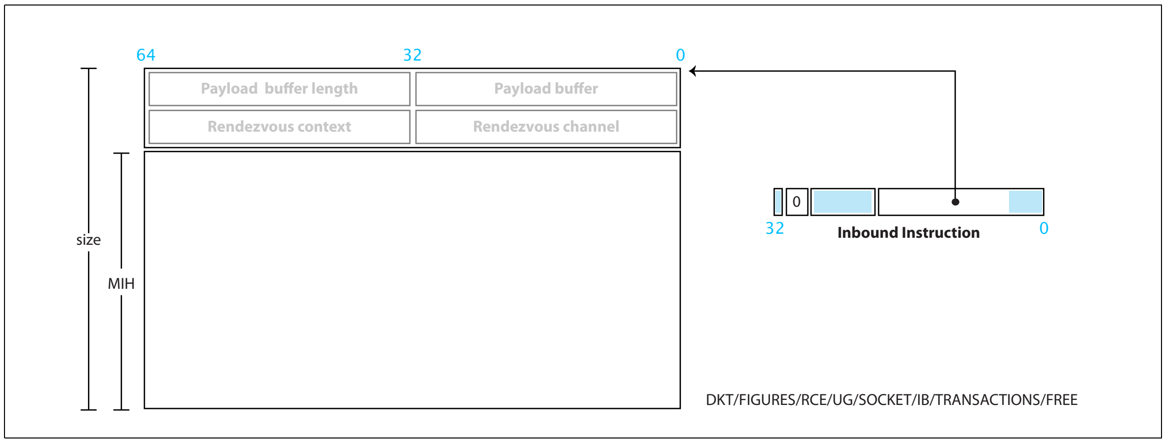

Figure 13 Free inbound descriptor (opcode = 0)

Note that the contents of the descriptor's header and body are ignored by the transfer engine. This instruction serves two purposes:

— To initially populate a socket's inbound free-list after a reset (see Section 2.3).

— To free a received frame which *only* contains a header.

2.3.1.2 Flush frame payload

Figure 14 illustrates an instruction which *discards* an inbound payload and *conditionally* frees a previously allocated inbound descriptor.The instruction contains a reference to a valid *inbound* descriptor (see Section 2.1.6) and an `opcode` (see Table 4 on page 39) with a value of *one* (1). The descriptor reference was acquired from a socket's pending-list as described in Section .

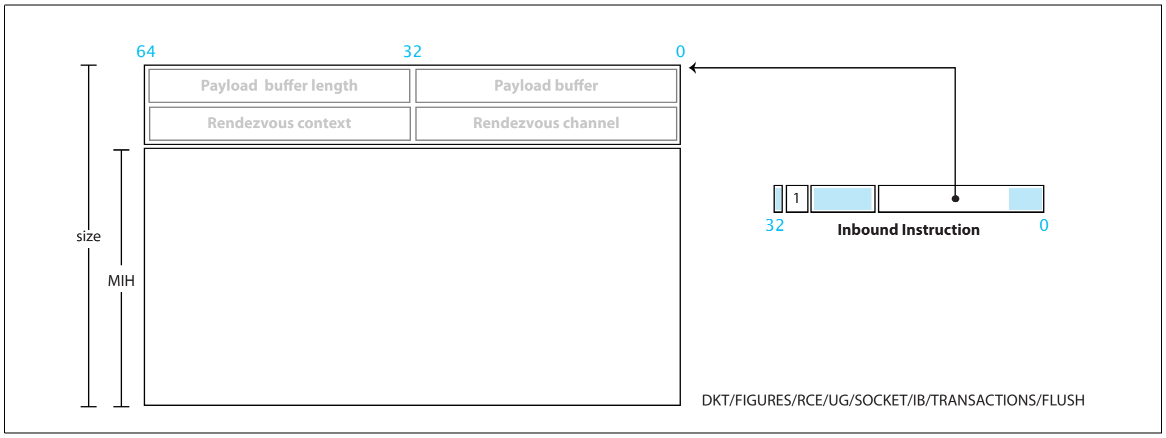

Figure 14 Flush inbound frame payload (opcode = 1)

Note that the contents of the descriptor's header and body are ignored by the transfer engine.

2.3.1.3 Receive frame payload

Figure 15 illustrates an instruction which receives an inbound frame's payload and *conditionally* frees a previously allocated inbound descriptor. The instruction contains a reference to a valid *inbound* descriptor (see Section 2.1.6) and an `opcode` (see Table 4 on page 39) with a value of *two* (2). The descriptor reference was acquired from a socket's pending-list as described in Section .

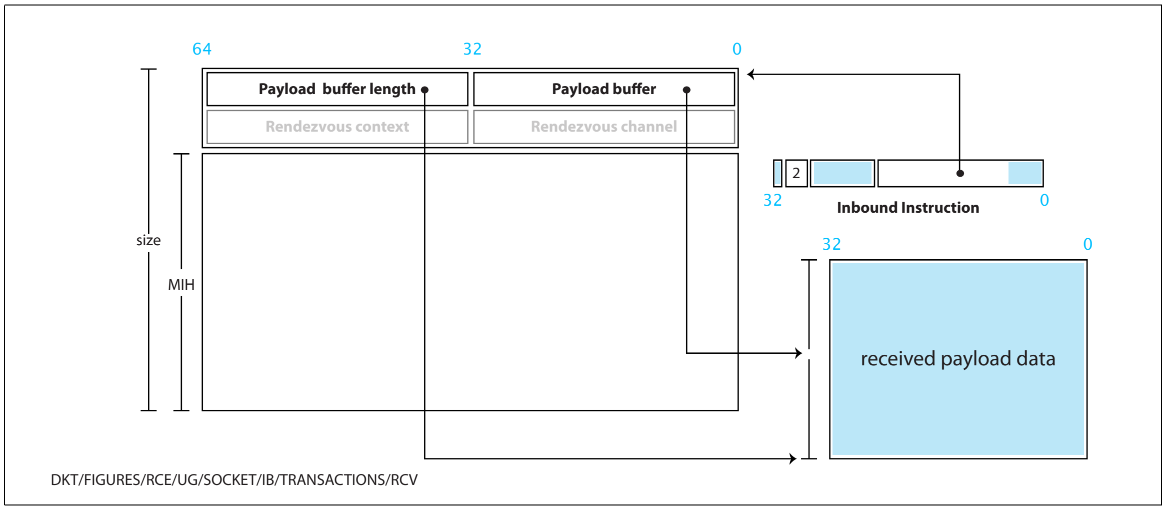

Figure 15 Receive inbound frame payload (opcode = 2)

The descriptor's header contains a valid *payload reference*. That reference occupies the header's first quadword and contains two 32-bit (*word*) fields whose interpretation is as follows:

Payload buffer: This field contains a *physical* address pointing to a receive buffer to contain the frame's payload. No restrictions are placed on its alignment. The transfer engine assumes the buffer has either previously been flushed or is located in uncached memory. Transfer behaviour is undefined if neither constraint is met. A buffer's *length* is determined by the field below.

Payload length: This field contains the length of the receive buffer described by the field above. Lengths are expressed in units of *bytes* (8-bits) and range from a value of *zero* (0) to the MIF of the plug-in associated with the socket (see Section 1.3.2). A value of zero is permitted, but will discard the entire payload. If such behaviour is required, see Section 2.3.1.2. The behaviour of a transfer which exceeds MIF is undefined, as is the behaviour of a transfer exceeding the processor's physical address space.

The contents of the rendezvous reference in the descriptor's header are ignored by the transfer engine and need not be initialized to any known value.

2.3.1.4 Receive payload and rendezvous frame

Figure 16 illustrates an instruction which receives an inbound frame's payload and *conditionally* frees a previously allocated inbound descriptor. However, additionally, once transfer is complete the instruction arranges and starts a rendezvous (see Chapter 3). The instruction contains a reference to a valid *inbound* descriptor (see Section 2.1.6) and an `opcode` (see Table 4 on page 39) with a value of *six* (6). The descriptor reference was acquired from a socket's pending-list as described in Section .

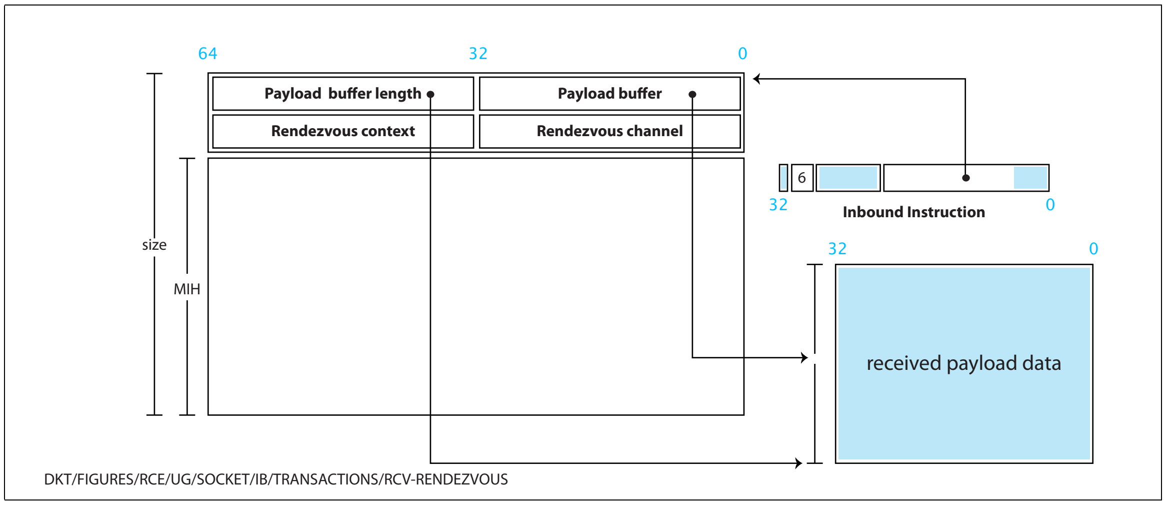

Figure 16 Receive inbound payload and rendezvous frame (opcode = 6)

The descriptor's header contains a valid *payload reference*. That reference occupies the header's first quadword and contains two 32-bit (*word*) fields whose interpretation is as follows:

Payload buffer: This field contains a *physical* address pointing to a receive buffer to contain the frame's payload. No restrictions are placed on its alignment. The transfer engine assumes the buffer has either previously been flushed or is located in uncached memory. Transfer behaviour is undefined if neither constraint is met. A buffer's *length* is determined by the field below.

