The GLAST experiment at SLAC

The Event Builder Module

Electronics group

Programming ICD specification

Document Version:	2.3
Document Issue:	2
Document Edition:	English
Document Status:	Under release control
Document ID:	LAT-TD-01546-03
Document Date:	March 5, 2005

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

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

Abstract

A conceptual design of the EBM (Event Builder Module). This module is designed to both build events from modules on the LAT event fabric and to transfer data from crate-to-crate or crate to SSR (Solid State Recorder).

Hardware compatibility

This document assumes the following hardware revisions:

EBM: Version TBD

Intended audience

This document is intended principally as a guide for the *developer* and *users* of the EBM. Users include:

- Developers of the sub-system electronics which interface with the EBM
- Developers of Flight-Software
- Developers of I&T (Integration and Test) based systems

All readers of this document are expected to be familiar with the concepts described in [1]

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 TEM).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

References

1 "LAT Inter-module Communications - A reference manual," Michael Huffer, LAT-TD-00606.

2 "GASU Based Teststands - A hardware and software Primer," Michael Huffer, LAT-TD-03664.

Note: For additional resources, refer to the LAT Electronics, DAQ Critical Design Requirements List. On the LAT Electronics, Data Acquisition & Instrument Flight Software page (http://www-glast.slac.stanford.edu/Elec_DAQ/Elec_DAQ_home.htm), click Hardware and then click List of all documents.

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	The Event Builder Module Programming ICD specification		
	Version:	2.3		
	Issue:	2		
	Edition:	English		
	ID:	LAT-TD-01546		
	Status:	Under release control		
	Created:	February 9, 2002		
	Date:	March 5, 2005		
	Access:	V:\GLAST\Electronics\Design Documents\EBM\v2.3\Frontmatter.fm		
	Keywords:	Event Builder Module EBM		
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	Michael Huffer		
	Written by:	Michael Huffer		

Table 2 Approval sheet

Name	Title	Signature	Date
Gunther Haller	LAT CHIEF ELECTRONICS ENGINEER		
JJ Russell	FLIGHT SOFTWARE LEAD		

Document Status Sheet

Table 3 Document Status Sheet

Title:	The Event Builder Module Programming ICD specification		
ID:	LAT-TD-01546		
Version	**Issue**	**Date**	**Reason for change**
1.0	1	3/03/2003	Initial draft, for internal comment
1.1	1	5/01/2003	Changed title and added verbiage to conform to both Gunther's and NASA's wishes for CDR. Incorporated various comments received from Tony and JJ.
1.2	1	11/13/2003	Changed title and added verbiage to conform to both Gunther's and NASA's wishes for CDR. Incorporated various comments received from Tony and JJ.
1.3	1	3/05/2004	Updated fonts.
2.0	1	5/07/2004	This version reflects the "as built" EBM. As the changes in the register interface are changes that are too numerous to enumerate, it's best to just start at scratch...
2.0	4	5/17/2004	Corrected typos. Sent document for sign-off.
2.0	5	7/14/04	Updated references and PDF TOC.
2.1	1	8/23/04	
2.2	1	2/2/05	For some, as yet undefined reason, I had the enable, contributors, and destination registers all screwed up. The hardware has been this way for ages. It has, I believe, no affect on software, as it seems everyone wisely ignored this document and coded to what was actually built. Three other changes: First, in all the enable registers, the GEM bit offset indicates *Read-Only*. This field is actually read/writeable. Second, the maximum value of the contribution time-out was doubled in the hardware. Third, the EBM contains "ghost" registers, which are now documented. These registers are reserved and are only documented, such that accessing a non-existent register now has the proper behaviour.
2.3	1	3/4/05	Added new register to allow (conditionally) inserting silly synch pattern word in all data sent to the spacecraft's SSR interface. (See Section 2.3.1 and Section 2.3.11).
2.3	2	3/5/05	Eric points out that my explanation of the new register (see above) is incorrect (not to mention some typos).

Table of Contents

List of Figures

List of Tables

Chapter 1
Principles of operation

1.1 Overview

The principal function of the EBM (Event Builder Module) is to receive LATp packets from up to twenty-four different *sources* and transfer these packets to as many as seven different *destinations*. These potential sources include:

- 1 GEM (GLT Electronics Module)
- 16 TEMs (Tower Electronic Module)
- 1 AEM (ACD Electronics module)
- 3 EPU (Event Processing Units) crates
- 3 SIU (Spacecraft Interface Units) crates

The potential *destinations* include:

- 3 EPU crates
- 3 SIU crates
- 1 SSR (Solid State Recorder)[1]

Note that crates are *both* a source and a destination to the EBM. Logically, the EBM consists primarily of 24 *Source Processors* (SP), with each processor assigned to handle the input from one source. Processors are numbered arbitrarily from *zero* (0) to *twenty-three* (23), and although somewhat irrelevant to operation of the EBM, in the interests of clarity, when using the EBM, one should maintain the mapping between LATp source address and SP number as enumerated in Figure 1.

Source Processors receive and buffer LATp packets for later processing by the *scheduler*. The function of the scheduler is to transfer packets buffered in the EBM's processors and transmit

1. Ignoring for the moment SSR redundancy. (See Section 2.3.1.)

these packets (potentially transformed) to one or more of the EBM's destinations. The scheduler is responsible for two fundamental activities:

- Event Building
- Packet Transfer (either from crate-to-crate or from crate-to-SSR)

These activities require the scheduler to partition the EBM's processors into two arbitrary sets:

Processors numbered from 0 - 17: The scheduler assumes any packets buffered in these processors will be used to *build events*. Packets buffered in such processors are called an event's *contributions*. The source of such packets are called the event's *contributors*. Thus, an event consists of contributions from the GEM, all the TEMs, and the AEM.

Processors numbered from 18 - 23: The scheduler assumes any packets buffered in these processors originated in a crate and are destined to be delivered to either a crate or the SSR. Packets buffered in such processors are called *data*. The source of such packets are simply called *data sources*.

Note that while Source Processors receive packets asynchronously with respect to each other, the scheduler is only capable of processing *one* activity at a time. Therefore, a significant fraction of the scheduler's implementation is taken up not only with the mechanics of transmitting packets, but also with the issue of efficiently scheduling potentially overlapping activities. A block diagram expressing the components of the EBM is illustrated in Figure 1.

The EBM requires a certain amount of configuration and maintains various statistics concerning its operations. To access this functionality, the EBM has a set of registers which are interfaced to the user through the LAT common Command/Response protocol. (See [1].) An enumeration and discussion of these registers can be found in Chapter 2. The protocol definition for commanding these registers through the Command/Response protocol is found in Chapter 3.

Physically, the EBM resides on single Printed Circuit Board (PCB) and consists of three FPGAs to handle the combinatorial logic, eight memories used for Source Processor input buffering, and a miscellaneousness number of commercial LVDS transmitters/receivers to process control, input, and output signals. These components share the same circuit board as the CRU (Command/Response Unit), the AEM, and the GEM. This PCB is called the DAQ board. (See [2].) In order to satisfy redundancy requirements each LAT contains *two* DAQ boards, both identical and both residing in the same GASU *box*.

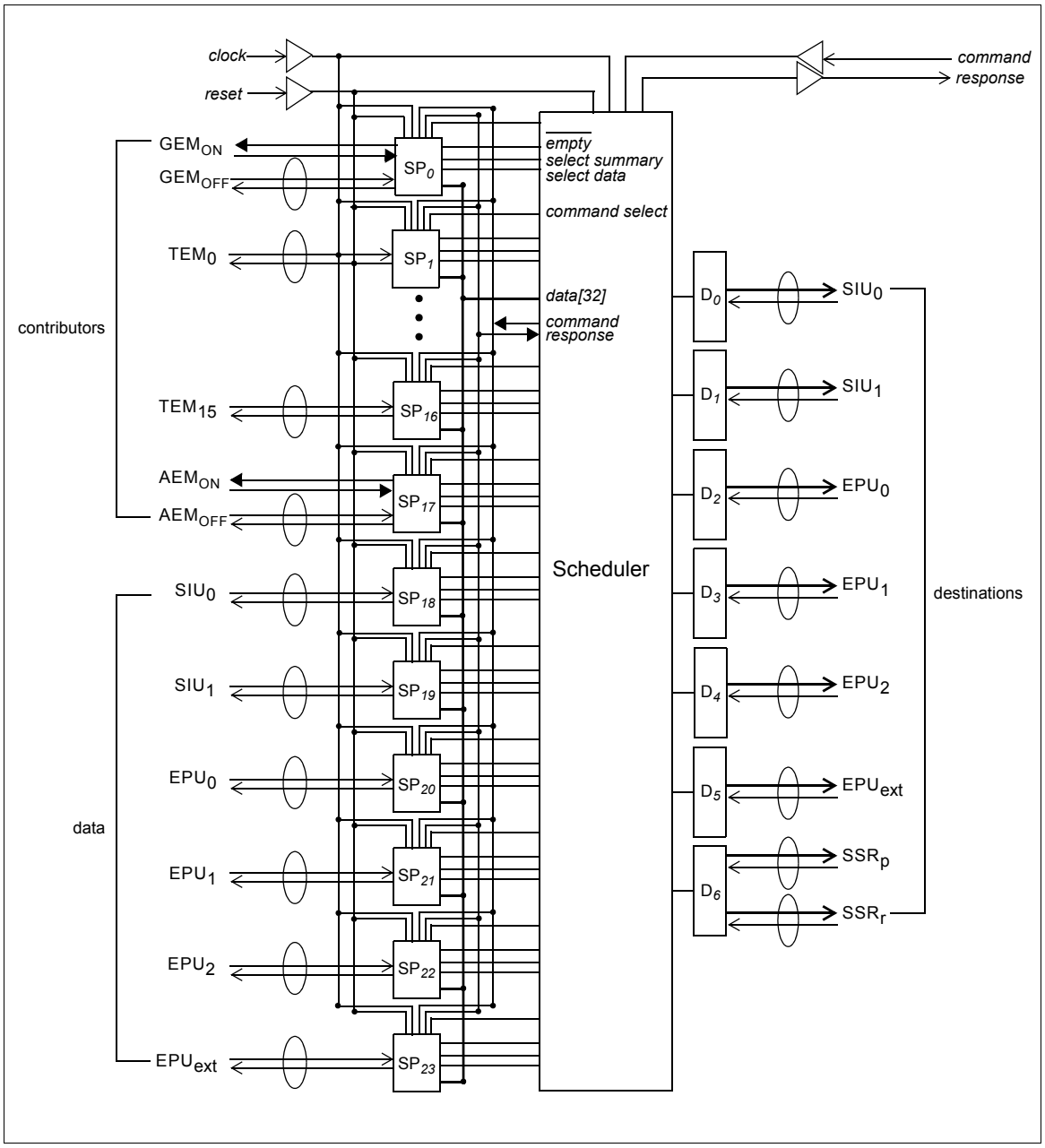

Figure 1 Block diagram of the EBM

1.2 The Source Processor

While each of the 24 source processors operate autonomously with respect to each other, they are identical in their implementation. The function of each processor is to receive and buffer

incoming LATp packets. These buffered packets are then made available to the *scheduler* (discussed in Section 1.3) in order to either event build or distribute to a specified destination. The bulk of the source processor is taken up by two FIFOs: one to buffer data from deserialized LATp packets and the other to summarize for each incoming packet the results of processing that packet. In addition to its FIFOs, a source processor contains the logic to manage deserialization, a statistics register, a Command/Response interface in order to access this register, and a series of interface signals used by the scheduler. This is all illustrated below in Figure 2:

