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 LCB (LAT Communications Board). This board is designed to provide communications between the modules within the LAT’s Compact PCI crates and the nodes of the LAT’s command and event fabrics.

Hardware compatibility

This document assumes the following hardware revision:

LCB: Version TBD

Intended audience

This document is intended principally as a guide for the developer and users of the LAT Communications Board (LCB). Users include:

- Developers of the sub-system electronics which interface with the LCB
- 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


2. Actel, CorePCI Target, Master, and Master/Target Product specification (v3.0), October 2000.

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/ElecDAQ/ElecDAQ_home.htm), click Hardware and then click List of all documents.
## Document Control Sheet

### Table 1 Document Control Sheet

<table>
<thead>
<tr>
<th>Document</th>
<th>Title: The LAT Communications Board Programming ICD specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>4.0</td>
</tr>
<tr>
<td>Issue:</td>
<td>4</td>
</tr>
<tr>
<td>Edition:</td>
<td>English</td>
</tr>
<tr>
<td>ID:</td>
<td>LAT-TD-00860</td>
</tr>
<tr>
<td>Status:</td>
<td>Under release control</td>
</tr>
<tr>
<td>Created:</td>
<td>February 9, 2002</td>
</tr>
<tr>
<td>Date:</td>
<td>January 31, 2005</td>
</tr>
<tr>
<td>Access:</td>
<td>V:\GLAST\Electronics\Design Documents\LCB\V4.0\frontmatter.fm</td>
</tr>
<tr>
<td>Keywords:</td>
<td>LAT Communications Board LCB</td>
</tr>
</tbody>
</table>

### Tools

<table>
<thead>
<tr>
<th>DTP System:</th>
<th>Adobe FrameMaker</th>
<th>Version: 6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout</td>
<td>Software Documentation</td>
<td>Version: V2.0 - 5 July 1999</td>
</tr>
<tr>
<td>Template:</td>
<td>Layout Templates</td>
<td>Version: --</td>
</tr>
<tr>
<td>Content Template:</td>
<td>--</td>
<td>Version: --</td>
</tr>
</tbody>
</table>

### Authorship

| Coordinator: | Michael Huffer |
| Written by:  | Michael Huffer |

### Table 2 Approval sheet

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gunther Haller</td>
<td>LAT Chief Electronics Engineer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JJ Russell</td>
<td>Flight Software Lead</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3  Document Status Sheet