Payload length: This field contains the length of the receive buffer described by the field above. Lengths are expressed in units of *bytes* (8-bits) and range from a value of *zero* (0) to the MIF of the plug-in associated with the socket (see Section 1.3.2). A value of zero is permitted, but will discard the entire payload. If such behaviour is required, see Section 2.3.1.2. The behaviour of a transfer which exceeds MIF is undefined, as is the behaviour of a transfer exceeding the processor's physical address space.

The descriptor's header also contains a valid *rendezvous reference*. A rendezvous is described in Chapter 3. A rendezvous reference occupies the header's *second* quadword and contains two 32-bit (*word*) fields. Section 3.1.1 describes these two fields as well as their interpretation.

2.3.1.5 Rendezvous frame

Figure 17 illustrates an instruction which arranges and begins a rendezvous (see Chapter 3) and *conditionally* frees a previously allocated inbound descriptor. The instruction contains a reference to a valid *inbound* descriptor (see Section 2.1.6) and an `opcode` (see Table 4 on page 39) with a value of *four* (4). The descriptor reference was acquired from a socket's pending-list as described in Section .

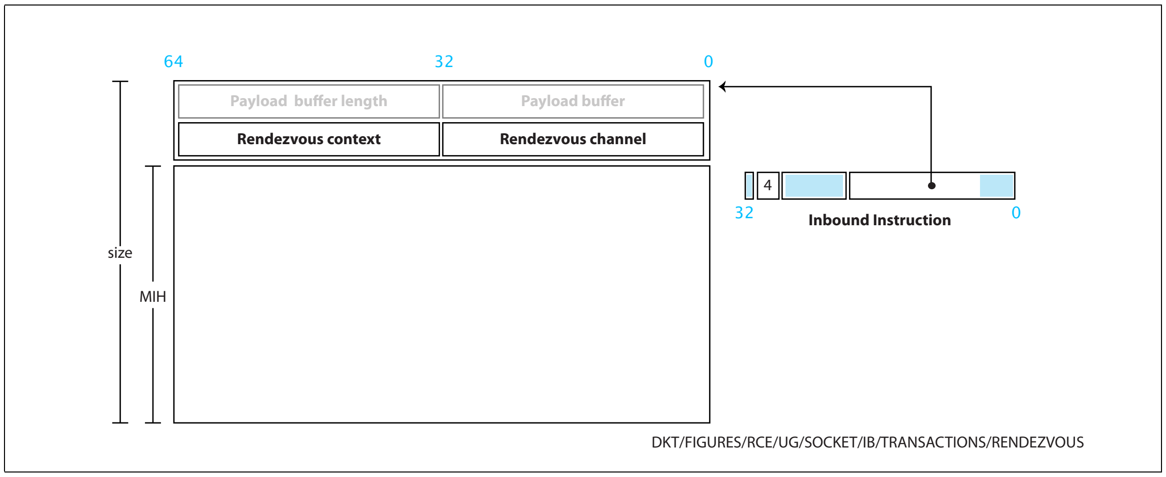

Figure 17 rendezvous descriptor (opcode = 4)

The contents of the payload reference in the descriptor's header are ignored by the transfer engine and need not be initialized to any known value. However, the descriptor's header does contains a valid *rendezvous reference*. A rendezvous is described in Chapter 3. A rendezvous reference occupies the header's *second* quadword and contains two 32-bit (*word*) fields. Section 3.1.1 describes these two fields as well as their interpretation.

2.4 Management

The socket interface provides the capability to *reset* a plug-in as well as conditionally mask *off* its inbound data. Those capabilities are expressed by modelling plug-in behaviour as an abstract *Finite State Machine* with *three* states. Those states are:

Online: This is a plug-in's nominal state. In this state a plug-in receives inbound data and transmits outbound data. The plug-in may *not* be reset from this state.

Offline: This is a plug-in's setup state. In this state a plug-in ignores inbound data, however continues to receive and process outbound frames. While in this state inbound frames *solicited* through an outbound frame may be posted to the socket. A plug-in may be reset *only* from this state. This state is used to resolve two race conditions: First, the race between the *setup* of a plug-in with its *use* and second, the race between a plug-in *reset* and its subsequent use.

Disabled: This is a transitory state. In this state the plug-in ignores *both* inbound and outbound data. A plug-in *is* reset from this state. Note: this is a plug-in's *initial* state. It is brought to this state after POR[1] and connection to its corresponding socket.

The *Plug-in Management* register of a socket's interface is used to manage its corresponding plug-in. The register contains two, single-bit fields. Setting and clearing these fields drives the plug-in through its various states, while reading the register returns the plug-in's current state. Table 5 on page 45 enumerates the correspondence between register fields and state, while the register itself is specified and described in Section 2.4.1.

TABLE 5 Plug-in state

Register Fields		
Enabled	**Online**	**Corresponding State**
`False`	`False`	Disabled
`True`	`False`	Offline
`False`	`True`	*N/A*
`True`	`True`	Online

Note that a state which has the plug-in online, but *not* enabled is prohibited. As well, only four transitions are allowed between the three states. These transitions are enumerated and described in Section 2.4.1. The socket interface enforces the allowed states and their corresponding transitions. Any transaction which would set register fields to an invalid state is silently rejected with the plug-in remaining in its current state.

1. *Power-On-Reset*

The three canonical operations necessary to successfully manage a plug-in are implemented through a combination of sequencing state and transition. Those operations are described in Section 2.4.2.

2.4.1 Transitions

2.4.1.1 Online to offline

The plug-in is in its online state (see Table 5 on page 45). To transit the plug-in to offline:

- *Clear* the **Online** field of a socket's plug-in register. Clearing this field will cause the plug-in to emit an inbound *marker* frame. The format and structure of this frame is plug-in specific.
- Wait for the plug-in's inbound marker frame. This operation is plug-in specific. Once a marker is received the plug-in's inbound pipeline is drained and will not emit further frames until the plug-in is brought online.
- Exit.

2.4.1.2 Offline to online

The plug-in is in its offline state (see Table 5 on page 45). To transit the plug-in to online:

- *Set* the **Online** field of a socket's plug-in register.
- Exit.

2.4.1.3 Offline to disabled

The plug-in is in its offline state (see Table 5 on page 45). To transit the plug-in to disabled:

- *Clear* the **Enabled** field of a socket's plug-in register. Clearing this field *holds* the plug-in within reset, where it ignores inbound data as well as outbound frames.
- Exit.

2.4.1.4 Disabled to offline

The plug-in is in its disabled state (see Table 5 on page 45). To transit the plug-in to offline:

- *Set* the **Enabled** field of a socket's plug-in register. Setting this field *removes* the plug-in from being held in reset, triggering its plug-in specific reset logic.
- Exit.

2.4.2 Operations

2.4.2.1 Plug-in reset

A plug-in must be offline in order to be reset. To reset:

— Transit the plug-in to disabled (see Section 2.4.1.3).

— Transit back to offline (see Section 2.4.1.4).

— Return the plug-in's reset status (this operation is plug-in specific) and exit.

2.4.2.2 Bringing a plug-in online

The plug-in begins its life disabled with its inbound pipeline drained. To bring the socket and its corresponding plug-in online:

— Transit the plug-in to offline (see Section 2.4.1.4). This transition will *reset* the plug-in.

— Initialize the plug-in's corresponding socket.

— If the plug-in did *not* reset, transit back to disabled (see Section 2.4.1.3) and exit.

— Initialize the plug-in as appropriate (this operation is plug-in specific).

— If the plug-in did *not* initialize, transit back to disabled (see Section 2.4.1.3) and exit.

— Transit the plug-in to online (see Section 2.4.1.2).

— Exit.

2.4.2.3 Reset a plug-in while online

The plug-in begins in its online state. To reset:

— Transit the plug-in to offline (see Section 2.4.1.1).

— Reset the plug-in (see Section 2.4.2.1).

— If the plug-in did *not* reset, exit.

— Reinitialize the plug-in as appropriate (this operation is plug-in specific).

— If the plug-in did *not* initialize, exit.

— Transit the plug-in to online (see Section 2.4.1.2).

— Exit.

2.5 Registers

The interface defines *four* (4) sockets. Each socket is accessed through *five* registers located in the processor's GPO AXI space. Appendix B specifies the base address for each of the four sockets. Table 6 on page 48 enumerates the relative address of the five registers with respect to any one of the four bases.

TABLE 6 Socket Register Map

Offset[1]	Description	Access	See:
`00`	Outbound Free-List	*Read/Only*	Section 2.5.1
`04`	Outbound Work-List	*Write/Only*	Section 2.5.2
`08`	Inbound Pending-List	*Read/Only*	Section 2.5.3
`12`	Inbound Work-List	*Write/Only*	Section 2.5.4
`16`	*Reserved*	*N/A*	*N/A*
`20`	*Reserved*	*N/A*	*N/A*
`24`	*Reserved*	*N/A*	*N/A*
`28`	Plug-in Management	*Read/Write*	Section 2.5.5

1. In (decimal) *bytes*

2.5.1 Outbound Free-List

The interface for a socket's *outbound free-list* is a single, *read/only* I/O register bound to the read port of the **OB-Free-List** FIFO described in Figure 4. Access to this interface is through a *Load* instruction whose target is the physical address of this register. Table 6 on page 48 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a FIFO mapped to a read/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 2.6.

If, when read, the free-list was *not* empty, one entry is removed and its value returned. That value will always have its *low-order* bit *clear* (0). If *clear* the returned value is a reference to a descriptor to be used as the *operand* for an outbound instruction (see Section 2.5.2). Conversely, if, when read, the free-list *was* empty, the *low-order* bit of the returned value is *set* (1).

The structure of a returned value is illustrated in Figure 18 below:

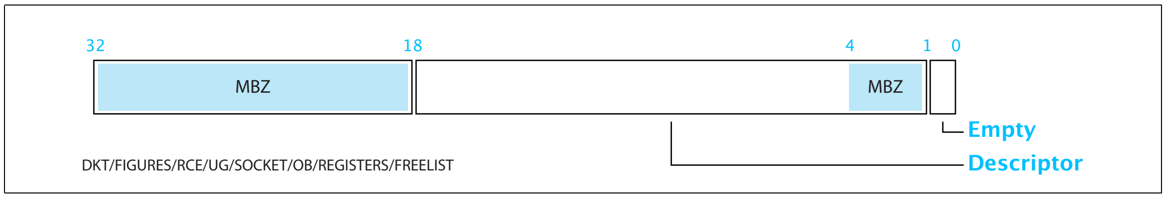

Figure 18 Returned value from outbound free-list.