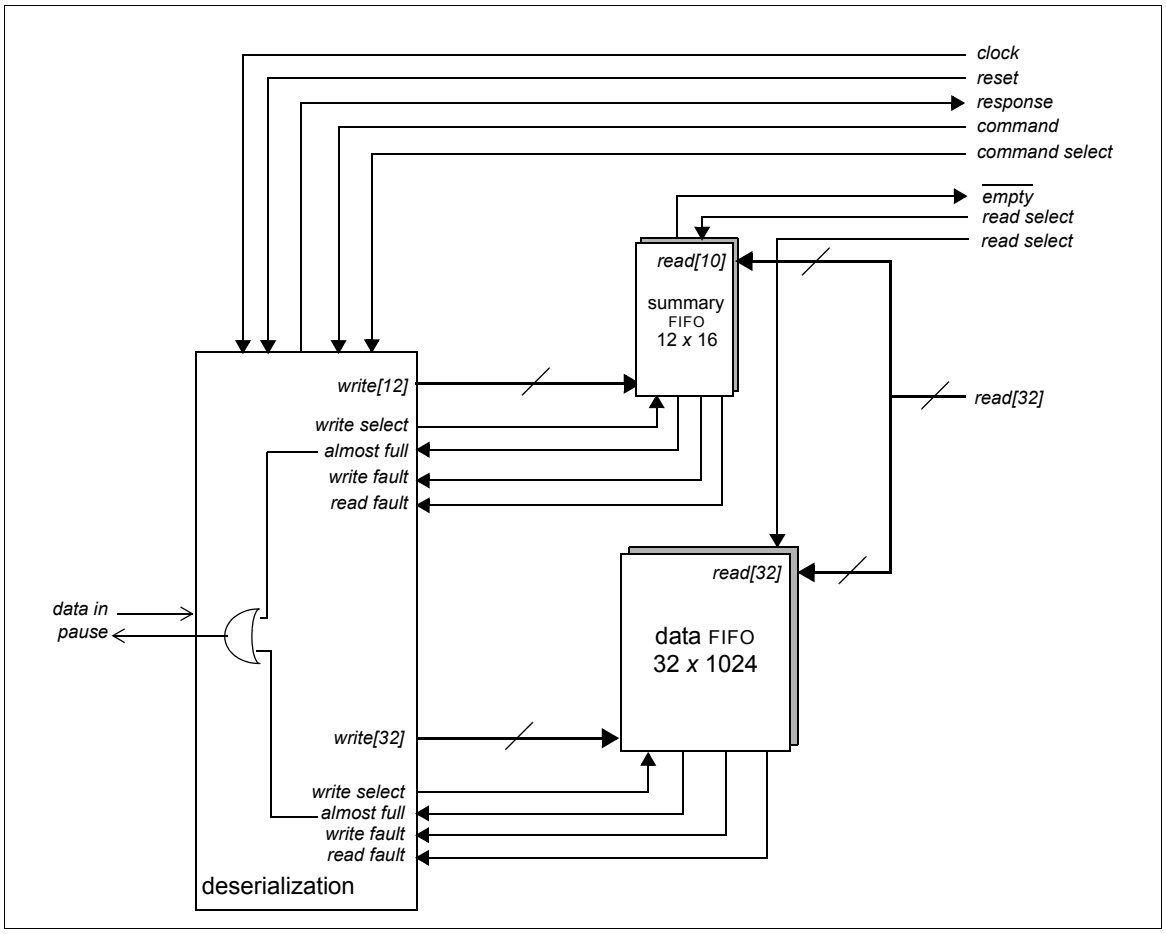

Figure 2 Block diagram of a Source Processor

The Source Processor can be represented abstractly as a two-state FSM (Finite State Machine), which at any one time is in one of the following two states:

idle: Waiting for a *start* bit from its incoming LATp data line. It will accept incoming packets only if its data FIFO has sufficient space for at least one cell and there is at least one free entry in its summary FIFO.

active: Deserialized a LATp packet. When the processor finishes deserialized one packet, it returns to its idle state.

While active, deserialization of a packet involves the following steps:

i. Initialize counters to keep track of the length of the incoming packet and tally any processing errors.

ii. Deserialize first 32-bit word. Compute header parity. If parity is correct, insert word on data FIFO and increment counter.

iii. Continue to deserialize, stripping delineator and parity information and writing subsequent words to the data FIFO. Increment counter, checking data parity on incoming cells and saving any processing errors.

iv. When packet is complete, fabricate a summary word which includes the length of the packet and an error summary.

v. Write the summary word to the summary FIFO.

vi. Update the appropriate receiver statistic. (See Section 2.4.1.)

When the processor returns to idle, the packet will be buffered within the data FIFO and its length (and processing status) will be contained within the summary FIFO. In short:

- The number of entries on the summary FIFO will be equal to the number of packets pending in the source processor.
- The $\overline{\textit{empty}}$ signal specifies whether the source processor has one or more pending packets.
- The packets themselves are contained within the data FIFO.

This is all illustrated in Figure 3:

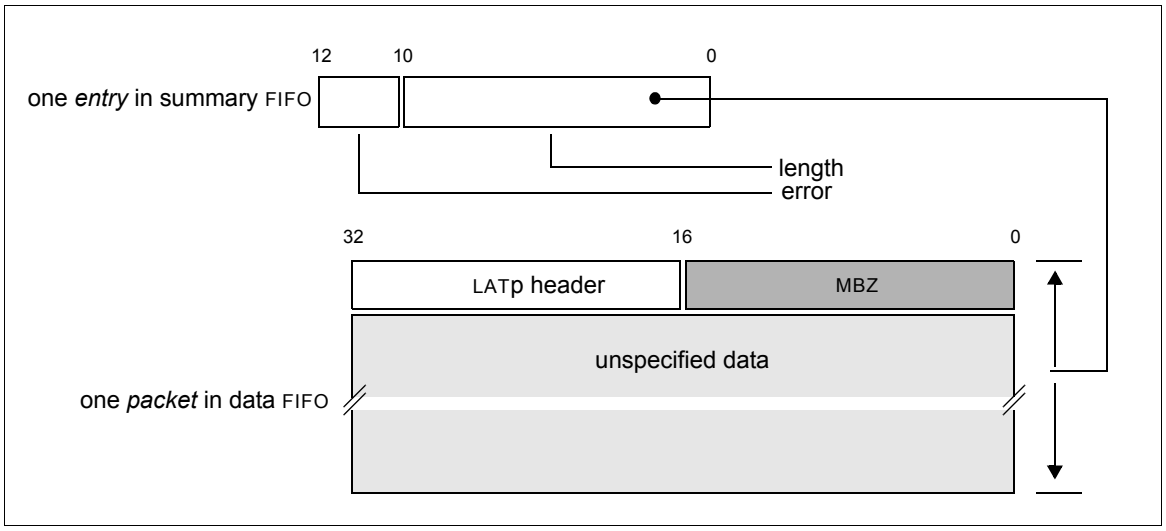

Figure 3 Relationship between deserialized packet in data FIFO and its corresponding summary entry

Note that the LATp header and the 16-bit MBZ value immediately following the header are simply the first word of the control cell of the received packet. The fields of a summary entry are defined as follows:

length: The length of the received packet in units of 32-bit words. If the ***error*** field is non-zero, the data contents are not to be trusted. LATp subjects incoming packets to two constraints: they are quantized in units of 128-bit *cells*, and any one packet is at least one cell long. However, this does not necessarily imply that this field is always non-zero or that its low-order two bits are zero. This is true only if the ***error*** field is *not* zero. Specifically, as the data FIFO is written on word boundaries, a `WRITE_FAULT` could violate any one of these two constraints.

error: An enumeration of the errors either computed or encountered in the processing of the packet. If the field has a value of *zero*, the packet was processed successfully. The possible values for this field are enumerated in Table 4.

Table 4 Error values found in the summary FIFO

Error	Value[1]	Reason
`NONE`	`00`	Success
`PARITY`	`01`	Parity incorrect for one or more data cells[2]
`TRUNCATED`	`10`	Incoming packet would not fit in the space remaining in the data FIFO
`WRITE_FAULT`	`11`	Data FIFO was written when full

1. In binary
2. Header parity errors do *not* cause a packet to be emitted; however, they are tallied in the statistics register.

1.3 Scheduler

The function of the scheduler is two-fold:

- Transfer packets stored in one of the EBM's six *data* Source Processors to one of the EBM's seven possible destinations.
- Build and transfer events sent by the EBM's eighteen possible contributors to one of the EBM's seven possible destinations. For each contributor, the packets from this contributor are buffered in a *contributor* Source Processor.

This corresponds to seven different kinds of potentially simultaneous activity on the part of the scheduler. However, the scheduler only processes *one* activity at a time. Consequently, one of the scheduler's basic functions is to assign priority to these activities and decide which activity to process when. It is convenient to express the function of the scheduler as a set of four different inter-related Finite State Machines (FSM) as shown in Figure 4. Each engine is discussed in detail in the following sections.

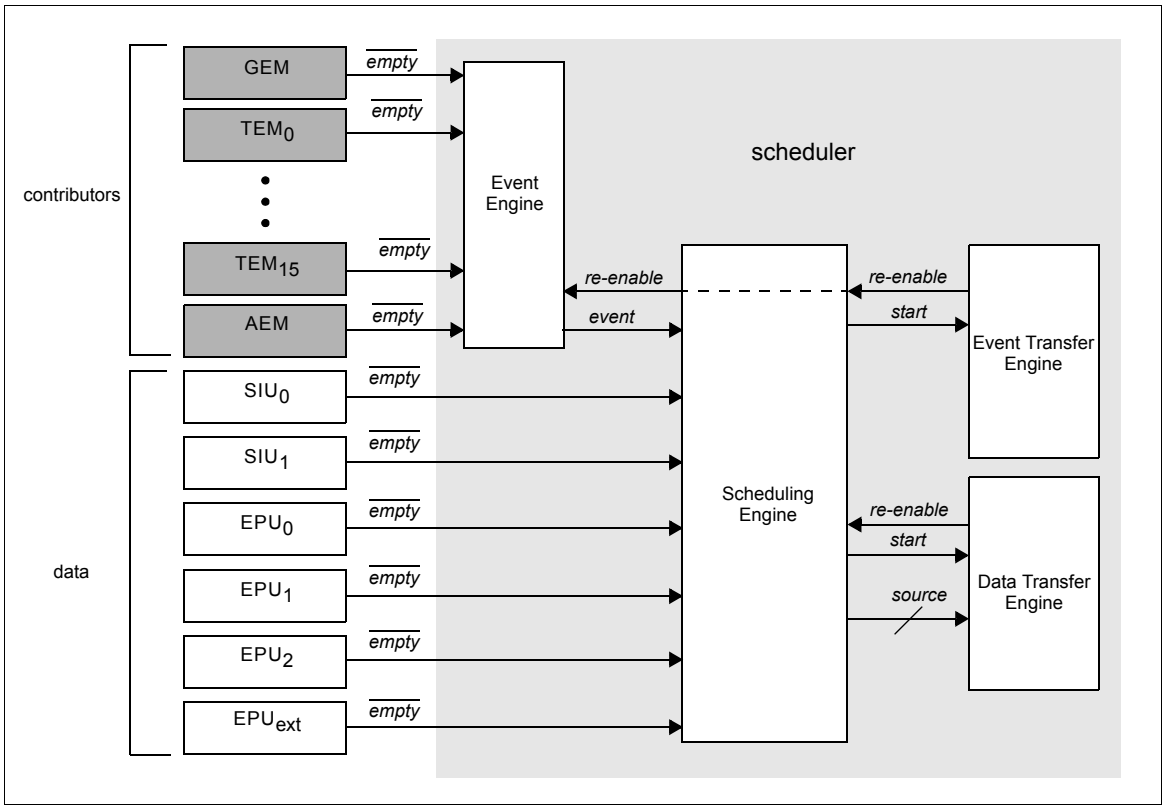

Figure 4 Block diagram of the scheduler

1.3.1 The Scheduling Engine

The function of the Scheduling Engine is to determine the highest priority request from up to seven different requesters and start up a transfer engine to satisfy this request. Abstractly, its function can be represented by the FSM illustrated in Figure 5. At any point in time the engine is in one of three states:

idle: Waiting for transfer requests from up to seven different requesters. This is specified by either the assertion of the ***event*** signal from the *Event Engine* (discussed in Section 1.3.3) or the assertion of an ***empty*** signal from one or more of the six *data* Source Processors (discussed in Section 1.2).

active: Determining the highest priority requester (shown in Table 5) and then waking up the appropriate transfer engine (discussed in sections 1.3.2 and 1.3.4) by asserting its corresponding *start* signal.

pending: Waiting for a transfer engine to complete its work. This is specified by the assertion of the *re-enable* signal from one of the two transfer engines.