<table>
<thead>
<tr>
<th>Version</th>
<th>Issue</th>
<th>Date</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>5/29/2002</td>
<td>Initial draft</td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
<td>6/25/2002</td>
<td>Incorporated comments from Curt, Gunther, JJ, and Tony</td>
</tr>
<tr>
<td>2.0</td>
<td>1</td>
<td>12/04/2002</td>
<td>System architectural decision made to support at most, only one CPU per crate (EPU or SIU). This allowed for a considerable simplification of the LCB, although conceptually the design remains the same. This document has been updated accordingly. Note: while conceptually similar, this simplification changes the LCB in many subtle ways. A careful reading of the document is advised. Fixed errata noted by Curt.</td>
</tr>
<tr>
<td>2.1</td>
<td>1</td>
<td>12/05/2002</td>
<td>Added capability to monitor environmental quantities and control power of the both the GASU and PDU.</td>
</tr>
<tr>
<td>2.2</td>
<td>1</td>
<td>1/29/2003</td>
<td>As we move closer to a real board:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. Removed monitoring and control capability as defined above (its moved to SIB).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Defined vendor and device ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. The results FIFO now returns a sentinel value rather then TARGET_ABORT on empty</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Explained more fully the use of the intermediate FIFOs in appendix C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5. moved around LATp status bits to accommodate above item</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6. Changed size of BAR0 and BAR1 spaces</td>
</tr>
<tr>
<td>2.4</td>
<td>1</td>
<td>1/31/2003</td>
<td>As pointed out by Leonid, I screwed up structure and explanation of intermediate FIFOs. Hopefully, I’ve fixed things up.</td>
</tr>
<tr>
<td>2.5</td>
<td>1</td>
<td>4/30/2003</td>
<td>Made the document layout and name in conformance with GXH (and NASA) wishes for CDR. Now that the board is coming together, we’ve decided all the registers should be in memory space. Have adjusted document accordingly. Also moved relative position of LATp header in commands and responses (I had it wrong).</td>
</tr>
<tr>
<td>Version</td>
<td>Issue</td>
<td>Date</td>
<td>Changes</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 2.6     | 1     | 6/09/2003  | More changes made as we continue to commission board. This version will be used as the baseline to develop the second generation LCB. Changes include:  
1. Removed interrupt acknowledge register from configuration space and moved its functions into registers within memory space. CorePCI interface in target/DMA configuration did not expose this register  
2. Incorrect field size for board ID field within CSR register. Also moved interrupt enable function to this register. |
| 3.0     | 1     | 6/14/2003  | More changes made as we continue to commission board. This version will be used as the baseline to develop the second generation LCB. Changes include:  
1. Removed interrupt acknowledge register from configuration space and moved its functions into registers within memory space. CorePCI interface in target/DMA configuration did not expose this register  
2. Incorrect field size for board ID field within CSR register. Also moved interrupt enable function to this register. |
| 3.7     | 1     | 3/05/2004  | Updated fonts.                                                          |
| 3.8     | 1     | 5/25/2004  | Corrected typos.                                                        |
| 3.8     | 2     | 6/10/04    | Updated references and PDF TOC.                                         |
| 3.9     | 1     | 9/24/04    | Continuing to update document, but more changes are still to come.      |
| 4.0     | 4     | 1/31/05    | Continuing to update document, but more changes are still to come.      |
Table of Contents

Abstract ................................................................. .3
Hardware compatibility .............................................. .3
Intended audience ....................................................... .3
Conventions used in this document ................................. .3
References .................................................................... .4
Document Control Sheet .................................................. .5
Document Status Sheet .................................................... .6
List of Figures .............................................................. 13
List of Tables .............................................................. 17