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list was *not* empty and the **Descriptor** field described below contains a valid offset. If this flag is *true* (*set*) the list *was* empty. In such a eventuality the **Descriptor** field will be *zero* (0).

Descriptor: If the **Empty** flag described above is *false*, this field contains a reference to a buffer to be used as an instruction *descriptor* (see Section 2.2). A *reference* is expressed as a *byte* offset from the beginning of the OCM. For example, this field would have a value of *zero* (0) for a buffer located at the OCM's first address. Note that as all descriptors are *quadword* aligned the low-order four bits of this field will be *zero*. If the **Empty** flag described above is *true*, this field will be *zero* (0).

2.5.2 Outbound Work-List

The interface for a socket's *outbound work-list* is a single, *write/only* I/O register bound to the write port of the **OB-Work-List** FIFO described in Figure 4. Access to this interface is through a *Store* instruction whose target is the physical address of this register. Table 6 on page 48 specifies the register's relative address and using Appendix B, its absolute address. The access policy for a FIFO mapped to a write/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 2.6.

The value written to this register is an outbound *instruction*. Outbound instructions are described in Section 2.2 and their structure illustrated in Figure 19 below:

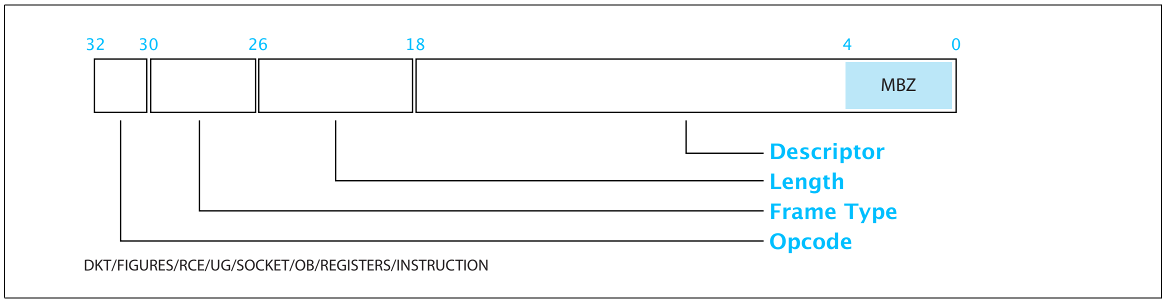

Figure 19 Outbound instruction.

Where:

Descriptor: This field contains a reference to the instruction's *descriptor* (see Section 2.2). A *reference* is expressed as a *byte* offset from the beginning of the OCM. For example, this field would have a value of *zero* (0) for a reference to a descriptor located at the OCM's first address. Note that as all descriptors are *quadword* aligned the low-order four bits of this field must be *zero*. A descriptor's structure is expected to be consistent with the **Opcode** field described below. If not the case the result is unpredictable.

Length: This field contains the length of the outbound header data within the descriptor whose reference is contained in the **Descriptor** field described above. Lengths are expressed in units of *quadwords* (64-bits) and range from a minimum of *zero* (0) to a maximum equal to the MOH of the socket's corresponding plug-in (see Section 2.2). A length of *zero* indicates the frame consists of *only* payload. If the **Opcode** field is *zero*, the value of this field is ignored, but should be set to *zero* (0).

Frame Type: This field contains the *type* (see Section 1.3.3) of the frame contained within the instruction's corresponding descriptor. A frame's type is expressed as a small enumeration ranging from *zero* (0) to *fifteen* (15). The permissible enumerations as well as their interpretation are entirely determined by the socket's corresponding plug-in. If the **Opcode** field is *zero*, the value of this field is ignored, but should be *zero* (0).

Opcode: This field specifies the operation requested of the socket's outbound transfer engine. The descriptor referenced by the **Descriptor** field above is assumed formatted appropriately for the operation. The opcode is a small enumeration ranging from *zero* (0) to *three* (3). The mapping between enumeration and operation is specified in Table 3 on page 32.

2.5.3 Inbound Pending-List

The interface for a socket's *inbound pending-list* is a single, *read/only* I/O register bound to the read port of the **IB-Pending-List** FIFO described in Figure 4. Access to this interface is through a *Load* instruction whose target is the physical address of this register. Table 6 on page 48 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a FIFO mapped to a read/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 2.6.

If, when read, the pending-list was *not* empty, one entry is removed and its value returned. That value will always have its *low-order* bit *clear* (0). If *clear* the returned value is a description of an inbound frame (see Section 2.3). Conversely, if, when read, the pending-list *was* empty, the *low-order* bit of the returned value is *set* (1).

The structure of a returned value is illustrated in Figure 20 below:

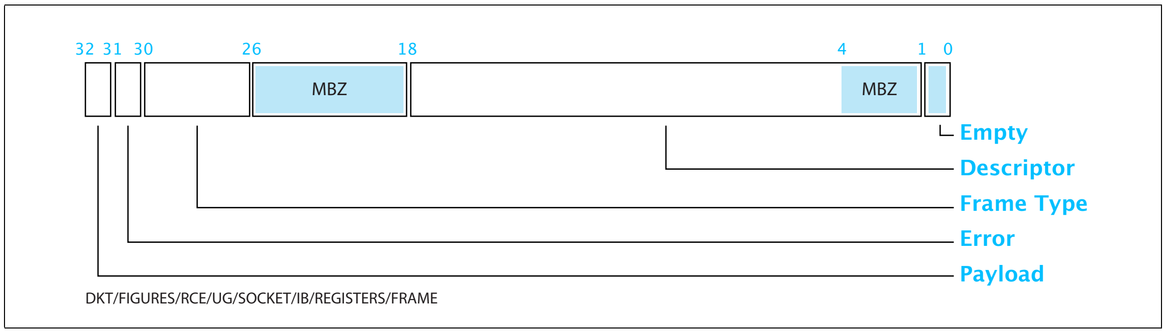

Figure 20 Returned value from inbound pending-list.

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list is *not* empty and the **Descriptor** field described below contains the offset to a buffer containing header data for an incoming frame. If this flag is *true* (*set*) the list *is* empty. In such a case all other fields described below will be *zero* (0).

Descriptor: If the **Empty** flag described above is *false*, this field contains a reference to a descriptor. That descriptor contains the header of an inbound frame (see Section 2.3). A *reference* is expressed as a *byte* offset from the beginning of the OCM. For example, this field would have a value of *zero* (0) for a buffer located at the OCM's first address. Note that as all descriptors are *quadword* aligned the low-order *four* bits of this field will be *zero*. If the **Empty** flag described above is *true*, this field will be *zero* (0).

Frame Type: This field contains the *type* (see Section 1.3.3) of the frame associated with the header whose reference is contained in the **Descriptor** field described above. A frame's type is expressed as a small enumeration ranging from *zero* (0) to *fifteen* (15). The permissible enumerations as well as their interpretation are entirely determined by the socket's corresponding plug-in. If the **Empty** field is *true*, the value of this field will be *zero* (0).

Error: This field contains a single bit flag signalling whether or not the received header data are in error. If this flag is *False* (clear), header data were received error free. If this flag is *True* (set), header data were received in error. In such a case the header's structure may vary from its error free form. For example, the header data could contain further information on the cause of the error. Note: the value of this flag does not effect the **Payload** flag described below. A frame with an header in error may still contain a payload. If the **Empty** field is *true*, the value of this field will be *zero* (0).

Payload: This field contains a single bit flag specifying whether the frame whose header data were returned in the buffer referenced by the **Descriptor** field described above has an associated payload. If this flag is *False* (clear), the header data contained in the header buffer *is* the entire frame. If this flag is *True* (set), the frame contains both header and payload. The data for that payload is obtained as described in Section 2.3. If the **Empty** field is *true*, the value of this field will be *zero* (0).

2.5.4 Inbound Work-List

The interface for a socket's *inbound work-list* is a single, *write/only* I/O register bound to the write port of the **IB-Work-List** FIFO described in Figure 4. Access to this interface is through a *Store* instruction whose target is the physical address of this register. Table 6 on page 48 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a FIFO mapped to a write/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 2.6.

The value written to this register is an inbound *instruction*. Inbound instructions are described in Section 2.3 and their structure illustrated in Figure 21 below:

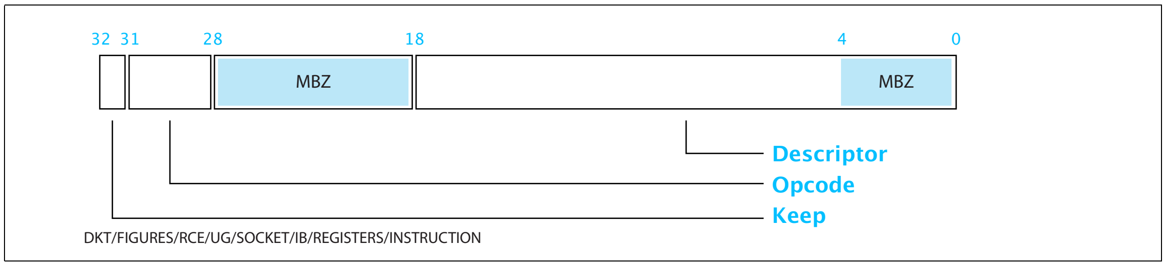

Figure 21 Inbound instruction.

Where:

Descriptor: This field contains a reference to the instruction's *descriptor* (see Section 2.2). A *reference* is expressed as a *byte* offset from the beginning of the OCM. For example, this field would have a value of *zero* (0) for a reference to a descriptor located at the OCM's first address. Note that as all descriptors are *quadword* aligned the low-order *four* bits of this field must be *zero*. A descriptor's structure is expected to be consistent with the **Opcode** field described below. If not the case the result is unpredictable.

Opcode: This field specifies the operation requested of the socket's inbound transfer engine. The descriptor referenced by the **Descriptor** field described above is assumed formatted appropriately for the operation. The opcode is a small enumeration ranging from *zero* (0) to *seven* (7). The mapping between enumeration and operation is specified in Table 4 on page 39.

Keep: This field contains a single bit flag specifying whether the descriptor referenced by the Descriptor field described above should be returned to its free-list. If this flag is *False* (clear), the descriptor is returned to its free-list. If this flag is *True* (set), the descriptor is *not* returned to its free-list.