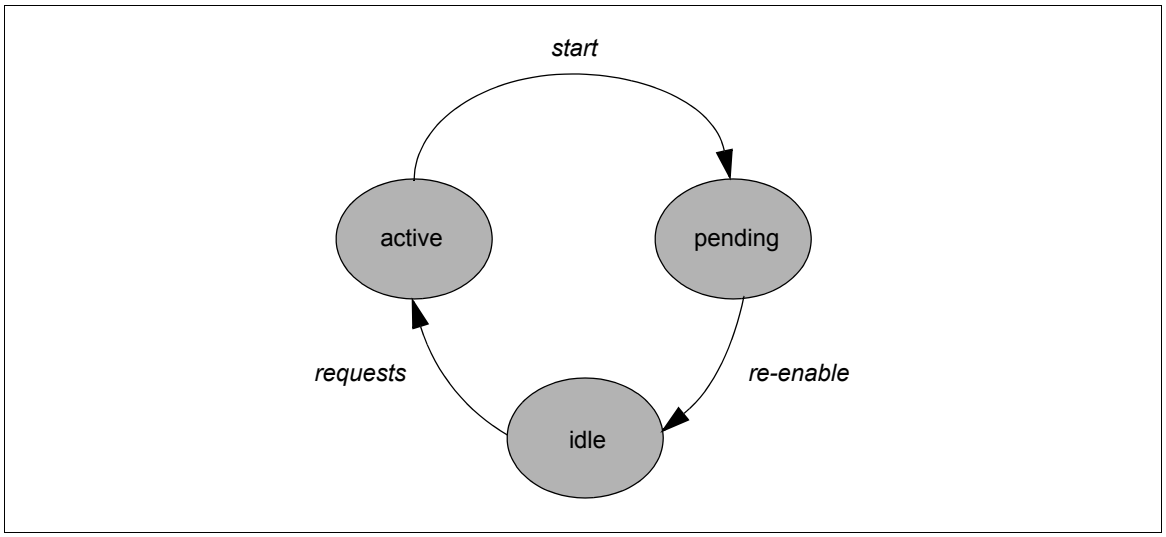

Figure 5 Scheduling engine states

As noted above, each of the requesters operate asynchronously with respect to each other and only one transfer engine is active at a time. Consequently, it is possible to have simultaneous outstanding requests. Each requester is therefore assigned a priority, and it is this priority which determines the order in which requests are serviced. This assignment is enumerated in Table 5:

Table 5 Scheduler activity priority

Activity	Priority[1]
SIU_{EXT} *not* empty	0
SIU_1 *not* empty	1
SIU_2 *not* empty	2
EPU_0 *not* empty	3
EPU_1 *not* empty	4
EPU_2 *not* empty	5
Event	6

1. Smaller values are higher priorities.

1.3.2 The Data Transfer Engine

The function of the *Data Transfer Engine* is to transfer a pending packet stored in one of the EBM's six Source Processors (discussed in Section 1.2), used by the scheduler as data sources, to one of the EBM's seven possible destinations. *Which* processor contains the packet to be transferred is determined by the *Scheduling Engine.* (See Section 1.3.1.) The determination of

the packet's final destination is discussed in Section 1.3.5. The Data Transfer Engine can be represented abstractly as a two-state FSM, which at any one time is in one of the following two states:

idle: Waiting for a *start* signal from the Scheduling Engine.

active: Transferring a packet from a specified Source Processor to one of the EBM's seven possible destinations. The exact processor is specified by the *source* signals between the scheduling and transfer engines. (See Figure 4.) When the engine finishes packet transfer, it asserts the *re-enable* signal back to the Scheduling Engine and returns to its idle state.

While active, the transfer of the packet involves the following steps:

i. Remove an entry from the summary FIFO of the appropriate Source Processor. This entry will specify how many words are to be removed from the processor's data FIFO.

ii. Remove a word from the data FIFO of the appropriate Source Processor. This word will contain the input packet's LATp header. Using the destination address in this header along with the algorithm described in Section 1.3.5, determine the destination of the output packet.

iii. Using the previously removed word from the data FIFO along with the summary entry, fabricate the first 32-bit word of the packet to be transferred.

iv. Transmit the packet. The first word of the first cell is determined by the previous step. Subsequent words of both control and data cells are simply copied from the source processor's data FIFO. This continues until the count is exhausted. If the count is not an even number of cells, the engine will pad the output packet with words consisting of *zero*.

v. Update the appropriate transmitter statistic. (See Section 2.4.2.)

This process and the packet resulting from the transfer is illustrated in Figure 6:

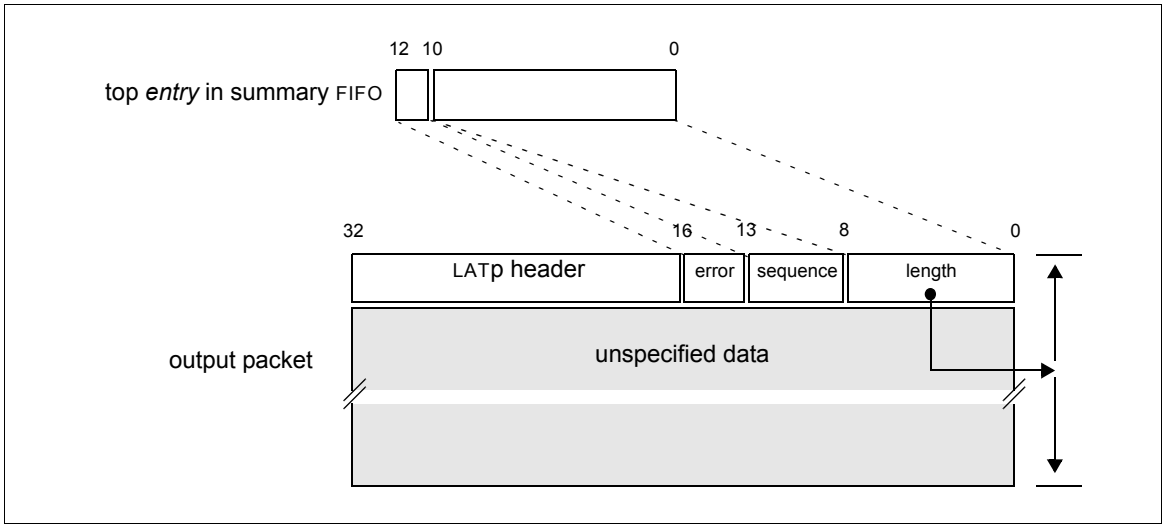

Figure 6 Transferring a data packet from a Source Processor

Where:

length: Is the size of the packet in *cells*. A cell is *four* 32-bit words long. The value of this field is derived by the entry removed from the appropriate source processor's summary FIFO.

sequence: Used when data or events span packets. The usage of this field is discussed in Section 1.3.6.

error: Specifies whether the transfer of the packet *into* the source processor failed. If the transfer succeeded, this field has a value of *zero*. If this field is non-zero, the transfer failed and the value of this field enumerates the reason why the transfer failed. These reasons are enumerated in Table 6.

LATp header: A copy of the LATp header of the packet transferred *into* the EBM.

In short, the first 32-bit word of the output packet is a copy of the first word of the corresponding packet input into the EBM, with the exception that the low-order, unspecified 16-bits which were MBZ (discussed in Section 1.2) have now been replaced with the ***length***, ***sequence*** and ***error*** fields, all of which were derived from the summary FIFO entry for the packet.

Table 6 Error values found in error field of transferred packet

Error	Value[1]	Reason
NONE	000	Success
PARITY	001	Parity incorrect for one or more data cells of incoming packet
TRUNCATED	010	Incoming packet did fit in the space remaining in source data FIFO
WRITE_FAULT	011	Source Data FIFO was written when full
N/A	100	Not applicable in a data transfer

1. In binary

1.3.2.1 Transferring to the SSR

To be written.

1.3.3 The Event Engine

The function of the *Event Engine* is to summarize event build requests from up to eighteen different event contributors. This summary is passed as one of the seven possible requests for activity to the Scheduling Engine (discussed in Section 1.3.1). Note that this engine is *not* responsible for the transfer of a built event to its destination. This is the responsibility of the *Event Transfer Engine* described in Section 1.3.4. Instead, this engine determines *when* a request for such a transfer should take place. Contributors to a built event store their packet contributions in a corresponding Source Processor. There are eighteen processors reserved for

this purpose. However, at any point in time only a subset of these processors may be enabled. This subset is called the EBM's *current* contributors. It is assumed by this document that, whenever either the Event or the Event Transfer engines must process information related to a processor, that processor is a current contributor. The set of current contributors is determined by the configuration register described in Section 2.3.1. Abstractly, the function of the Event Engine can be represented by the FSM illustrated in Figure 7. At any point in time, the engine is in one of three states:

idle: Waiting for build requests from up to eighteen different requesters. This is specified by the assertion of an $\overline{\textbf{\textit{empty}}}$ signal from one or more of the eighteen *Contribution* Source Processors (discussed in Section 1.2) suitably masked against the list of current contributors.

active: Wait either for all the current contributors to have at least one contribution or for the event time-out clock to expire. When so, assert *event* signal to the Scheduling Engine. (See 1.3.2.) This state is described in more detail below.

pending: Waiting for the Event Transfer Engine (discussed in Section 1.3.4) to complete its work. This is specified by the assertion of the *re-enable* signal from the transfer engine.

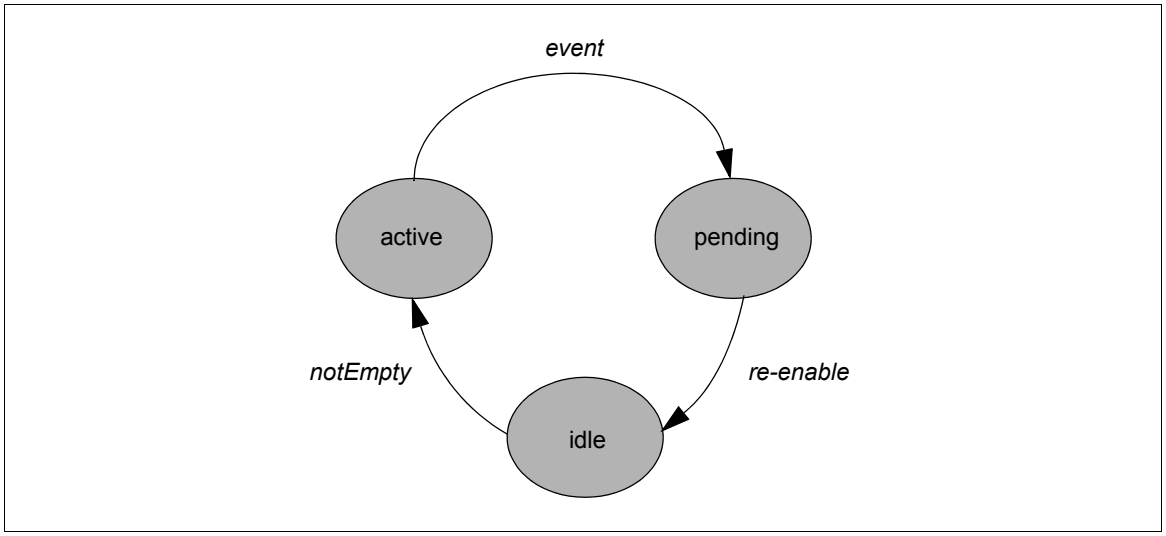

Figure 7 Event state machine

Naively, one might assume the implementation of the active state is nothing more then the AND of the empty signals from each of the eighteen contributions (suitably masked against the current contributor list). However, this does not take into account the fact that the delivery of any one contribution is not reliable and so it is necessary to *time-out* contributions, if data-flow is not to stall. Therefore, this engine includes an event time-out mechanism. The time-out algorithm is based on the idea that as long as an event build process is making "good" progress, it will continue to wait for contributions before timing out. Good progress is defined as the prompt delivery of any one contribution *after* the arrival of a previous contribution. The definition of prompt delivery is a configuration register of the EBM. (This register is described in Section 2.3.1.) If a contribution does not arrive in this specified time (relative to the last

arrived contribution), the Event Engine will time-out, causing the assertion of the *event* signal. Listing 1 presents some pseudo-code to express the activity of the engine while in the active state.

Listing 1 Action on “notEmpty” transition (active state)

```
 1:  onNotEmpty()
 2:    current_Contributors = *SOURCE_ENABLES_REGISTER & 0x3FFFF;
 3:    timeout              = *TIMEOUT_REGISTER;
 4:    clock                = timeout;
 5:    previous             = Current_Contributors;
 6:    while(clock) {
 7:       current = previous & ~Current_Contributors;
 8:       if(!current) break;
 9:       if(current ^ previous)
10:          clock = timeout;
11:       else
12:          clock--;
13:       previous = current;}
14:    eventSignal = TRUE;
15:    return TO_PENDING;
```

1.3.4 The Event Transfer Engine

The function of the *Event Transfer Engine* is to transfer one or more pending packets stored in the EBM’s eighteen Source Processors (discussed in Section 1.2) used for Event Building, to one of the EBM’s seven possible destinations. The determination of the event’s final destination is discussed in Section 1.3.5. Abstractly, the Event Transfer Engine can be represented as a two-state FSM, which at any one time is in one of the following two states:

idle: Waiting for a *start* signal from the Scheduling Engine.

active: Transferring an event from a set of specified Source Processors to one of the EBM’s seven possible destinations. When the engine finishes event transfer, it asserts the *re-enable* signal back to the Scheduling Engine and returns to its idle state.