Chapter 1
Principals of operation .................................................. 19
  1.1 Introduction .......................................................... 19
  1.2 FIFOs .................................................................. 21
  1.3 Front-End Interface ............................................... 22
      1.3.1 Command/Response ........................................ 22
      1.3.1.1 Sysclk signals ........................................... 22
      1.3.1.2 Reset signals ........................................... 23
      1.3.1.3 Command/Response signals ......................... 24
  1.3.2 Event ............................................................. 25
  1.4 Back-End Interface ............................................... 25
      1.4.1 The Request Engine ....................................... 25
1.4.2 The Event Engine .................................................. 27
1.5 Back-End Interface ................................................... 28
  1.5.1 Transfer Request ................................................. 29
  1.5.2 Transfer Result .................................................. 31
  1.5.3 Transfer Event ................................................... 32
  1.5.4 Interrupts ......................................................... 34
Chapter 2
Front-End Interface ..................................................... 37
  2.1 Introduction ....................................................... 37
    2.1.1 Accessing the Front-End CSR through the Command/Response fabric ........................................... 38
  2.2 Control and Status Register (CSR) ................................ 39
  2.3 FIFO faults register ................................................ 41
  2.4 Event Statistics ...................................................... 42
Chapter 3
PCI Interface ........................................................... 45
  3.1 Configuration space ................................................ 46
    3.1.1 Command Register ............................................... 49
    3.1.2 Status Register .................................................. 49
    3.1.3 Revision ID register ............................................ 49
  3.2 Memory space ....................................................... 50
    3.2.1 Control and Status Register (CSR) ................................ 51
    3.2.2 Circular buffer ................................................... 52
    3.2.3 Circular Buffer Deallocation register ............................ 54
    3.2.4 Request queue ..................................................... 54
    3.2.5 Result queue ...................................................... 54
    3.2.6 Event queue ....................................................... 55
    3.2.7 Interrupt State .................................................... 55
    3.2.8 Fabric Select ...................................................... 58
Chapter 4
Transactions ............................................................ 61
  4.1 Request-descriptors ............................................... 62
  4.2 Request-lists ....................................................... 63
  4.3 Request-items ....................................................... 64
    4.3.1 Requests which involve packet transmission ................. 65
      4.3.1.1 Constructing the header for transmitted packets ............ 66
      4.3.1.2 Transmit event request ..................................... 67
      4.3.1.3 Transmit command request ................................... 67
      4.3.1.4 Transmit command, wait on response request ............... 68
    4.3.2 Requests which do not involve packet transmission .......... 69
      4.3.2.1 Accessing Front-End registers .............................. 70
4.3.2.2 Mark time request ............................. 70
4.3.2.3 Reset LAT request ............................ 70
4.3.2.4 Access CSR request ........................... 71
4.3.2.5 Access FIFO faults register request .......... 71
4.3.2.6 Access Event Statistics register request .... 72
4.4 Result-descriptors ................................ 72
  4.4.1 Transfer Engine Errors .......................... 73
  4.4.2 Null descriptors .................................. 75
4.5 Result-lists ......................................... 75
4.6 Result-items ........................................ 76
  4.6.1 Request Engine errors ......................... 76
  4.6.2 Results which involve packet transmission .. 77
    4.6.2.1 Transmit event result ...................... 77
    4.6.2.2 Transmit command result .................... 78
    4.6.2.3 Transmit command, wait on response result 78
  4.6.3 Results which do not involve packet transmission 79
    4.6.3.1 Mark time result ............................ 79
    4.6.3.2 Reset LAT result ............................ 79
    4.6.3.3 Access CSR result ............................ 79
    4.6.3.4 Access FIFO faults result ................... 80
    4.6.3.5 Access Event Statistics result .............. 80

Chapter 5

Events .................................................... 83
  5.1 Events and Event Packets ........................ 83
  5.2 Overview of packet processing .................... 84
  5.3 Event-descriptors .................................. 86
    5.3.1 Event errors .................................... 87
      5.3.1.1 Receive errors ................................ 88
      5.3.1.2 Transfer Errors ............................... 88
  5.4 Event packet structure ............................ 89
  5.5 Processing truncated events ....................... 90
  5.6 Deallocating packets .............................. 94
## List of Figures