2.5.5 Plug-in Management

The interface to manage a socket's *plug-in* is a single, *read/write* I/O register. Access to this interface is through either a *Load* or *Store* instruction whose target is the physical address of this register. Table 6 on page 48 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a read/write register is described in Section 1.4.1. The structure of this register is illustrated in Figure 22 below:

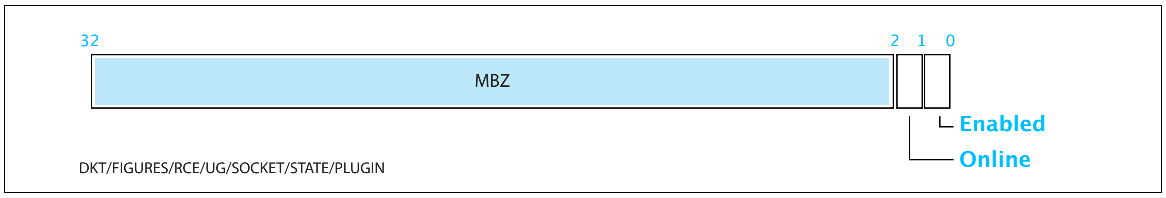

Figure 22 Plug-in Management Register

Where:

Enabled: This field is a single bit flag indicating whether or not the plug-in is enabled. If the flag is *true* (*set*), the plug-in is *enabled*. If this flag is *false* (*clear*) the plug-in is *disabled*. Table 5 on page 45 enumerates how the value of this field effects plug-in state.

Online: This field is a single bit flag indicating whether or not the plug-in is *online*. If the flag is *true* (*set*), the plug-in is *online*. If this flag is *false* (*clear*) the plug-in is *offline*. Table 5 on page 45 enumerates how the value of this field effects plug-in state.

2.6 Interrupt Sources

The interface defines *four* (4) sockets and in turn, each socket defines *four* (4) different *interrupt sources*. An interrupt source is a firmware signal which, while asserted and not masked may trigger a processor interrupt. Each source is uniquely identified by its source *address*. Table 9 on page 65 specifies the *base* source address for each of the four sockets. Table 7 on page 54 enumerates the relative address of a source with respect to any one of the four bases. See Chapter 4 for a discussion of the interrupt interface.

TABLE 7 Socket Interrupt Source Map

Offset	Description of signal	Associated register
`0`	Outbound Free-List is *Almost-Empty*	Section 2.5.1
`1`	Outbound Work-List is *Full.* This corresponds to the *Write-Fault* (`WRERR`) flag of the FIFO.	Section 2.5.2
`2`	Inbound Pending-List is *Not-Empty*	Section 2.5.3
`3`	Inbound Work-List is *Full.* This corresponds to the *Write-Fault* (`WRERR`) flag of the FIFO.	Section 2.5.4

Chapter 3
The Rendezvous Interface

3.1 Overview

The rendezvous interface allows an entity receiving or transmitting a frame to coordinate the *completion* of its transfer with either itself or another entity. Such coordination is a frame *rendezvous*. The entity receiving or transmitting a frame is said to *arrange* the rendezvous and the entity waiting on the frame's completion is said to *attend* the rendezvous. Associated with each declared rendezvous is a *30-bit* message called rendezvous *context*. Context is used to uniquely tag or identify a specific rendezvous. Context is established by an *arranger* and delivered to an *attendee* when a rendezvous's corresponding frame transfer is complete.

For example, a buffer is allocated for an outbound frame to transmit. Only when the frame is completely transferred can its associated buffer be safely deallocated. In such a case, a rendezvous could be used to coordinate the end of transmission with the buffer's deallocation, with the rendezvous context containing the frame buffer's address.

The interface supports up to *thirty-two* (32) concurrent rendezvous. Each independent rendezvous is associated with a rendezvous *channel*. Channels are numbered and addressed from *zero* (0) to *thirty-one* (31). The channel number is used to coordinate rendezvous between *arranger* and *attendee*.

As a rendezvous is associated with a frame transfer the socket interface described in Chapter 2 is used by the arranger to establish and initiate a rendezvous, while the channel interface described in Section 3.2 is used by the attendee to wait for context at a rendezvous. The first *thirty-one* (31) rendezvous channels are available for generic rendezvous as outlined in Section 3.1.1, while the last channel[1] is reserved for transfer faults as outlined in Section 3.3.2.

In using the rendezvous interface two cases must be considered. In the first and dominant case, the socket is able to successfully complete the frame transfer associated with the rendezvous. This case is discussed in Section 3.1.1. In the second case, for various pathological

1. Numbered *thirty-one* (31).

reasons likely outside the arranger's control a socket cannot successfully transfer a frame. This results in a *transfer fault*. For example, a DMA error occurs in moving a frame from memory to plug-in or from plug-in to memory. Transfer faults are discussed in Section 3.3.2.

A block diagram of the interface is illustrated in Figure 23:

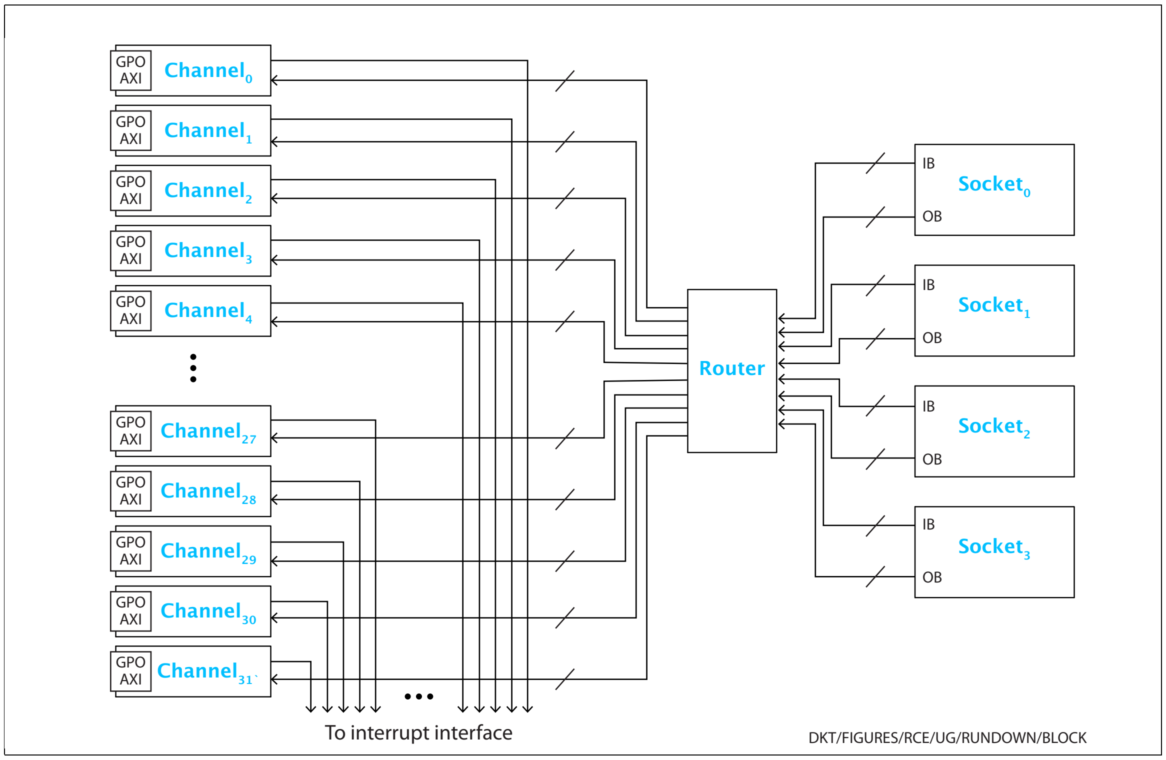

Figure 23 Block diagram of the Rendezvous Interface

The interface connects the four sockets to thirty-two rendezvous channels through a *router*. The router itself serves only the implementation and has no public interface. It functions to both route and transfer rendezvous context from its socket to its appropriate rendezvous channel. A socket contains *two 512* entry FIFOs, each independently buffering a socket's in-flight inbound and outbound rendezvous requests. Each entry on a FIFO contains the context and destination information for a single rendezveous.The read port of each FIFO is connected directly to the router. Each channel also contains a *512* entry FIFO, but whose *write* rather than read port is directly connected to the router. Entries in its FIFO contains only rendezvous context.

Therefore, the router must respond *fairly* to up to eight asynchronous input *sources* and distribute their context to up to thirty-two *destinations*. To do so, arbitration is conducted in a round-robin fashion using a priority sorted queue whose entries correspond to its eight sources. A destination is considered *ready* while its FIFO is *not-full*. A source is considered *ready* while its FIFO is *not-empty* and the destination pointed to by the entry at its FIFO's head is also ready. The Both engines communicate though a socket's *inbound free-list*. router performs one transfer at a time. It initiates a transfer when two conditions are met: First, the router is idle and second, one or more of its sources are *ready*. When these conditions are met the router will:

- — Remove the highest priority, *ready* source from its priority queue.
- — Remove an entry from the source's FIFO.
- — Determine from the entry the rendezvous *destination*.
- — Determine from the entry the rendezvous *context*.
- — Insert rendezvous context on destination's FIFO.
- — Re-insert the removed source at the tail of its priority queue.

3.1.1 Rendezvous and extended context

Rendezvous are *arranged* through either the outbound socket interface described in Section 2.2 or the inbound socket interface described in Section 2.3. Independent of transfer direction however, a rendezvous is always *attended* through the channel interface discussed in Section 3.2 below. To execute a rendezvous both arranger and attendee must *a-priori* agree on usage of a rendezvous channel. How that agreement is reached is beyond the scope of the rendezvous interface; however, once in agreement an arranger will reference the channel through its *number* and an attendee through its context *register* as described in Section 3.3.1.

Arranger and attendee must also agree on rendezvous context. Context is always *established* (written) by the arranger and always *acquired* (read) by the attendee. Specifically, context is written using the socket interface and read using the channel interface.