While active, the transfer of an event involves the following steps:

i. Sample the contributor $\overline{\textbf{\textit{empty}}}$ lines. Mask the result with the current contributors list. This value is called the ***event contributor list***. This list specifies the presence or absence of packets buffered in the EBM’s source processors for a particular event. If the bit at a specified offset is *set*, the corresponding source processor contains a packet to be contributed to the event. (A contribution is *present*.) If the bit is *clear*, the source processor does not have a packet to contribute to the event. (The contribution is *absent*.)

ii. Process the event using the current contributor list. This list is processed in *ascending* Source Processor order. Using this list and the event contributor list, find the first present contribution. As the GEM is always enabled and its contributions arrive on SP_0, it should always be the first present contribution. If it is not, see Section 1.3.4.3.

Get the first word of this contribution. This word will contain the input packet's LATp header. Using the destination address in this header, along with the algorithm described in Section 1.3.5, determine the destination of the event. Assuming the first valid contribution is the GEM, process the contribution as described in Section 1.3.4.1.

iii. Using the current contributor list, continue to process each contributor in the event contributor list until the list is exhausted. If a contributor's packet contribution is *present*, process the contribution as described in Section 1.3.4.1. If the contributor's packets contribution is absent, the contribution is considered *timed-out* and it is processed as described in Section 1.3.4.2.

iv. Update the appropriate transmitter statistics. (See Section 2.4.2.)

In this fashion, each contribution is simply appended to the previous contribution. The final result on the "output wire" appears to its receiver as *one* packet. The LATp header for the packet always corresponds to the LATp header of the first *present* contribution (which should be the GEM, as it is always enabled). As the transfer engine is not started unless at least one source processor had a packet, the minimum number of contributions for any one event is *one*. The total number of contributions for any one event is always equal to the number of enabled contributors as specified by the current contributors list. (See Section 2.3.6.) For example, if the current contributor list specifies TEM_2, TEM_7, and the AEM, a built event would contain *four* contributions in the order: GEM, TEM_2, TEM_7, and the AEM. This is all illustrated in Figure 8:

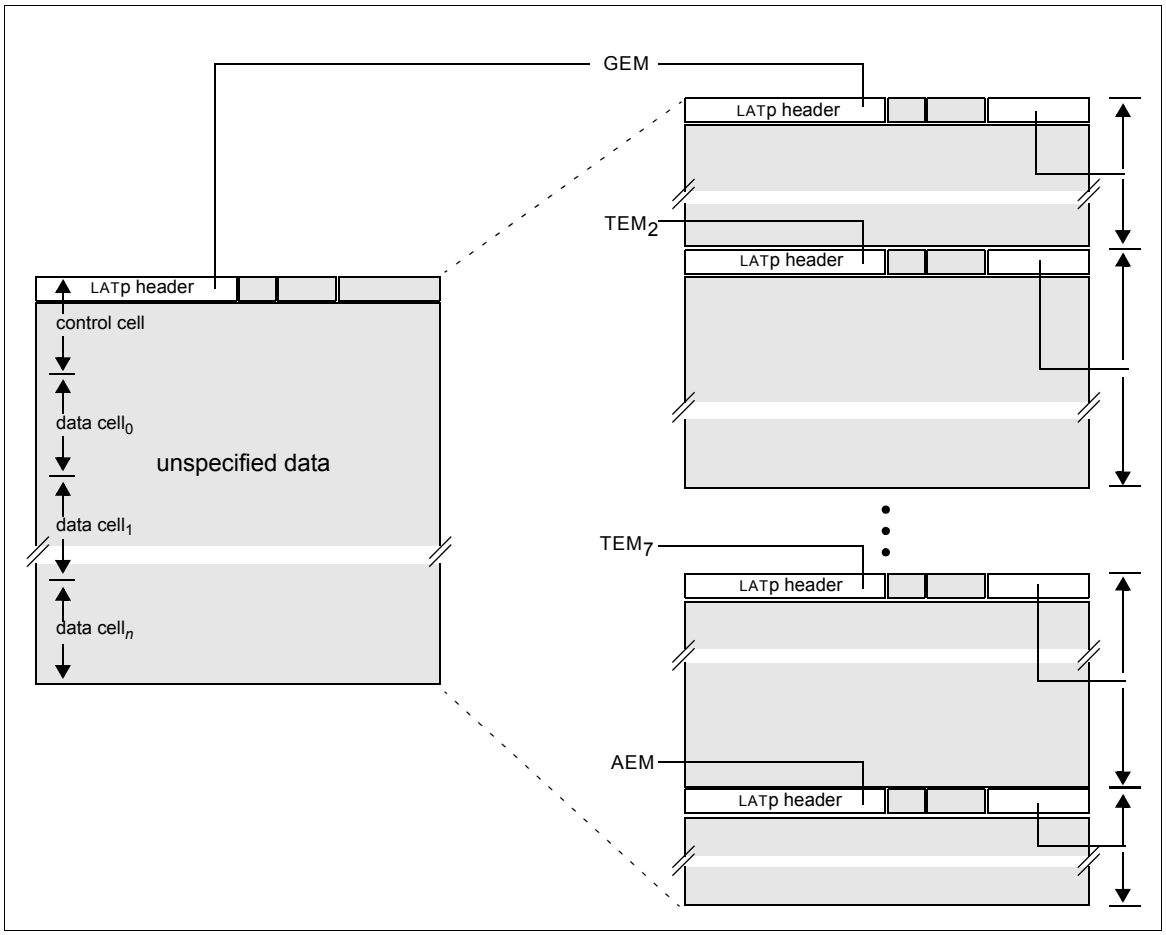

Figure 8 Structure of a built event

1.3.4.1 Processing a present contribution

The transfer of any one contribution involves the following steps:

i. Remove an entry from the summary FIFO of the appropriate Source Processor. This entry will specify how many words are to be removed from the processor's data FIFO.

ii. Remove the first word from the data FIFO of the appropriate Source Processor. This word will contain the input packet's LATp header. Using this word along with the summary entry, fabricate the first 32-bit word of the contribution to be transferred.

iii. Transmit the contribution. The first word transmitted is determined by the previous step. Subsequent words (whether of control or data cells) are simply copied from the source processor's data FIFO. This continues until the count is exhausted. If the count is not an even number of cells, the engine will pad the output packet with words consisting of *zero*.

This process and the contribution resulting from the transfer is illustrated in Figure 9:

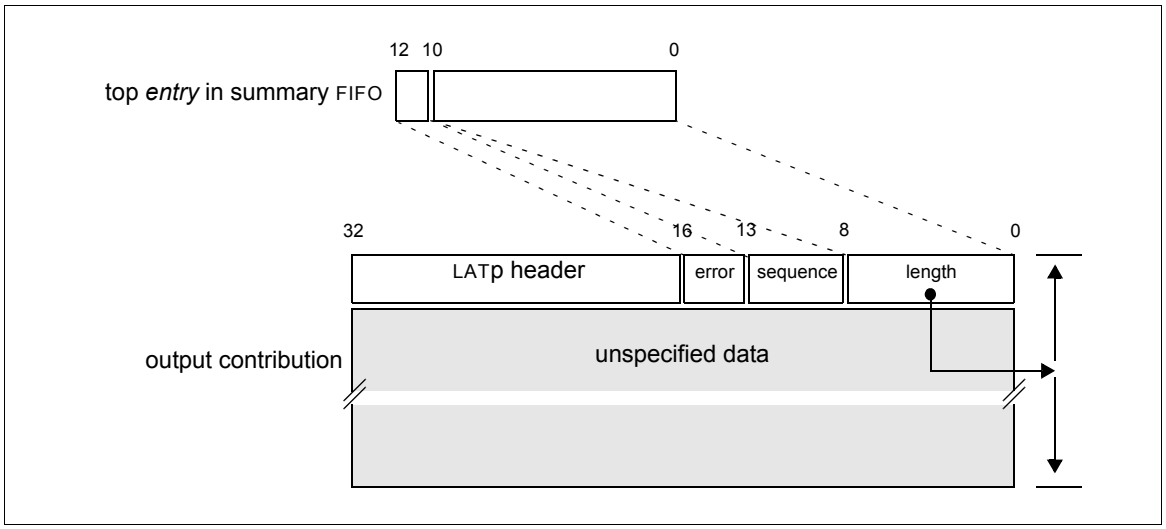

Figure 9 Transferring a contribution packet from a Source Processor

Where:

length: Is the size of the contribution in *cells*. A cell is *four* 32-bit words long. The value of this field is derived by the entry removed from the appropriate source processor's summary FIFO.

sequence: Used when data or events span packets. The usage of this field is discussed in Section 1.3.6. This field will always be *zero* if this is not the first contribution to the event.

error: Specifies whether the transfer of the contribution *into* the source processor failed. If the transfer succeeded, this field has a value of *zero*. If this field is non-zero, the transfer failed and the value of this field enumerates the reason why the transfer failed. These reasons are enumerated in Table 7.

LATp header: A copy of the LATp header of the contribution transferred *into* the EBM.

In short, the first 32-bit word of the contribution is a copy of the first word of the corresponding packet input into the EBM, with the exception that the low-order, unspecified 16-bits which were MBZ (discussed in Section 1.2) have now been replaced with the ***length***, ***sequence*** and ***error*** fields, all of which were derived from the summary FIFO entry for the packet.

1.3.4.2 Processing an absent contribution

An absent contribution will simply appear in the output packet as a fixed-size contribution as illustrated in Figure 9:

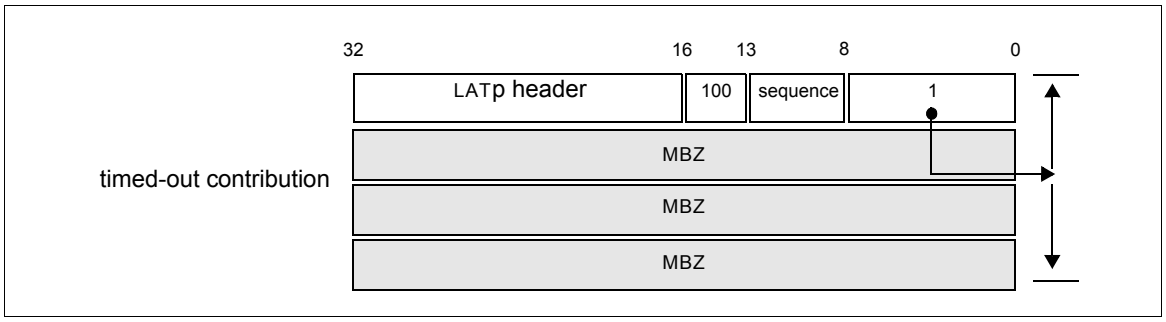

Figure 10 Transferring a timed-out contribution

Where:

length: Is the size of the contribution in *cells*. A cell is *four* 32-bit words long. A timed-out cell has a length of *one* (1). The data following the header has all bits set to *zero*.

sequence: Used when data or events span packets. The usage of this field is discussed in Section 1.3.6. This field will always be *zero* if this is not the first contribution to the event.

error: Will always have the value `TIMEDOUT` as specified in Table 7.

LATp header: A copy of the LATp header of the *first* present contribution of the event with the exception of its ***source address*** field. This field will have a value corresponding to the number of the Source Processor whose contribution was not present.

Table 7 Error values found in error field of a transferred contribution

Error	Value[1]	Reason
`NONE`	`000`	Success
`PARITY`	`001`	Parity incorrect for one or more data cells of incoming packet
`TRUNCATED`	`010`	Incoming packet did fit in the space remaining in source data FIFO
`WRITE_FAULT`	`011`	Source Data FIFO was written when full
`TIMEDOUT`	`100`	Contribution should be, but is was not present when building event

1. In binary

1.3.4.3 Timing out the GEM contribution

To be written.