| Figure 1 | p. 21 | Block diagram of the LAT Communications Board |
| Figure 2 | p. 23 | Processing of the Sysclk signals |
| Figure 3 | p. 23 | Processing of the Reset signals |
| Figure 4 | p. 24 | Processing of the Command/Response signals |
| Figure 5 | p. 25 | Path decode logic |
| Figure 6 | p. 26 | Relationship between intermediate FIFOs (normal transfer) |
| Figure 7 | p. 26 | Relationship between intermediate FIFOs (aborted transfer) |
| Figure 8 | p. 27 | The Request Engine |
| Figure 9 | p. 28 | The Event Engine |
| Figure 10 | p. 29 | Developing the signals used to activate the Transfer Engine |
| Figure 11 | p. 30 | Structure of any one item in the pending requests FIFO |
| Figure 12 | p. 32 | Structure of any one item in the pending results FIFO |
| Figure 13 | p. 34 | The pending event descriptor |
| Figure 14 | p. 35 | Developing the signals used to generate an interrupt |
| Figure 15 | p. 38 | Command Packet used to access Front-End CSR |
| Figure 16 | p. 39 | Front-End CSR Register |
| Figure 17 | p. 41 | Structure of a FIFO fault |
| Figure 18 | p. 42 | FIFO faults register |
| Figure 19 | p. 42 | Event statistics register |
| Figure 20 | p. 46 | PCI configuration header for LCB |
| Figure 21 | p. 49 | Command Register of the LCB’s PCI Configuration Header |
| Figure 22 | p. 49 | Status Register of the LCB’s PCI Configuration Header |
| Figure 23 | p. 50 | Revision ID Register of the LCB’s PCI Configuration Header |
Figure 24  p. 51  Command and Status Register (CSR)
Figure 25  p. 53  LCB Circular Buffer
Figure 26  p. 53  EVENTS_BASE register
Figure 27  p. 54  EVENTS_FREE register
Figure 28  p. 56  Interrupt State Register
Figure 29  p. 59  Fabric Select Register
Figure 30  p. 62  Relationship between a transaction request and its result
Figure 31  p. 63  The request-descriptor
Figure 32  p. 64  The request-list
Figure 33  p. 64  One item of a request-list
Figure 34  p. 65  Structure of opcodes used for packet transmission and reception
Figure 35  p. 66  Structure of a template header
Figure 36  p. 67  Structure of “Transmit event” request
Figure 37  p. 68  Structure of “Transmit Command” request
Figure 38  p. 68  Structure of “Transmit command, wait for response” request
Figure 39  p. 69  Structure of opcodes that do not involve packet transmission
Figure 40  p. 70  Structure of “Mark time” request
Figure 41  p. 71  Structure of “Reset LAT” request
Figure 42  p. 71  Structure of “Access CSR” request
Figure 43  p. 72  Structure of “Access FIFO faults register” request
Figure 44  p. 72  Structure of “Command Address register” request
Figure 45  p. 73  Result-descriptor
Figure 46  p. 75  Result-list
Figure 47  p. 76  One generic item of a LCB result-list
Figure 48  p. 78  Structure of “Transmit event” result
Figure 49  p. 78  Structure of “Transmit command” result
Figure 50  p. 78  Structure of a “Transmit command, wait on response” result
Figure 51  p. 79  Structure of a “Mark time” result
Figure 52  p. 79  Structure of a “LAT reset” request
Figure 53  p. 80  Structure of a “Access CSR” result
Figure 54  p. 80  Structure of a “Access FIFO faults” result
Figure 55  p. 80  Structure of a “Command Address” result
Figure 56  p. 85  Event packets and the event queue
Figure 57  p. 86  Event packets and the event queue
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 58</td>
<td>p. 86</td>
<td>Event processing flow</td>
</tr>
<tr>
<td>Figure 59</td>
<td>p. 87</td>
<td>The event descriptor</td>
</tr>
<tr>
<td>Figure 60</td>
<td>p. 87</td>
<td>Event errors</td>
</tr>
<tr>
<td>Figure 61</td>
<td>p. 90</td>
<td>Event packet structure</td>
</tr>
<tr>
<td>Figure 62</td>
<td>p. 91</td>
<td>Processing an event which fits in a single packet</td>
</tr>
<tr>
<td>Figure 63</td>
<td>p. 92</td>
<td>Processing an event which spans two packets</td>
</tr>
<tr>
<td>Figure 64</td>
<td>p. 93</td>
<td>Processing an event which spans “n” packets</td>
</tr>
</tbody>
</table>
List of Tables

Table 1  p. 5  Document Control Sheet
Table 2  p. 5  Approval sheet
Table 3  p. 6  Document Status Sheet
Table 4  p. 22  Attributes of the FIFOs used by the LCB
Table 5  p. 29  Scheduling priority
Table 6  p. 48  CorePCI customization constants for LCB
Table 7  p. 50  Memory registers for LCB
Table 8  p. 52  Interrupt conditions for event queue
Table 9  p. 54  PCI access rules for the REQUEST_QUEUE
Table 10  p. 55  PCI access rules for the RESULT_QUEUE
Table 11  p. 55  PCI access rules for the EVENT_QUEUE
Table 12  p. 66  Opcodes for requests which involve packet transmission
Table 13  p. 69  Opcodes for requests that do not involve packet transmission
Table 14  p. 74  Meaning of result-descriptor status with the direction field indicates an out-bound error
Table 15  p. 74  Meaning of result-descriptor status with the direction field indicates an in-bound error
Table 16  p. 77  Receive errors
Table 17  p. 88  Receive status codes
Table 18  p. 88  Transfer status codes
Chapter 1