To arrange a rendezvous requires either an outbound or inbound descriptor (see Sections 2.1.3 and 2.1.6) as well as a channel number. With descriptor and channel allocated a rendezvous is established by simply writing the rendezvous channel number and necessary context into the descriptor. Figure 24 illustrates the relationship between descriptor, rendezvous channel and context:

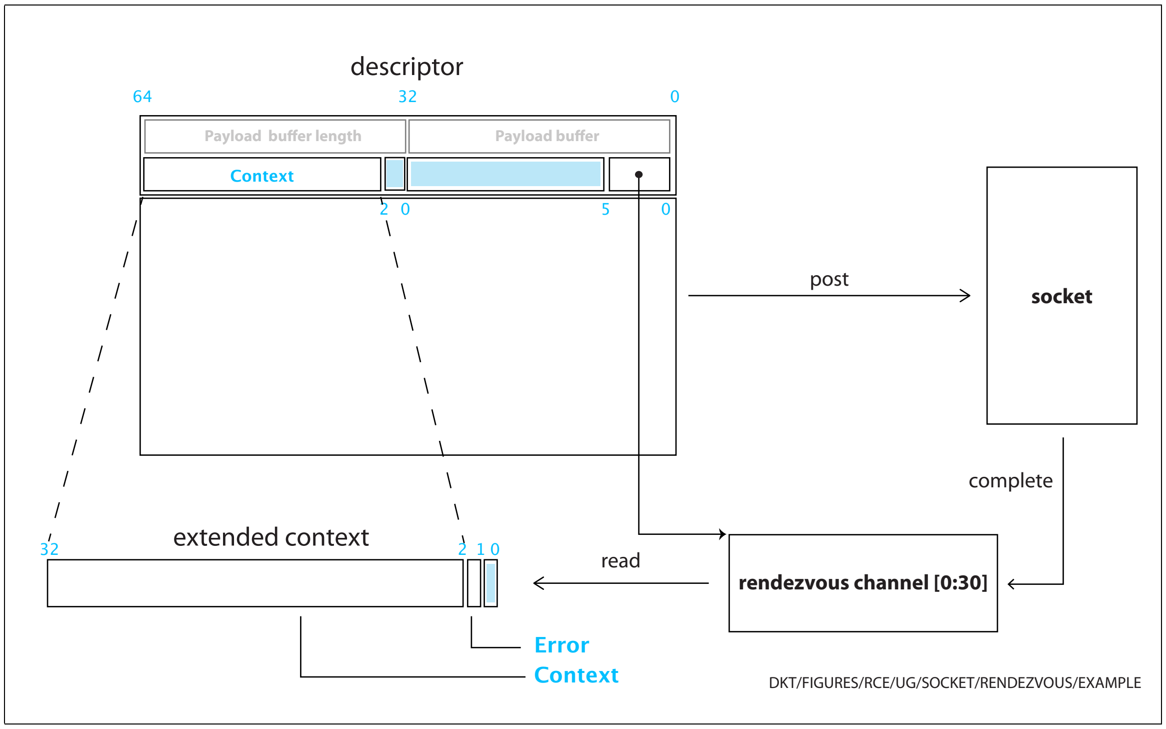

Figure 24 Socket descriptor, rendezvous channel and context

Note that for any given descriptor the channel number is written starting at the *zeroth-bit* offset of its *third* word and the context is written starting at the *second-bit* offset of its *fourth* word. Any unused field of either word is set to *zero* (0). Once established a rendezvous begins by posting the descriptor to its appropriate inbound or outbound work-list. When the work associated with the descriptor completes the socket delivers both context and status to the specified rendezvous channel. The attendee waits for the rendezvous context by reading the context register of a rendezvous channel. That channel corresponds to the channel number written to the descriptor (see Section 3.3.1). The *extended* context returned by reading this register contains two fields:

— The 30-bit rendezvous context written into the descriptor by its arranger.

— A error flag indicating whether the socket used with the rendezvous failed to transfer its associated frame. The flag is set if *either* the socket's plug-in *or* its transfer engine fails to transfer. In addition, if a transfer engine failed a fault message is generated and sent by the socket to the fault channel as described below in Section 3.1.2.

3.1.2 Rendezvous and frame transfer-fault

As a transfer fault, if present, occurs *after* a rendezvous begins, a rendezvous which results in a fault is arranged and handled by its attendee no differently then any normal rendezvous as described above in Section 3.1.1. The only difference is the returned extended context, where in this case its **Error** field (see Section 3.3.1) will always be *set*.

However, a socket engine incurring a fault has one additional responsibility. In such a case, the engine must, in addition to its normal rendezvous handling, also generate and send a *second* message to the rendezvous interface. This message is used to characterize the fault. Unlike a normal rendezvous initiated by the arranger this message is always sent, independent of engine or socket to a fixed, well known rendezvous channel, which in this case is the last channel or number *thirty-one* (31). This process is illustrated in Figure 25:

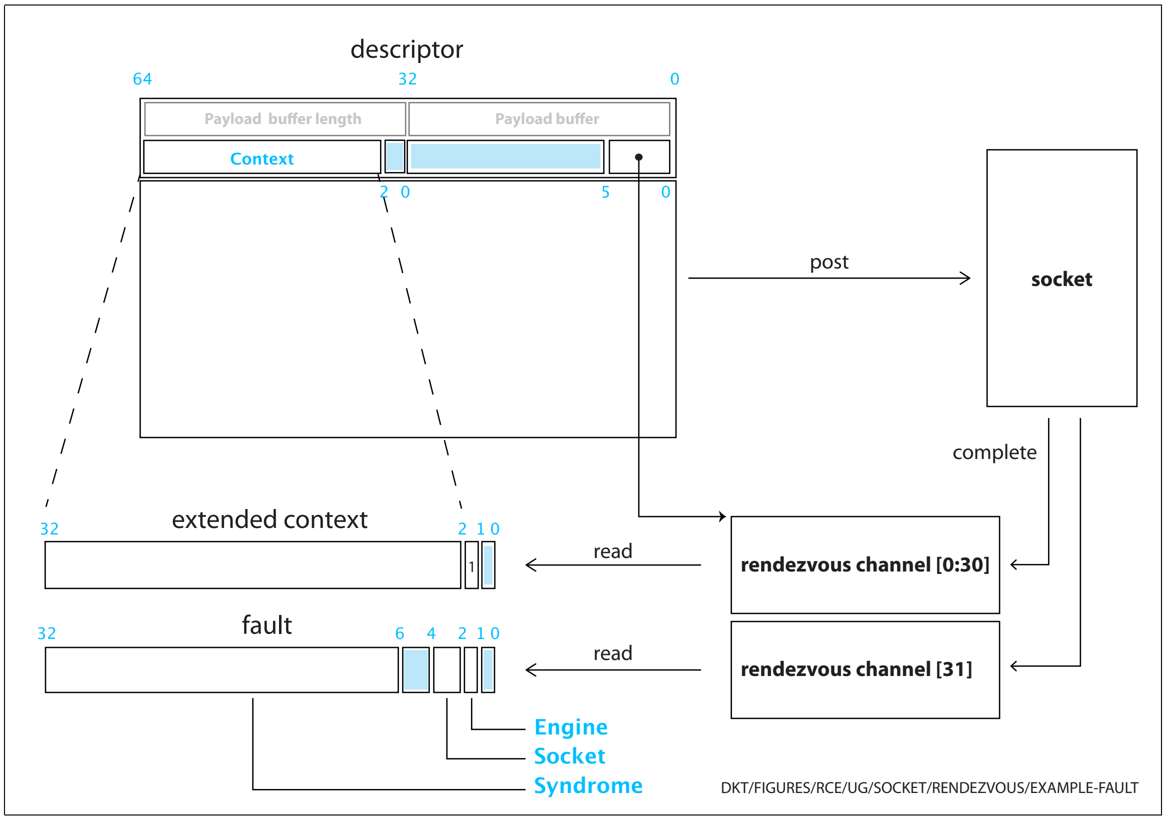

Figure 25 Rendezvous with transfer fault

An attendee waits on a fault message in a fashion no different then any other rendezvous. The only difference is the rendezvous channel which is always thirty-one. Note that the fault message specifies the engine and socket which incurred the fault as well as the reason. or syndrome.The structure of a fault message is described in Section 3.3.2.

3.2 Channel Interface

TBD.

Figure 26 is a block diagram illustrating the channel interface:

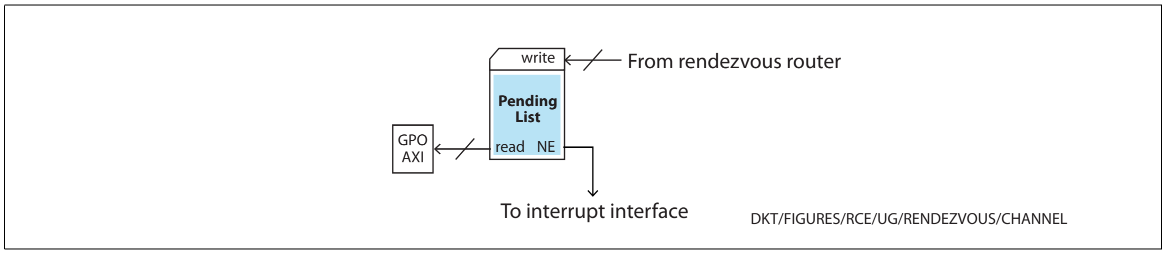

Figure 26 Block diagram of the Rendezvous Channel

TBD

3.3 Registers

As discussed in Section 3.2 the rendezvous interface defines *thirty-two* (32) channels. Each channel is accessed through a single, *read/only* I/O register located in the processor's GPO AXI space and bound to the read port of the *pending-list* FIFO described in Figure 26. Access to this interface is through a *Load* instruction whose target is the physical address of this register. The access policy for a FIFO mapped to a read/only register is described in Section 1.4.1. Note that this FIFO is mapped to an *interrupt source* as discussed in Section 3.4.

If, when read, a pending-list was *not* empty, one entry is removed and its value returned. That value will always have its *low-order* bit *clear* (0). Conversely, if, when read, a pending-list *was* empty, the *low-order* bit of the returned value is *set* (1).

The interface's set of thirty-two registers is organized as a thirty-two element *vector*. For example, the vector's *fourth* element contains the pending-list for Channel$_3$.

Vector elements *zero* through *thirty* (0 - 30), when read return a value whose structure is described in Section 3.3.1, while element *thirty-one* (31), when read returns the value described in Section 3.3.2. Appendix B specifies the base address for the vector.