1.3.5 Determining an output packet's destination

The scheduler uses as input the ***destination address*** field (discussed in [1]) of the pertinent LATp header. (See sections 1.3.2 and 1.3.4.) Depending on the value of this field, the destination of the packet is enumerated in Table 8:

Table 8 Effect of the LATp destination field on packet destination

Field value[1]	Packet destination
0	Round-Robin. Alternate to *all* destinations enabled for round-robin
1	Send to SIU_{ext}
2	Send to SIU_0
3	Send to SIU_1
4	Send to EPU_0
5	Send to EPU_1
6	Send to EPU_2
7	Send to SSR
8 - 1E	Reserved (discard)
1F	Broadcast address. Send to *all* destinations enabled for broadcast

1. Hexadecimal

There are two interesting cases in this table:

Round-robin: The source of the packet has deferred destination assignment to the EBM. The EBM maintains a internal register which defines the "next" round-robin assignment. The list of destinations which participate in the round-robin is defined by a configuration register. (See Figure 18.) Once the packet has been transmitted the EBM "increments" its next pointer. *Note: Need to determine what affect flow-control should have on this decision.*

Broadcast: The source of the packet intends that the packet should be sent to the *all* destinations supported by the EBM. The definition of *all* is specified by a configuration register. (See Figure 18.) In this case, flow-control is the OR of all enabled destinations. *Note: This needs to be thought about some more.*

1.3.6 Transmitting and flow-control

To be written.

Chapter 2
Registers

2.1 Introduction

The EBM contains a series of *registers* for configuration and management. These registers are divided into two groups of related functionality:

- EBM Control
- Transmit/Receive Statistics

Within each group, registers have a common length to allow their access through broadcast operations. The access model for these registers is through the LAT common Command/Response Protocol. This protocol is specified generically in [1]. The specific implementation of this protocol for the EBM is described in Chapter 3.

2.2 Conventions

Certain conventions apply to the fields within a register. These conventions fit into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated *grayed out*. An MBZ field will:

— Read back as zero

— Ignore writes

— Reset to zero

Read/Write: A *Reset* will set a read/write field to zero.

Read-only: Read-only fields are illustrated *lightly* grayed-out along with their intended value. Any *read-only* field will:

— Ignore writes

— Reset to zero, unless otherwise documented

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

2.3 Controller registers

This section incorporates all the registers whose configuration has a global affect over all the functional blocks of the EBM.

Table 9 The registers of the controller block

Name	Address	Access	Description
BACK_END_CONFIG	00	R/W	Back-End configuration and setup
FRONT_END_A_CONFIG	01	R/W	First Back-End configuration
FRONT_END_B_CONFIG	02	R/W	Second Back-End configuration
ADDRESS	03	R/W	Command/Response Node address
INPUT_ENABLES	04	R/W	Source Processors with enabled input
CONTRIBUTORS	05	R/W	List of contributor sources
DESTINATION_ENABLES	06	R/W	List of enabled destinations
TIMEOUT	07	R/W	Time to wait on any one Contributor
TEM_STATISTICS	08	R/W	Mux for received TEM statistics
C/R_STATISTICS	09	R/W[1]	Command/Response statistics
SSR_HEADER	10	R/W	Header appended to SSR data
RESERVED	11-15	R/O[2]	Reserved for future use
Total	**16**		

1. On write, the value is ignored and the register is set to zero (0).
2. These registers are *Read-Only* and will always read back MBZ.

2.3.1 Back-End configuration register

In general, this register allows defeating those features of the EBM which in the normal course of operation would always be enabled. This functionality is present only to allow *testing* of

those features and perhaps to recover from single-point failure. Great care should be exercised in using anything other then the default values for this register.

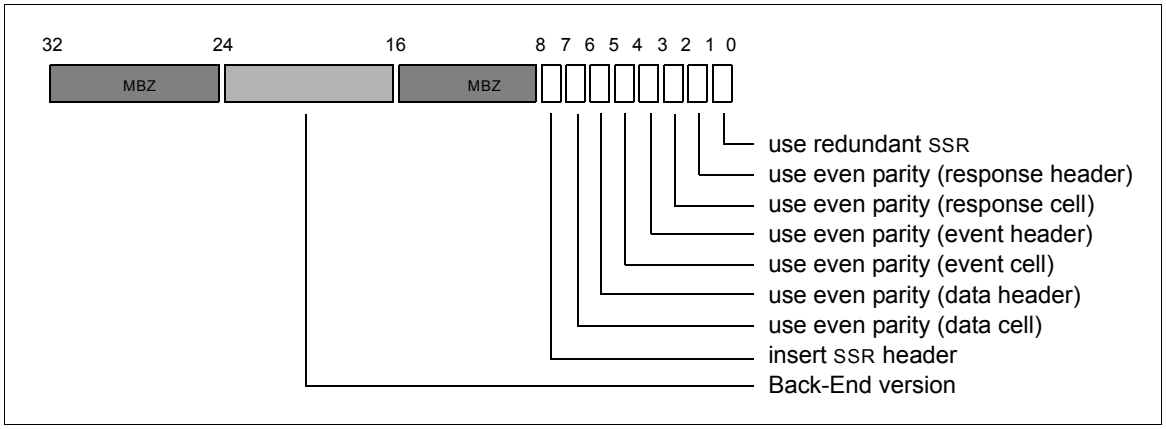

Figure 11 Back-End configuration register

use redundant SSR: Determines which one of the two Solid State Recorders (SSR) interfaces will serve as one of the EBM's potential destinations. For reliability, there are two identical SSRs interfaces, by convention, labelled the *primary* and *redundant* SSR. If the field is *clear*, the *primary* SSR is used by the EBM. If the field is *set*, the *redundant* SSR is used by the EBM.

use even parity (response header): Determines whether the header parity of the packet generated by the EBM when transmitting a *response* is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the response receiver will in fact detect header parity errors. In normal operation this field should be always be *clear*.

use even parity (response cell): Determines whether the cell parity of the packet generated by the EBM when transmitting a *response* is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the response receiver will in fact detect cell parity errors. In normal operation this field should be always be *clear*.

use even parity (event header): Determines whether the header parity generated by the EBM when transmitting *event* packets is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the destination receiver will in fact detect header parity errors. In normal operation this field should be always be *clear*.

use even parity (event cell): Determines whether the cell parity generated by the EBM when transmitting *event* packets is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the destination receiver will in fact detect cell parity errors. In normal operation this field should be always be *clear*.

use even parity (data header): Determines whether the header parity generated by the EBM when transmitting *data* packets is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the destination receiver will in fact detect header parity errors. In normal operation this field should be always be *clear*.

use even parity (data cell): Determines whether the cell parity generated by the EBM when transmitting *data* packets is *odd* or *even*. If the field is *clear*, *odd* parity is generated. If the field is *set*, *even* parity is generated. Note: This field is intended to be used to test whether the destination receiver will in fact detect cell parity errors. In normal operation this field should be always be *clear*.

insert SSR header: Determines whether or not any events sent to the SSR interface include as their first 32-bit word the value programmed into the register described in Section 2.3.11. If the field is *clear*, the pattern is *not* applied. If the field is *set*, the pattern *is* applied.

Back-End version: Specifies the hardware revision level of the FPGA which implements the Back-End interface of the EBM. The structure of this field is defined in Figure 12. The Front-End interface of the EBM uses two FPGA's. The version information for these FPGAs is discussed in sections 2.3.2 and 2.3.3. Note that this field is *read-only*.

2.3.1.1 Version ID

The EBM has both a Front-End and Back-End interface. The location of version information for the Back-End interface is described in Section 2.3.1. The location of version information for the Front-End interface is described in Section 2.3.2. In either case, the structure of this information is the same and is illustrated by Figure 12. The fields of this structure are somewhat self-explanatory with the exception of the ***type*** field. This field is intended to differentiate both the context in which the firmware for the FPGA was implemented and how it was intended to be used. Possible values for this field are enumerated within Table 10.

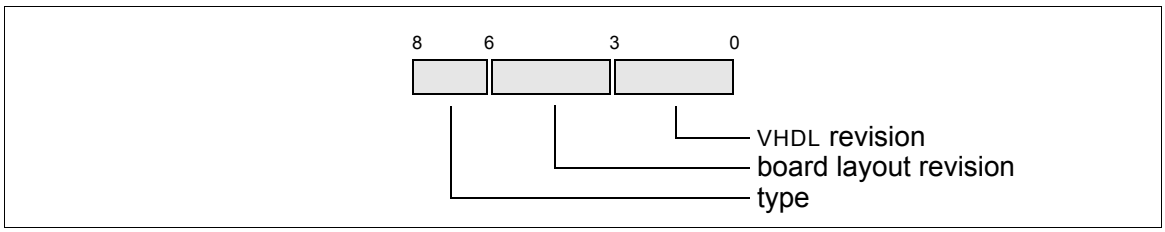

Figure 12 Structure of Revision Register or field

Table 10 Usage of the type field of the revision register

Value[1]	Description
00	Software emulation/engineering model
01	Engineering model
10	Qualification model
11	Flight model

1. In binary

2.3.2 Configuration register for Front-End "A"

The EBM consists of three FPGAs. One FPGA is used to implement its Back-End interface (discussed in Section 2.3.1) and the other two, its Front-End interface. The register described in this section (shown in Figure 13) defines the configuration of one half of the Front-End interface. Configuration of the other half is identical and is described in Section 2.3.3. The default value of this register provides a nominal Front-End configuration. This register allows defeating the nominal configuration features of the Front-End, which in the normal course of operation would never be changed.

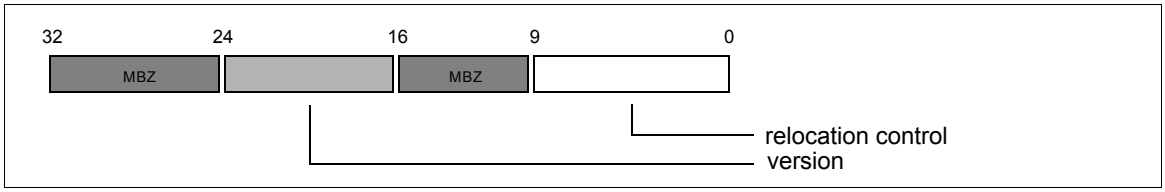

Figure 13 Front-End "A" configuration register

relocation control: To be described. The nominal values for relocation are specified by the default value of this field; however, any values for this field are allowed and will provide a legitimate relocation configuration. Note: This field currently has only six significant bits.

version: Specifies the hardware revision level of the *first* FPGA providing the Front-End implementation. The structure of this field is defined in Figure 12. Note that this field is *read-only.*

2.3.3 Configuration register for Front-End "B"

The EBM consists of three FPGAs. One FPGA is used to implement its Back-End interface (discussed in Section 2.3.1) and the other two, its Front-End interface. The register described

in this section (shown in Figure 14) defines the configuration of one half of the Front-End Interface. Configuration of the other half is identical and is described in Section 2.3.2. The default value of this register provides a nominal Front-End configuration. This register allows defeating the nominal configuration features of the Front-End, which in the normal course of operation would never be changed.

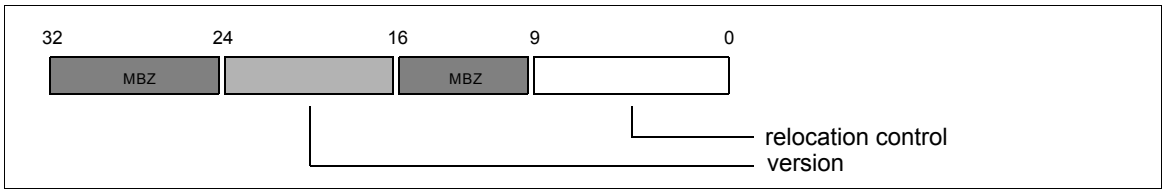

Figure 14 Front-End "B" configuration register

relocation control: To be described. The nominal values for relocation are specified by the default value of this field; however, any values for this field are allowed and will provide a legitimate relocation configuration. Note: This field currently has only six significant bits.

version: Specifies the hardware revision level of the *second* FPGA providing the Front-End implementation. The structure of this field is defined in Figure 12. Note that this field is *read-only.*

2.3.4 Address register

This register is used to specify the EBM's node address on the Command/Response fabric. (See [1].) Note that all nodes on a fabric must have a unique value. This register allows for definition of *only* the address's lower five bits. As the EBM is a slave on the fabric, the high order bit is an implied *zero* (0).