3.3.1 Rendezvous Pending-List

When read, the register associated with Rendezvous channels *zero* (0) through *thirty* (30) returns a rendezvous's *extended context* as described in Section 3.1.1. The returned value is described in Figure 27 described below:

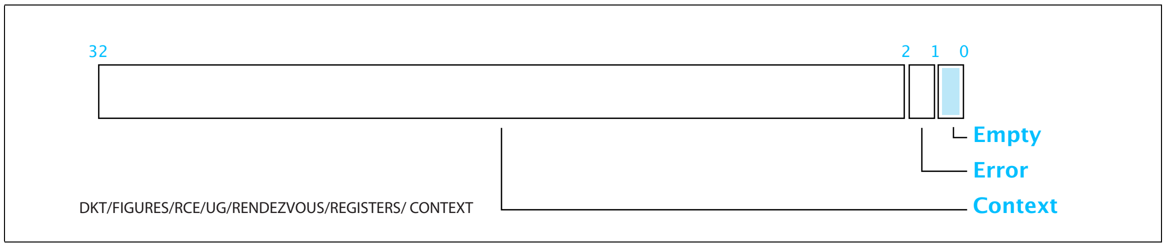

Figure 27 Rendezvous extended context register

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list is *not* empty and the **Context** and **Error** fields described below are valid. If this flag is *true* (*set*) the list *is* empty. In such a case all fields described below will also be *zero* (0).

Error: This field contains a single bit flag signalling whether or not the work associated with the rendezvous completed successfully. This flag is *False* (clear) if that work completed successfully and *True* (set) if that work failed. Note that independent of error the **Context** field described below is always valid. If the **Empty** field is *true*, this field will be *zero* (0).

Context: If the **Empty** flag described above is *false*, this field contains the context defined by the arranger of the corresponding rendezvous. See Section 3.1.1 for a discussion of meaning and usage of this field. If the **Empty** flag described above is *true*, this field will be *zero* (0).

3.3.2 Transfer-fault Pending-List

When read, the register associated with Rendezvous channel *thirty-one* (30) returns a rendezvous's *transfer-fault* as described in Section 3.1.2. The returned value is described in Figure 28 described below:

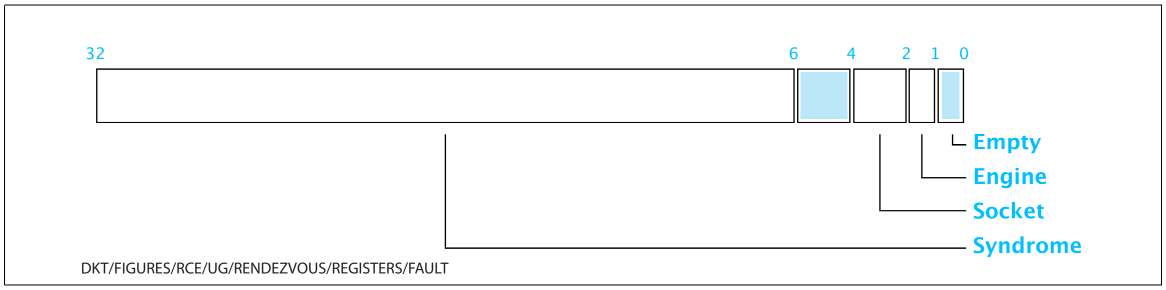

Figure 28 Transfer fault register

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list is *not* empty and all fields described below are valid. If this flag is *true* (*set*) the list *is* empty. In such a case all the fields described below will be *zero* (0).

Engine: If the **Empty** flag described above is *false*, this field specifies whether the fault occurred with either a socket's inbound or outbound engine. This field is *Clear* (0) if the fault occurred with the *Inbound* engine and *Set* (1) if the fault occurred with the *Outbound* engine. If the **Empty** field is *true*, this field will be *zero* (0).

Socket: If the **Empty** flag described above is *false*, this field contains an enumeration which identifies the socket in which the fault occurred. If the **Empty** field is *true*, this field will be *zero* (0).

Syndrome: If the **Empty** flag described above is *false*, this field contains an enumeration of the corresponding fault. If the **Empty** flag described above is *true*, this field will be *zero* (0).

3.4 Interrupt Sources

The rendezvous interface defines *thirty-two* (32) channels (see Section 3.2). In turn, each channel defines *one* (1) *interrupt source*. An interrupt source is a firmware signal which, while asserted and not masked may trigger a processor interrupt. Each source is uniquely identified by its source *address*. Table 8 on page 62 specifies the *base* address for the interrupt sources defined by the thirty-two channels. Table 9 on page 65 enumerates the relative address of any one source with respect to its base. See Chapter 4 for a discussion of the interrupt interface.

TABLE 8 Rendezvous Interrupt Source Map

Offset	Description	See:
0	Rendezvous Pending-List is *Not-Empty* (Channel_0)	Section 3.3.1
1	Rendezvous Pending-List is *Not-Empty* (Channel_1)	Section 3.3.1
2	Rendezvous Pending-List is *Not-Empty* (Channel_2)	Section 3.3.1
3	Rendezvous Pending-List is *Not-Empty* (Channel_3)	Section 3.3.1
4	Rendezvous Pending-List is *Not-Empty* (Channel_4)	Section 3.3.1
5	Rendezvous Pending-List is *Not-Empty* (Channel_5)	Section 3.3.1
6	Rendezvous Pending-List is *Not-Empty* (Channel_6)	Section 3.3.1
7	Rendezvous Pending-List is *Not-Empty* (Channel_7)	Section 3.3.1
8	Rendezvous Pending-List is *Not-Empty* (Channel_8)	Section 3.3.1
9	Rendezvous Pending-List is *Not-Empty* (Channel_9)	Section 3.3.1

TABLE 8 Rendezvous Interrupt Source Map

Offset	Description	See:
`10`	Rendezvous Pending-List is *Not-Empty* (Channel$_{10}$)	Section 3.3.1
`11`	Rendezvous Pending-List is *Not-Empty* (Channel$_{11}$)	Section 3.3.1
`12`	Rendezvous Pending-List is *Not-Empty* (Channel$_{12}$)	Section 3.3.1
`13`	Rendezvous Pending-List is *Not-Empty* (Channel$_{13}$)	Section 3.3.1
`14`	Rendezvous Pending-List is *Not-Empty* (Channel$_{14}$)	Section 3.3.1
`15`	Rendezvous Pending-List is *Not-Empty* (Channel$_{15}$)	Section 3.3.1
`16`	Rendezvous Pending-List is *Not-Empty* (Channel$_{16}$)	Section 3.3.1
`17`	Rendezvous Pending-List is *Not-Empty* (Channel$_{17}$)	Section 3.3.1
`18`	Rendezvous Pending-List is *Not-Empty* (Channel$_{18}$)	Section 3.3.1
`19`	Rendezvous Pending-List is *Not-Empty* (Channel$_{19}$)	Section 3.3.1
`20`	Rendezvous Pending-List is *Not-Empty* (Channel$_{20}$)	Section 3.3.1
`21`	Rendezvous Pending-List is *Not-Empty* (Channel$_{21}$)	Section 3.3.1
`22`	Rendezvous Pending-List is *Not-Empty* (Channel$_{22}$)	Section 3.3.1
`23`	Rendezvous Pending-List is *Not-Empty* (Channel$_{23}$)	Section 3.3.1
`24`	Rendezvous Pending-List is *Not-Empty* (Channel$_{24}$)	Section 3.3.1
`25`	Rendezvous Pending-List is *Not-Empty* (Channel$_{25}$)	Section 3.3.1
`26`	Rendezvous Pending-List is *Not-Empty* (Channel$_{26}$)	Section 3.3.1
`27`	Rendezvous Pending-List is *Not-Empty* (Channel$_{27}$)	Section 3.3.1
`28`	Rendezvous Pending-List is *Not-Empty* (Channel$_{28}$)	Section 3.3.1
`29`	Rendezvous Pending-List is *Not-Empty* (Channel$_{29}$)	Section 3.3.1
`30`	Rendezvous Pending-List is *Not-Empty* (Channel$_{30}$)	Section 3.3.1
`31`	*Transfer-Fault* Pending-List is *Not-Empty* (Channel$_{31}$)	Section 3.3.2

Chapter 4
The Interrupt Interface

4.1 Interrupt Sources

The interrupt interface is designed to manage up to *512* independent interrupt *sources*, where a source is defined as any one firmware block capable of the interrupting a processor core. Each source is uniquely assigned and identified by a source *address* ranging from *zero* (0) to *five-hundred-eleven* (511). Table 9 on page 65 summarizes assignment of source address to firmware block.

TABLE 9 Interrupt sources

Offset	Length	Description	See:
0	4	Socket_0	Section 2.6
4	4	Socket_1	Section 2.6
8	4	Socket_2	Section 2.6
12	4	Socket_3	Section 2.6
16	48	*Reserved*	*N/A*
64	32	Rendezvous Pending-Lists	Section 3.4
96	32	*Reserved*	*N/A*
128	4	Utility Free-Lists	Appendix A.2
132	60	*Reserved*	*N/A*
192	1	BSI Pending-Change-List	Section 5.3
193	63	*Reserved*	*N/A*
256	256	User defined	*N/A*

TBD.

4.1.1 Interrupt Groups

The ARM *Interrupt Controller* supports up to *sixteen* (16) interrupts from the FPGA fabric. Clearly, this does not allow for a direct map between interrupt and source. Therefore, some sort of source aggregation must be provided by the interface. A unit of aggregation is called a *Group* and Figure 29 illustrates a group's block diagram:

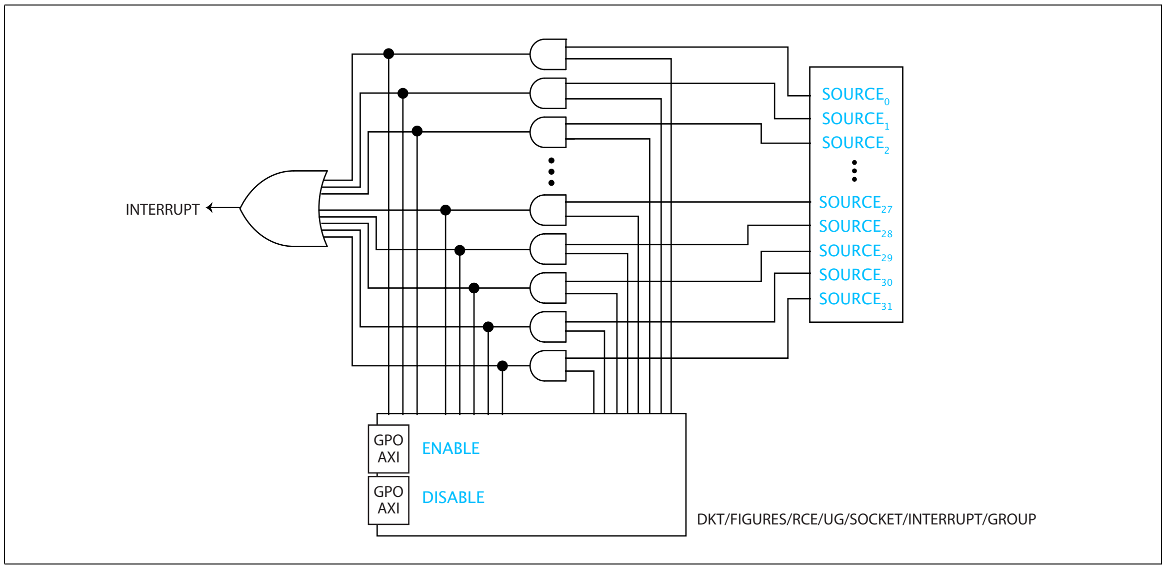

Figure 29 Block diagram of an Interrupt Group

One group sums up to *thirty-two* (32) individual sources to form a single interrupt. To handle the up to *512* potential sources requires *sixteen* groups. The interrupt from each group is connected to one of the sixteen lines from the processor interrupt controller. Each group is numbered and addressed from *zero* (0) to *fifteen* (15) and within a group its thirty-two sources are addressed relative to that group and numbered from *zero* (0) to *thirty-one* (31).