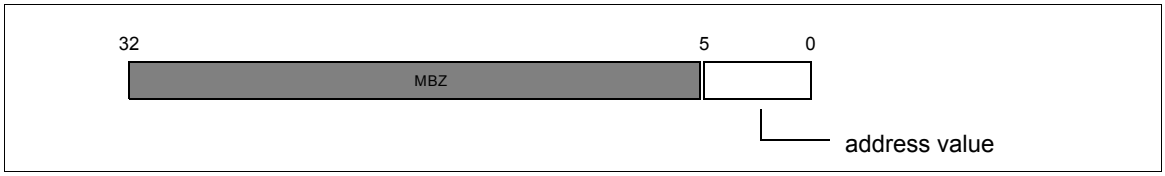

Figure 15 Address register

2.3.5 Input enables register

This register is used to mask or enable *input* to the twenty-four possible Source Processors of the EBM's Front-End. The format of this register is illustrated in Figure 16. Each bit offset corresponds to a specific Source Processor. If the bit at a specified offset is *set*, the input to the corresponding Source Processor is *enabled*. If the bit is *clear*, the input to the Source Processor is *disabled*. Note that the field corresponding to bit offset *zero* (the GEM) is *read-only* and is *set*, which implies that input from the GEM is always accepted and can never be masked off.

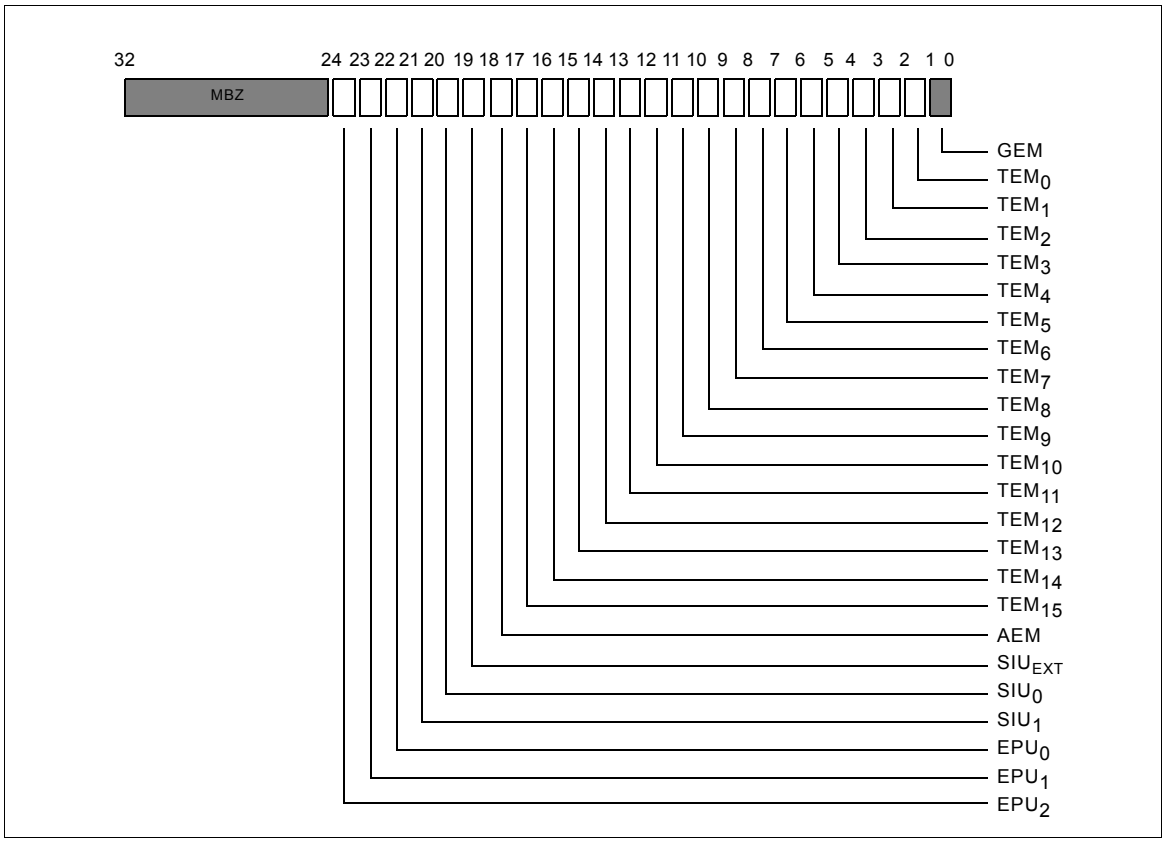

Figure 16 Input enables register

2.3.6 Contributors register

In general this register determines whether or not the back-end engines will ignore output from any one of their twenty-four Source Processors. The format of this register is illustrated in Figure 17. The Event-Building engine is only interested in the first eighteen (18) bit offsets, while the Transfer Engine only examines offsets 18 through 23. Consequently, this register, depending on engine type, serves two functions:

i. Determines which of eighteen possible sources will contribute to a built event. Potential sources are in bit-offsets zero (0) through seventeen (17). If the bit at a specified offset is *set*, a contribution from the corresponding source is expected to contribute to a built event If the bit is *clear*, the source is not expected to contribute to a built event.

ii. Determines whether a source processor, whether the transfer engine will transfer packets from processor to the packet's specified destination. The sources examined by the transfer engine are specified in bit-offsets eighteen (18) through twenty-three (23). If the bit at a specified offset is *set*, packets from the corresponding source are transferred. If the bit is *clear*, packets from the corresponding source are not transferred.

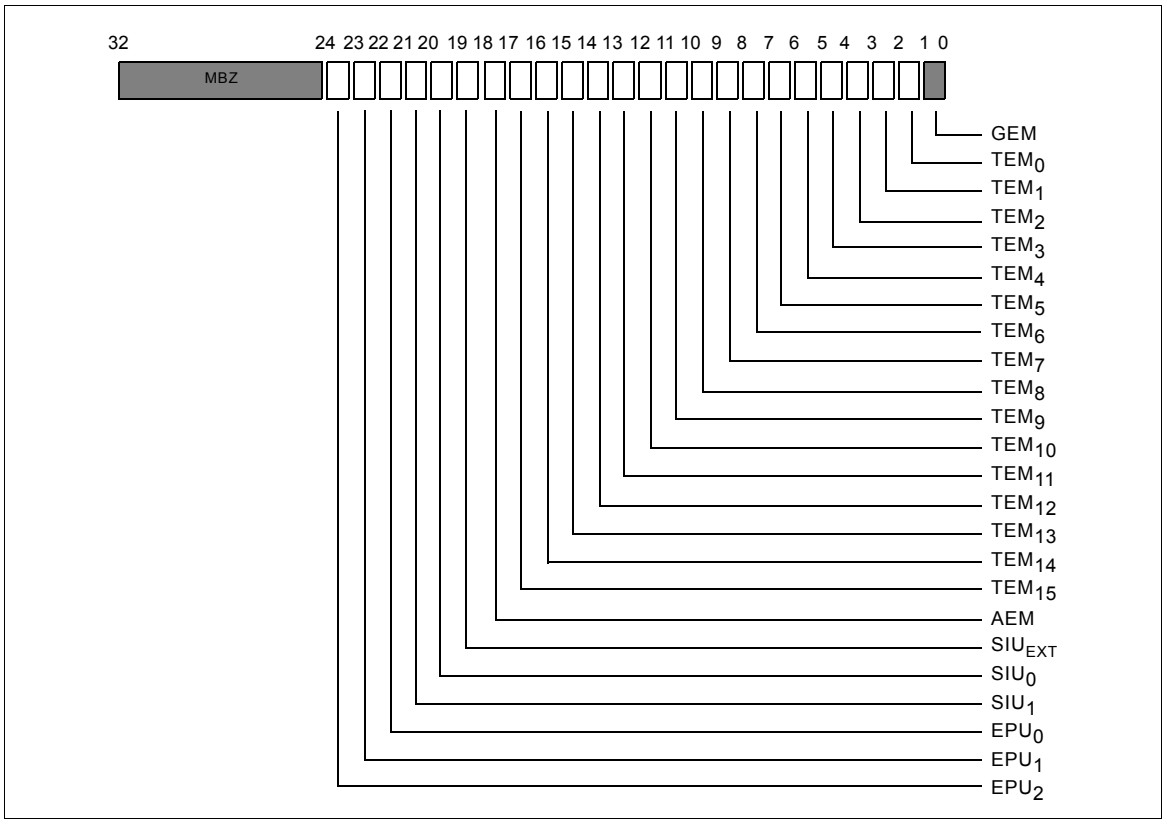

Figure 17 Contributors register

2.3.7 Destination enables register

The EBM determines the destination of any event packet it builds or data packets it transfers by using the destination address (of the LATp header) of the first contribution processed when building events and the destination address of a data packet it must transfer. (See Section 1.3.5.) These addresses fit into three classes:

singlecast: Send to one and only one destination. The destination is determined by the address specified in the LATp header as determined by Section 1.3.5.

broadcast: Sends to all destinations. The definition of "all" is specified by a destination list.

round-robin: Sends to the "next" available destination. The definition of which destinations participate in round-robin is specified by a destination list.

In both broadcast and round-robin addressing the set of destinations which will participate in either is configurable using the destination enables register. (See Figure 19.) This register specifies three different destination lists. The EBM has *seven* potential destinations for the packets it transmits; however, the SSR participates only in the singlecast class and cannot participate in either the broadcast or round-robin classes. Consequently, a bit-list which can be used to specify any one set of destinations has *six* fields as illustrated in Figure 19. If the bit at a specified offset is *set*, the corresponding destination is enabled. If the bit is *clear*, the destination is disabled.

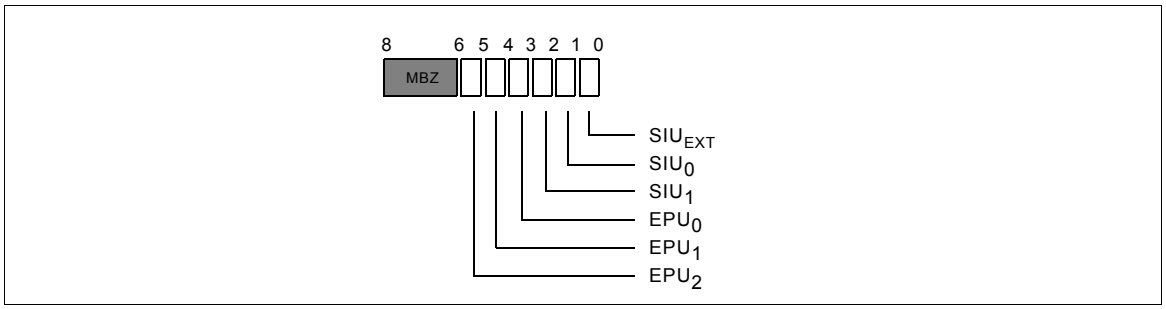

Figure 18 Destination list

The order of the three different destination lists within the register is as follows:

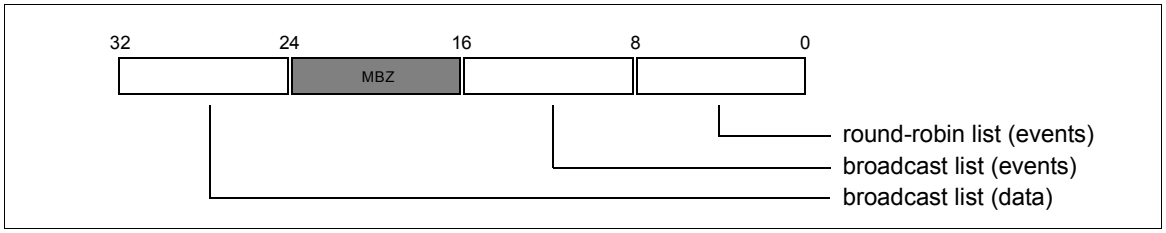

Figure 19 Destination enables register

round-robin list (events): List used when the EBM must round-robin an event packet.

broadcast list (events): List used when the EBM must broadcast an *event* packet.

broadcast list (data): List used when the EBM must broadcast a *data* packet.

2.3.8 Timeout

Determines how long the EBM should wait for any one contribution when building an Event. How this value is used in building an event is discussed in Section 1.3.3. The timeout value is expressed in units of system clock (`sysclk`), where one count is nominally 50 *nanoseconds*.

The minimum value for the timeout is 0 *x* 50 *nanoseconds* which equals 0, and the maximum value is 131,072 *x* 50 *nanoseconds* which equals approximately 6 milliseconds.