For any one group the interface provides for each of its thirty-two sources to be individually masked either on or off. The register interface to manage masking for a group is described in Section 4.2.2 and Section 4.2.3.

4.1.2 Interrupt Remapping

Figure 30 is a block diagram illustrating the entire interrupt architecture:

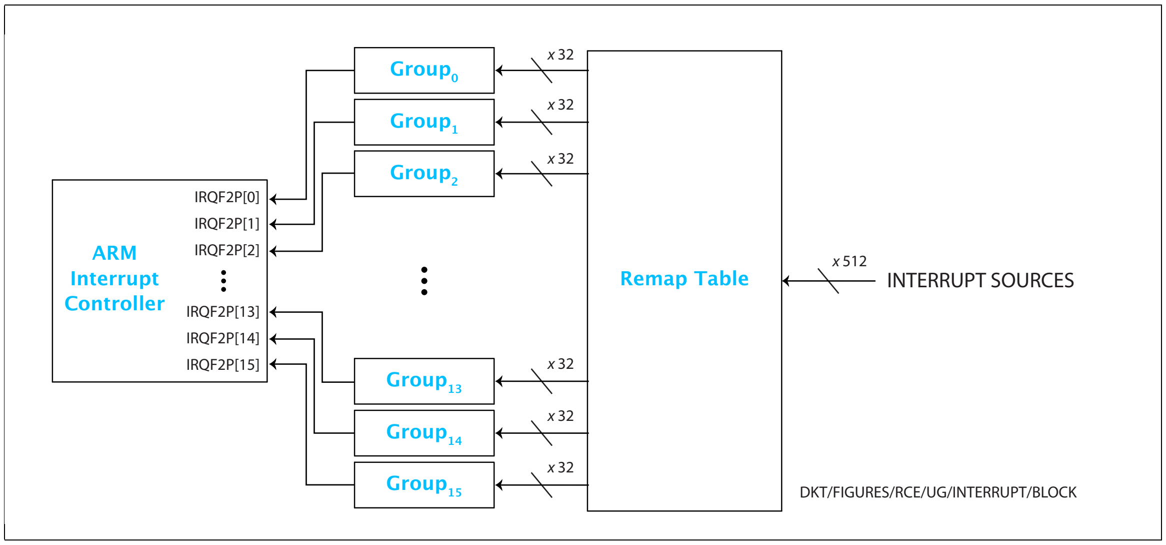

Figure 30 Block diagram of the Interrupt interface.

Note the mapping between groups and interrupt controller lines is fixed. That mapping is enumerated in Table 10 below:

TABLE 10 Interrupt group address and interrupt lines

Group address	Interrupt line	Group address	Interrupt line
0	IRQF2P[0]	1	IRQF2P[1]
2	IRQF2P[2]	3	IRQF2P[3]
4	IRQF2P[4]	5	IRQF2P[5]
6	IRQF2P[6]	7	IRQF2P[7]
8	IRQF2P[8]	9	IRQF2P[9]
10	IRQF2P[10]	11	IRQF2P[11]
12	IRQF2P[12]	13	IRQF2P[13]
14	IRQF2P[14]	15	IRQF2P[15]

Between source and group is the *Remap Table*. This table allows the set of 512 sources to be arbitrarily mapped to the set of the 16 groups. It contains 512 entries, one for each source. An entry specifies whether or not the source is mapped and if so, to which group and within that group to which source offset. The register interface used to configure this mapping is described in Section 4.2.1.

4.2 Registers

4.2.1 Source remap

As described in Section 4.1.2 the interface allows each of its potential 512 interrupt sources to be remapped. Each source is accessed through a single register located in the processor's GPO AXI space. The set of 512 registers is organized as a 512 element *vector*. Each element of the vector corresponds to the remap directions for one source. For example, the vector's *second* element contains the remap specification for *Source*$_1$. Appendix B specifies the base address for the vector. An element may be either read or written. Processor access is though either a *Load* or *Store* instruction whose target is an address corresponding to one element of the vector. Section 1.4.1 describes the access policy for *read/write* registers.

Each element specifies whether or not its corresponding source shall be mapped and if mapped, to which of the interface's sixteen groups (see Section 4.1.1). Figure 31 below illustrates the structure of one element:

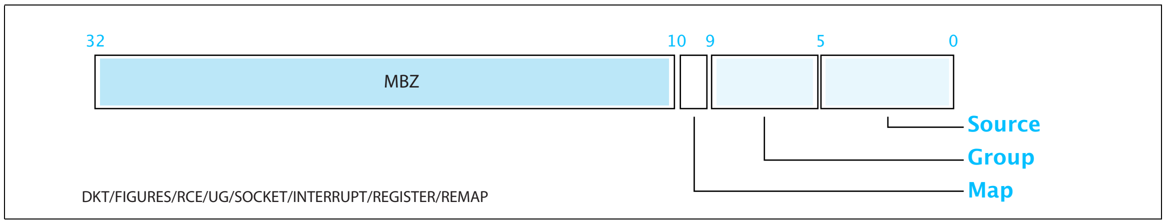

Figure 31 Source Interrupt remap register

Where:

Source:	This field specifies *where* in the group, as determined by the **Group** field described below the source is represented. This field is *only* valid if the **Map** field described above is *set*. If not the case, this field will be *zero* (0).
Group:	This field specifies which of the *sixteen* groups (see Section 4.1.1) manages the corresponding source. This field is *only* valid if the **Map** field described above is *set*. If not the case, this field will be *zero* (0).
Map:	This one-bit field is a flag determining whether the source is mapped. If this field is *set*, the source is mapped. In such a case, its corresponding interrupt is managed by the group enumerated by the **Group** field below and the **Source** field determines *where* in that group the source is represented. If this field is *clear*, the source is *not* mapped and cannot generate an interrupt. In such a case, both **Group** and **Source** fields will be *zero* (0).

4.2.2 Source Group Enables

As described in Section 4.1.1 the interface defines *sixteen* (16) group *enables*. Each group is accessed through a single register located in the processor's GPO AXI space. The set of sixteen registers is organized as a sixteen element *vector*. Each element of the vector corresponds to one group. For example, the vector's *third* element contains the *Group*$_2$ *Enables*. Appendix B specifies the base address for the vector. An element may be either read or written. Processor access is though either a *Load* or *Store* instruction whose target is an address corresponding to one element of the vector. Section 1.4.1 describes the access policy for *read/write* registers.

One group represents *thirty-two* (32) interrupt sources. Therefore, one register manges the *enables* for thirty-two individual sources. Each source is represented as a *one-bit* field whose bit *offset* corresponds to its source number. Field value interpretation is dependent on whether the register is either read or written. When a field is written, its value determines whether or not its corresponding source is enabled. If a field is *set*, the source, if currently disabled will be *enabled*. If the field is *clear*, the corresponding source is ignored and its state, whether enabled or disabled remains unchanged. When read, the returned value determines whether or not the corresponding source was enabled. If the field is *set*, the source was *enabled*. If clear, the source was *disabled*.

Note: this vector manages *only* group enables. To manage group *disables* see the vector described in Section 4.2.3. Figure 32, below illustrates the structure of one element of the group enable vector:

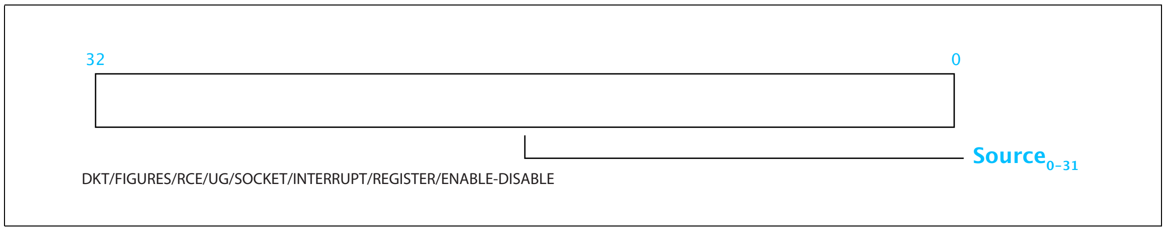

Figure 32 Interrupt group enables

4.2.3 Source group disables

As described in Section 4.1.1 the interface defines *sixteen* (16) group *disables*. Each group is accessed through a single register located in the processor's GPO AXI space. The set of sixteen registers is organized as a sixteen element *vector*. Each element of the vector corresponds to one group. For example, the vector's *third* element contains the *Group*$_2$ *Disables*. Appendix B specifies the base address for the vector. An element may be either read or written. Processor access is though either a *Load* or *Store* instruction whose target is an address corresponding to one element of the vector.

One group represents *thirty-two* (32) interrupt sources. Therefore, one register manges the *disables* for thirty-two individual sources. Each source is represented as a *one-bit* field whose bit *offset* corresponds to its source number. Field value interpretation is dependent on whether the register is either read or written. When a field is written, its value determines whether or not its corresponding source is disabled. If a field is *set*, the source, if currently enabled will be *disabled*. If the field is *clear*, the corresponding source is ignored and its state, whether enabled or disabled remains unchanged.

Reading a field triggers an internal *read-modify-write* and consequently is *destructive*. This type of access is intended to facilitate interrupt service code and may result in unintended side-effects if used elsewhere. See Section 4.2.2 for a non-destructive read. When read, a field will, if *set*, become *clear*. The returned value specifies a field's value immediately *before* it was modified. If *set*, the corresponding source was enabled and is now disabled. If *clear*, the source was disabled and remains disabled.

Note: this vector manages *only* group disables. To manage group *enables* see the vector described in Section 4.2.2. Figure 33, below illustrates the structure of one element of the group enable vector:

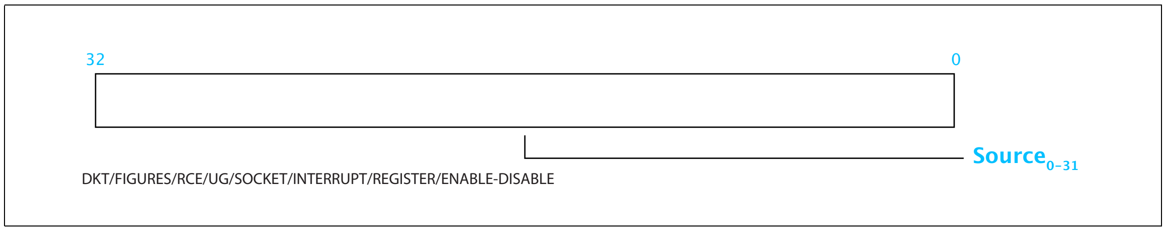

Figure 33 Interrupt group disables

Chapter 5
The Bootstrap Support Interface

5.1 Overview

TBD.

Figure 34 Block diagram of the Boot Support Interface

TBD

5.2 Registers

TBD

5.2.1 I2C Signal

The structure for xxx is illustrated in Figure 35:

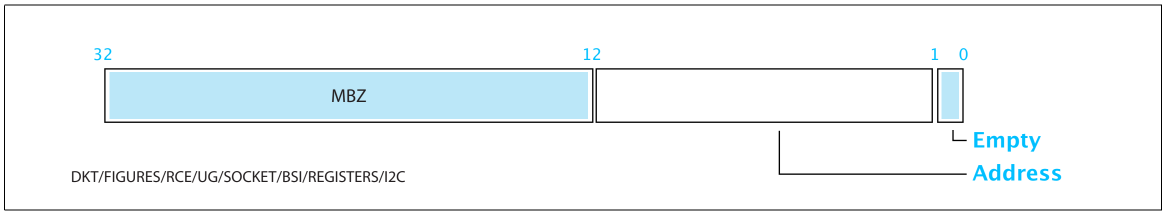

Figure 35 I2C Signal

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list was *not* empty and the **Address** field described below contains a valid offset. If this flag is *true* (*set*) the list *was* empty. In such a eventuality the **Address** field will be *zero* (0).

Address: If the **Empty** flag described above is *false*, this field contains a xxx. If the **Empty** flag described above is *true*, this field will be *zero* (0).

5.2.2 Configuration

The structure for xxx is illustrated in Figure 36:

Figure 36 BSI Configuration

Where:

XXX: TBD.

XXX: TBD.

XXX: TBD.

5.3 Interrupt Sources

TBD

Appendix A
Utility Free-list Interface

This interface is reserved for the implementation of the *Socket-Abstraction-Services* (see xxx). However, it is documented here for completeness. The interface contains *four* (4) *Free-lists*. Where a free-list is simply a 512-entry FIFO whose read and write ports are connected to a single *read-write* I/O register located in the processor's GPO AXI space. A block diagram of one of four free-lists is illustrated in Figure A.1 below:

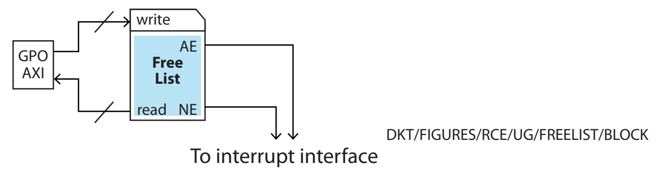

Figure A.1 The utility Free-list

A.1 Registers

The interface defines *four* (4) free-lists. Each free-list is accessed through *two* (2) registers located in the processor's GPO AXI space. Appendix B specifies the base address for the set of eight registers. Table A.1 on page 76 enumerates their relative address with respect to their base.

Table A.1 Utility Free-list Register Map

Offset[1]	Description	Access	See:
`00`	Allocate from Utility Free-List$_0$	*Read/Only*	A.1.1
`04`	Deallocate to Utility Free-List$_0$	*Write/Only*	A.1.2
`08`	Allocate from Utility Free-List$_1$	*Read/Only*	A.1.1
`12`	Deallocate to Utility Free-List$_1$	*Write/Only*	A.1.2
`16`	Allocate from Utility Free-List$_2$	*Read/Only*	A.1.1
`20`	Deallocate to Utility Free-List$_2$	*Write/Only*	A.1.2
`24`	Allocate from Utility Free-List$_3$	*Read/Only*	A.1.1
`28`	Deallocate to Utility Free-List$_3$	*Write/Only*	A.1.2

1. In (decimal) *bytes*

A.1.1 Allocate from Utility Free-List

The interface for allocating from a socket's *utility free-list* is a single, *read/only* I/O register bound to the read port of the FIFO described in Figure A.1. Access to this interface is through a *Load* instruction whose target is the physical address of this register. Table A.1 on page 76 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a FIFO mapped to a read/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 4.1.

If, when read, the free-list was *not* empty, one entry is removed and its value returned. That value will always have its *low-order* bit *clear* (0). Conversely, if, when read, the free-list *was* empty, the *low-order* bit of the returned value is *set* (1). The structure of a returned value is illustrated in Figure A.2 below:

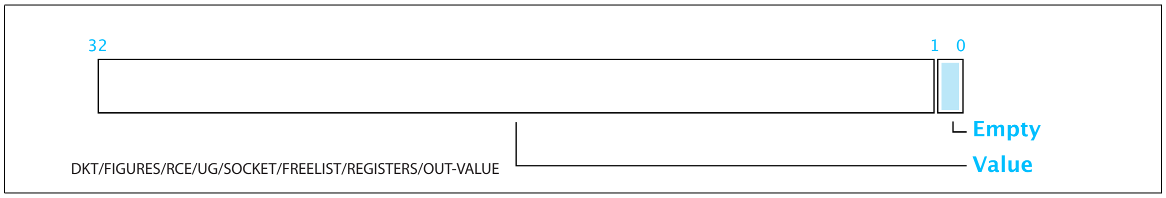

Figure A.2 Value removed from utility free-list.

Where:

Empty: This field is a single bit flag indicating whether or not the list was *empty* when read. If the flag is *false* (*clear*), the list was *not* empty and the **Value** field described below is valid. If this flag is *true* (*set*) the list *was* empty. In such a eventuality the **Value** field will be *zero* (0).

Value: If the **Empty** flag described above is *false*, this field contains the returned value r If the **Empty** flag described above is *true*, this field will be *zero* (0).

A.1.2 Return to Utility Free-List

The interface for de-allocation to a socket's *utility free-list* is a single, *write/only* I/O register bound to the write port of the FIFO described in Figure A.1. Access to this interface is through a *Store* instruction whose target is the physical address of this register. Table A.1 on page 76 specifies the relative address of this register and using Appendix B, its absolute address. The access policy for a FIFO mapped to a write/only register is described in Section 1.4.1. Note that the register's associated FIFO is mapped to an *interrupt source* as discussed in Section 4.1.

The *low-order bit* of the value written to this register is ignored, but should set to *zero* (0) as illustrated in Figure A.3 below:

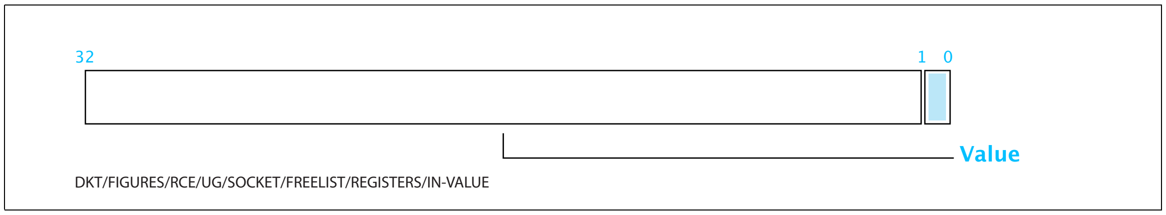

Figure A.3 Value inserted to utility free-list.

A.2 Interrupt Sources

The interface defines *four* (4) free-lists and in turn each free-list defines *two* (2) *interrupt sources*. An interrupt source is a firmware signal which, while asserted and not masked may trigger a processor interrupt. Each source is uniquely identified by its source *address*. Table 9 on page 65 specifies the *base* address for the set of eight sources. Table A.2 on page 78 enumerates the relative address of a source with respect to its base. See Chapter 4 for a discussion of the interrupt interface.

Table A.2 Free-list Interrupt Source Map

Offset	Description of signal	Associated register
0	Free-List$_0$ is *Almost-Empty*	Section A.1.1
1	Free-List$_0$ is *Not-Empty*	Section A.1.1
2	Free-List$_1$ is *Almost-Empty*	Section A.1.1
3	Free-List$_1$ is *Not-Empty*	Section A.1.1
4	Free-List$_2$ is *Almost-Empty*	Section A.1.1
5	Free-List$_2$ is *Not-Empty*	Section A.1.1
6	Free-List$_3$ is *Almost-Empty*	Section A.1.1
7	Free-List$_3$ is *Not-Empty*	Section A.1.1

Appendix B
Register Map

TBD

Table B.1 Register Map

Offset[1]	Description	Length[2]	See:
00000	Version	32	Section 2.1
00032	*Reserved*	992	*N/A*
01024	BSI Configuration	4096	Section 5.2.2
05120	Interrupt source remap table	2048	Section 4.2.1
07168	Interrupt group *Enables*	64	Section 4.2.2
07232	Interrupt group *Disables*	64	Section 4.2.3
07296	Socket_0	32	Section 2.5
07328	Socket_1	32	Section 2.5
07360	Socket_2	32	Section 2.5
07392	Socket_3	32	Section 2.5
07424	*Reserved*	256	*N/A*
07680	Utility free-lists	16	Section A.1
07696	*Reserved*	496	*N/A*
08192	Rendezvous pending vector	128	Section 3.3

Table B.1 Register Map

Offset[1]	Description	Length[2]	See:
08320	*Reserved*	380	*N/A*
08700	BSI I2C Signal	4	Section 5.2.1
08704	*Reserved*	7680	*N/A*
16384	User defined	16384	N/A

1. In (decimal) *bytes*
2. In (decimal) *bytes*