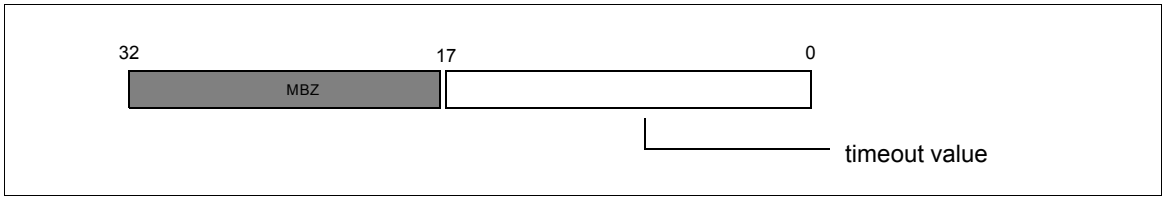

Figure 20 Timeout register

2.3.9 TEM event statistics mux register

The EBM has two counters (discussed in Section 2.4) to keep track of the number of received packets from its TEM's. As there are sixteen TEMs there must be a mechanism to specify which two of sixteen TEMS will be counted. The structure of this register is illustrated by Figure 21. The value written to any one the fields is the number (0 through 15) of the TEM to be counted.

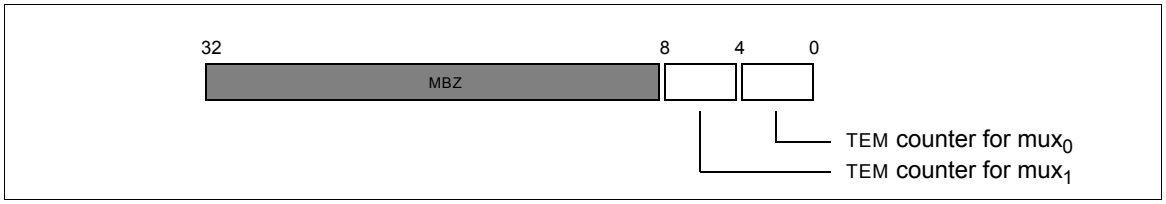

Figure 21 TEM event statistics mux register

2.3.10 Command/Response statistics register

The EBM is a node on the Command/Response fabric. As such it is obligated (as discussed in [1]) to keep statistics on both the commands it receives and the responses it transmits. These statistics are accessed in the register illustrated in Figure 22. Note that when the register is written, the value to be written is ignored and the register is set to *zero*.

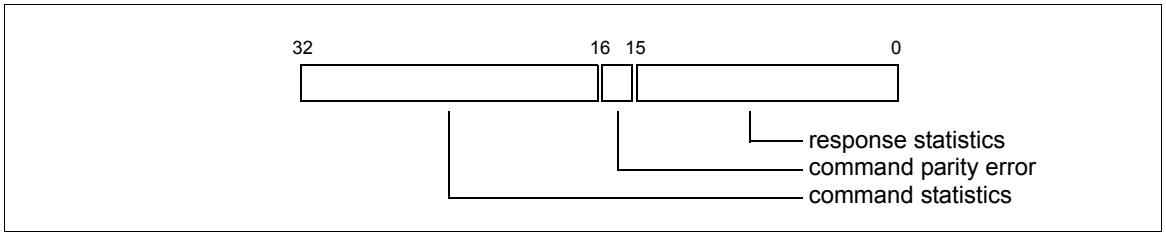

Figure 22 The Command/Response statistics register

response statistics: The packet statistics for the *outgoing response* wire. See [1] for a description of the structure of this field.

command parity error: This field is set when the ***EBM access descriptor*** (shown in Figure 27) of an incoming command has a parity error.

command statistics: The packet statistics for the *incoming command* wire. See [1] for a description of the structure of this field.

2.3.11 SSR header pattern

The first 32-bits of any LATp packet destined to the SSR interface receive special processing by the EBM *before* the packet is sent to the interface. These 32-bits of the packet include both the LATp header and its adjacent 16 bit MBZ field. Which type of processing depends on the state of the ***insert SSR header*** field of the register described in Section 2.3.1:

— If this field is *clear*, the first 32-bits of the sent data are *removed* before transmission to the spacecraft's SSR interface.

— If this field is *set*, the first 32-bits of the sent data are *replaced* before transmission to the spacecraft's SSR interface. The value which replaces these 32-bits is determined by the contents of the register illustrated in Figure 23.

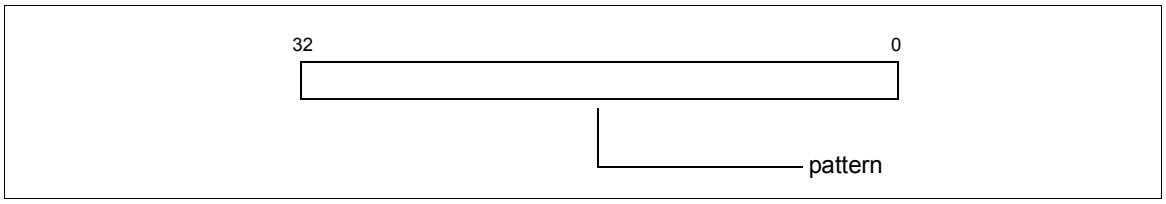

Figure 23 The SSR header pattern register

2.4 Event statistics

This section provides the counters used to tally and record LATp statistics associated with event reception and transmission. They may be broken into two classes:

Receive statistics: One register for each Source Processor with the exception of the sixteen TEMs. The sixteen individual TEM statistics are muxed into two registers as described in Section 2.3.9.

Transmit statistics: One register for each of the seven possible destinations.

The registers of this block are all 32-bits in length. Note that these registers are *read-only*. However, these counters may be set to zero, either on receipt of a global RESET command or on receipt of the dataless RESET command to the statistics block. (See Section 3.4.)

Table 11 The registers of the Event Statistics block

Name	Address	Access	Description
RECEIVE_GEM	00	R/O	Received statistics for the GEM
RECEIVE_AEM	01	R/O	Received statistics for the AEM
RECEIVE_SIU0	02	R/O	Received statistics for SIU_{ext}
RECEIVE_SIU1	03	R/O	Received statistics for SIU_0
RECEIVE_EPU0	04	R/O	Received statistics for SIU_1
RECEIVE_EPU1	05	R/O	Received statistics for EPU_0
RECEIVE_EPU2	06	R/O	Received statistics for EPU_1
RECEIVE_SIU2	07	R/O	Received statistics for $SIU_{external}$
RECEIVE_TEM_MUX0	16	R/O	Statistics for TEM muxed to MUX_0
RECEIVE_TEM_MUX1	17	R/O	Statistics for TEM muxed to MUX_1
RESERVED	18-31	R/O[1]	Reserved for future use
TRANSMIT_SIU0	32	R/O	Transmitted statistics for SIU_{ext}
TRANSMIT_SIU1	33	R/O	Transmitted statistics for SIU_0
TRANSMIT_EPU0	34	R/O	Transmitted statistics for SIU_1
TRANSMIT_EPU1	35	R/O	Transmitted statistics for EPU_0
TRANSMIT_EPU2	36	R/O	Transmitted statistics for EPU_1
TRANSMIT_SIU2	37	R/O	Transmitted statistics for $SIU_{external}$
TRANSMIT_SSR	38	R/O	Transmitted statistics for SSR
RESERVED	39-63	R/O[2]	Reserved for future use
Total	**64**		

1. These registers are *Read-Only* and will read always read back MBZ.
2. These registers are *Read-Only* and will read always read back MBZ.

2.4.1 Events received registers

The EBM's Source Processors receive LATp packets. The register illustrated in Figure 24 defines the packet statistics kept by each of the 24 processors.

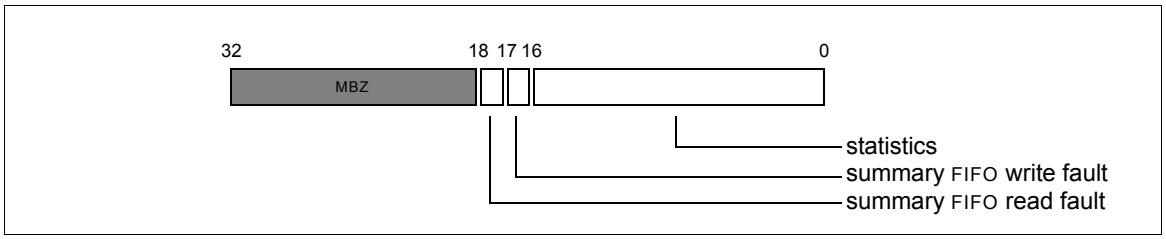

Figure 24 An Events received register

statistics: The packet statistics for an incoming *event* wire. See [1] for a description of the structure of this field.

summary FIFO (write fault): This field is set when an attempt is made to write to the Source Processor's Summary FIFO when it is already *full*. This indicates a failure of the EBM to properly generate back pressure in a timely fashion or else a failure on the sender's part to properly respect flow-control by initiating transmission while the `pause` line is asserted.

summary FIFO (read fault): This field is set when an attempt is made to read from the Source Processor's Summary FIFO when it is *empty*. This indicates a logic failure in the EBM.

2.4.2 Events transmitted registers

The EBM's scheduler transmits LATp packets. The register illustrated in Figure 25 defines the packet statistics kept for each of the possible 7 destinations.

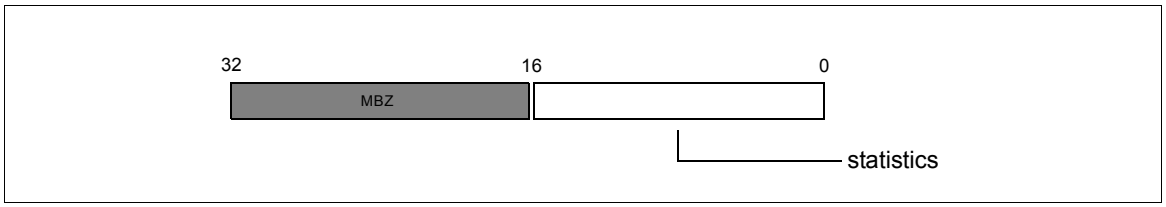

Figure 25 An Events transmitted register

statistics: The packet statistics for an outgoing *event* wire. See [1] for a description of the structure of this field.

Chapter 3
Commanding

3.1 Overview

This chapter describes the remote protocol necessary to access both the registers[1] and functional blocks of the EBM. It follows the Command/Response Protocol discussed in [1]. The registers of the EBM are organized into a hierarchy of two blocks as illustrated in Figure 26.

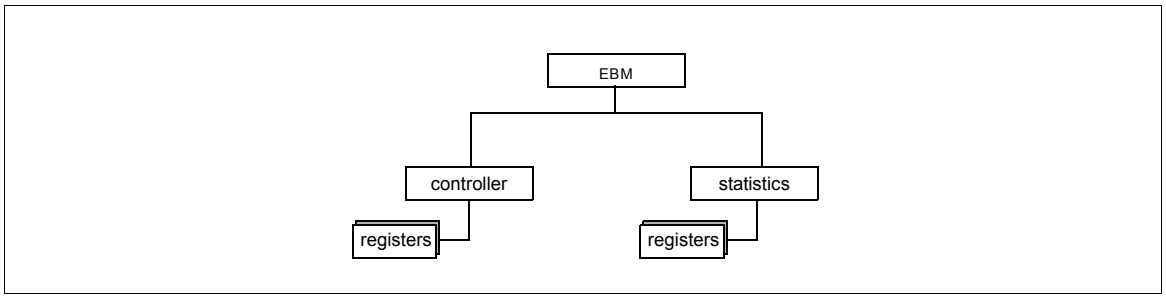

Figure 26 Hierarchy of target types

3.1.1 Conventions

All data structures described in this chapter are from the perspective of being "on-the-wire." Therefore, the left-most field in any description is transmitted *first*, or is considered to be transmitted on the *zeroth* clock. Fields are numbered from the beginning of the ***packet header*** described in [1].

1. Enumerated and described in Chapter 2

3.2 The EBM's access descriptor

Directly following the LATp header of any received command packet is a fixed size, 16-bit structure, called the EBM's ***access descriptor***. This descriptor is a parameterization of the access rules required to address the functional blocks and registers of the EBM. Its structure is illustrated in Figure 27:

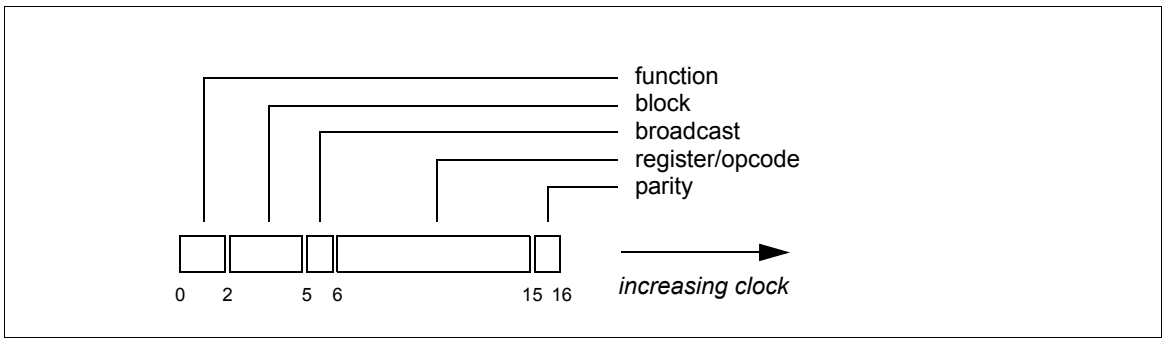

Figure 27 EBM access descriptor

function: Enumerates what *type* of access is required of the target by the command, for example, whether the command will either *read* or *write* the specified register. The valid enumerations for this field are described in [1].

block: This field enumerates which of the two blocks of the EBM are to be accessed. The correspondence between block type and number is enumerated in Table 12.

broadcast: Determines how the ***register/opcode*** field is interpreted. If this field is *false*, the ***register/opcode*** field is used to determine which register to access. If this field is *true*, the ***register/opcode*** field is ignored and the access descriptor is applied to *all* the registers of the type specified by the ***block*** field. Note: A broadcast operation is *not* permitted if the ***function*** field specifies a `read` operation.

register/opcode: If the ***function*** field has a value of either `read` or `load`, this field contains the *number* of the register to be accessed. If the function is `dataless`, this field determines the *type* of dataless access.

parity: The *odd* parity value over the entire ***access descriptor***.

Table 12 Block numbers of the EBM

Name	Number
`controller`	`0`
`statistics`	`1`

3.3 Accessing the controller

An enumeration and description of the registers of the controller block may be found in Section 2.3, "Controller registers". All registers of the controller are 32-bits in length.

3.3.1 Dataless commands

Table 13 The controller's dataless commands

Name	Opcode	Description
NOP	0	No operation
RESET	1	Hard reset of the EBM
Total	**2**	

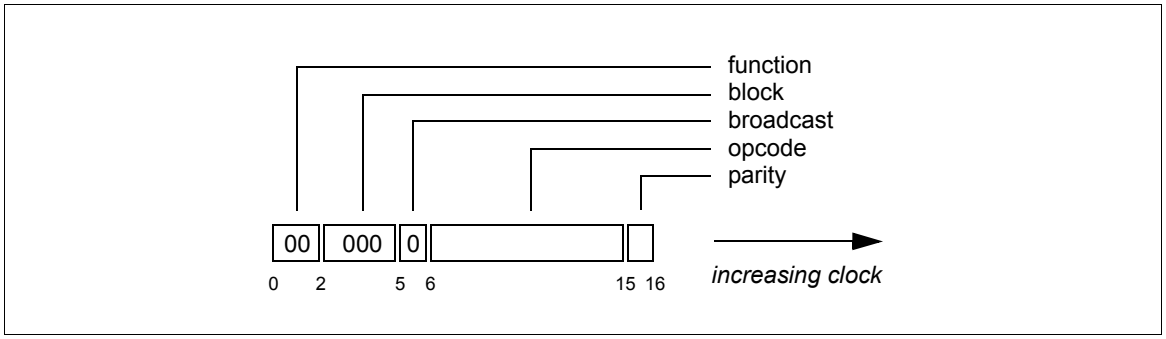

Figure 28 Access descriptor for the controller's dataless commands

`Dataless` functions do *not* require a payload. As a `dataless` function requires no response, the ***respond*** field of the packet is set to *false*.

3.3.2 Load commands

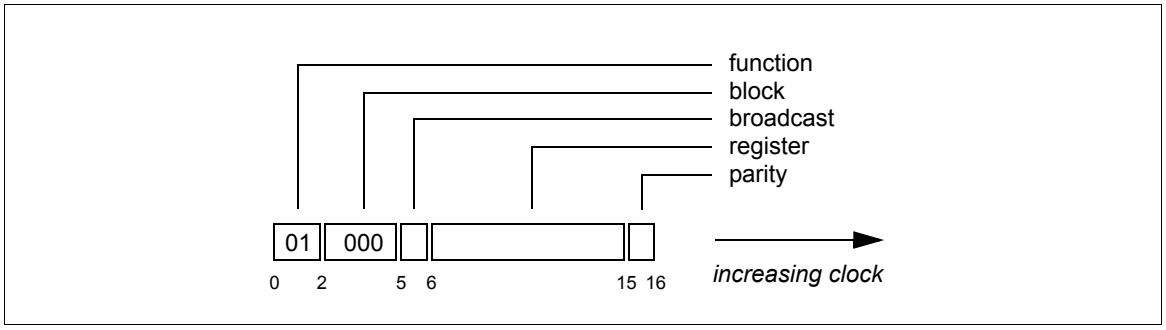

Figure 29 Access descriptor for the controller's register load commands

All registers of the controller are 32 bits long. Consequently, all `Load` functions require a 32-bit payload. The format of this payload is illustrated in Figure 30. As a `Load` function does not require a response, the ***respond*** field of the packet is set to *false*.

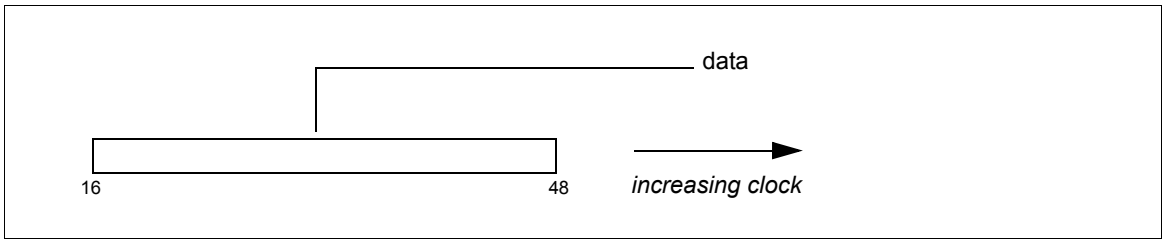

Figure 30 Payload for the controller's register load commands

3.3.3 Read commands

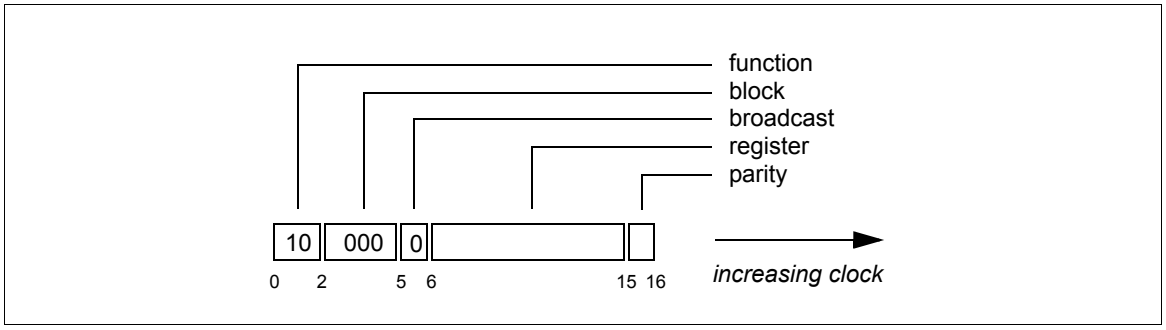

Figure 31 Access descriptor for the controller's register read commands

`Read` functions require *no* payload. The value of the register read is returned as a response. As these `reads` *do* generate a response, the command packet's ***respond*** field is set to *true*. The format of that response is illustrated in Figure 32.

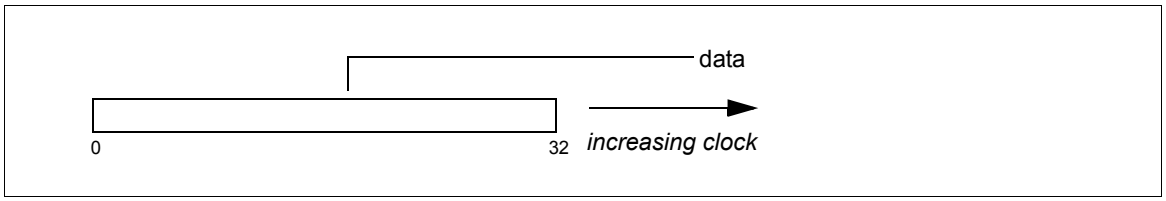

Figure 32 Response to a register read command of the controller

3.4 Accessing the statistics block

An enumeration and description of the registers of the controller block may be found in Section 2.4, "Event statistics". All registers of this block are 32-bits in length.

3.4.1 Dataless commands

Table 14 The dataless commands for the statistics block

Name	Opcode	Description
RESET	0	Reset all statistics registers
Total	**1**	

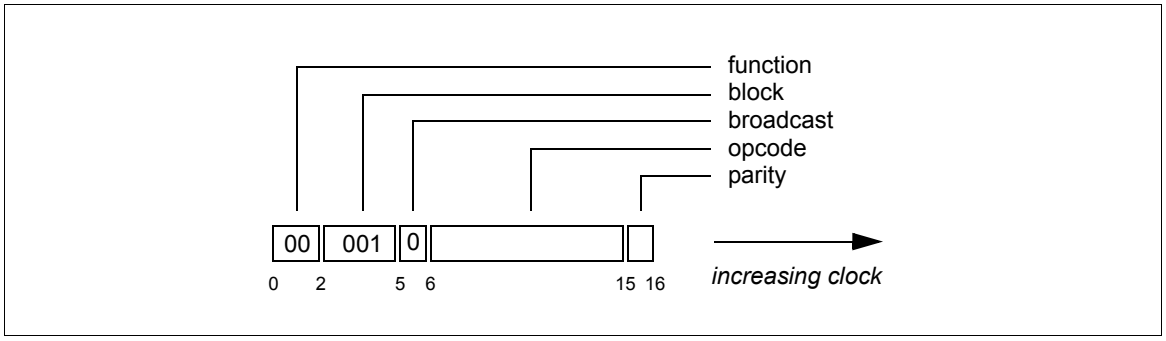

Figure 33 Access descriptor for dataless commands to the statistics block

`Dataless` functions do *not* require a payload. As a `dataless` function requires no response, the ***respond*** field of the packet is set to *false*.

3.4.2 Read commands

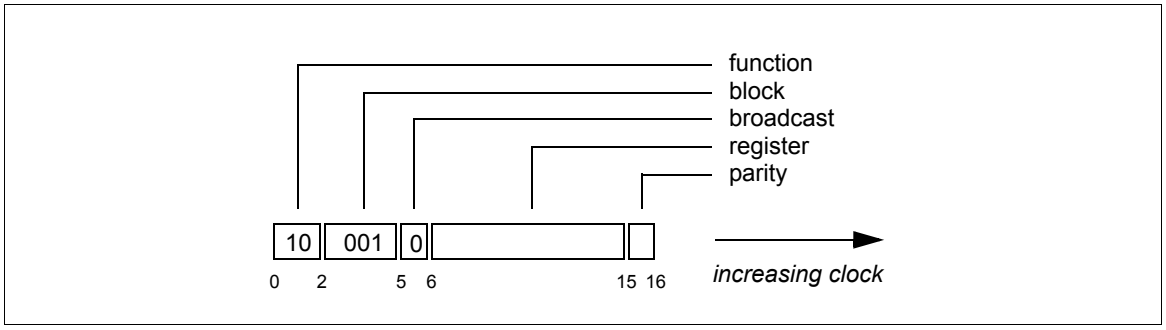

Figure 34 Access descriptor for register read commands of the statistics block

`Read` functions require *no* payload. The value of the register read is returned as a response. As these `reads` *do* generate a response, the command packet's ***respond*** field is set to *true*. The format of that response is illustrated in Figure 35.

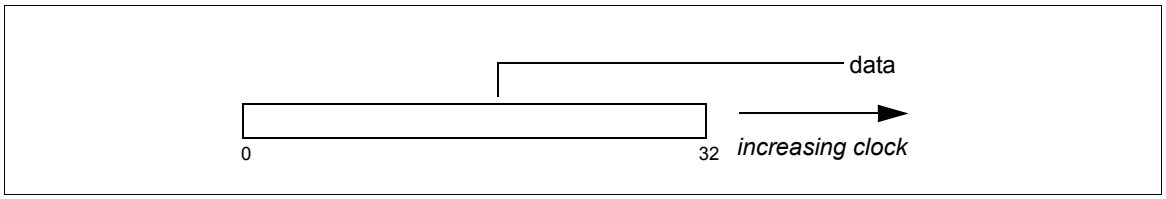

Figure 35 Response to a register read of the statistics block