Principals of operation

1.1 Introduction

The LAT Communication Board (LCB) is a bridge between the two different communication fabrics used by the LAT and PCI. Using this board, PCI clients\(^1\) may both send and receive packets to and from the nodes of these fabrics. The LCB is composed of two interfaces:

**The Front-End:** Provides an interface to the two different (Command/Response and Event) fabrics of the LAT. These fabrics are described in [1]. The interface consists principally of LATP transmitters and receivers used by the LCB to send and receive packets to and from the various nodes on the two different fabrics.

**The Back-End:** The back-side interface is a 33 MHZ, 2.2 compliant PCI interface. The interface contains a DMA capability and may be both a master (initiator) and slave (target) on the PCI bus. Notification of I/O activity and completion is signalled through PCI interrupts. The PCI interface is provided by the *Actel* CorePCI configured as Target+Master (see [2]) and is implemented on an *Actel* SX-xxA series FPGA.

The Front-End and Back-End interfaces also define two distinct clock domains. The Front-End interface operates at the LAT system clock frequency (nominally 20 MHZ) and the Back-End interface operates at the nominal PCI frequency (33 MHZ). Of course, information must be exchanged between the two different clock domains, and all such exchanges are performed through asynchronous FIFOs. One side of the FIFO operates in one clock domain, and the other side in the other clock domain. Physically, the LCB comes in two form factors:

---

\(^1\) Typically, software, residing on a SBC (Single Board Computer) sharing the same PCI backplane as the LCB.
i. A 6U (100 mm by 160 mm) compact PCI (cPCI) card. This form factor is used for flight. For this form factor, Front-End I/O signals are brought in through the board’s P2 rear connector by assigning these signals to PCI’s “user defined I/O” pins. Consequently, this form factor assumes it will be used in conjunction with a custom rear-panel I/O backplane. (See Appendix A.)

ii. A PMC mezzanine card. This form factor is used for ground support (EGSE). For this form factor, Front-End I/O signals are brought in through the board’s front panel using a 100 pin (socket) “micro-D” connector. (See Appendix A.)

Independent of form factor, the logical and functional interface for the LCB remains the same. Logically, the board may be thought of as three, relatively independent, functional units:

**Request Engine:** This engine is part of the Front-End interface and has four primary functions:

- Allows command transmission and response reception with the modules of the Command/Response fabric.
- Couples command and response together and thus allows their treatment by the user as a single atomic *transaction*.
- Time tags and synchronizes transactions (to within the granularity of the system clock).
- Allows *transmission* of events onto the Event fabric. Note that the Event Engine (described below) is used to receive events from the event fabric.

The programming model allows the user to compose, bundle and submit *lists* of transactions for the LCB to execute.

**Event Engine:** This engine is part of the Front-End interface and allows for the reception of data from the Event fabric. This data may have been sourced from any of the modules or LCBs of the event fabric. Note that the Request Engine (described above) is used to *transmit* events to the event fabric.

**Transfer Engine:** This engine is part of the Back-End interface and is responsible for transferring transaction requests, results, and events either from or to user PCI memory. In order to maintain adequate performance all information is transferred using Direct Memory Access (DMA). Since events arrive unsolicited, the Transfer engine includes a memory allocator in order to determine where in user PCI memory events are to be transferred.

In short, the Request’s Engine’s principal function is to provide access to the registers and functional blocks of the modules on the Command/Response fabric. The Event Engine *receives* event data from the various nodes of the event fabric and this could include data from both electronics modules (for example, the TEMs) and other LCBs. The Request Engine performs double duty, as in addition to its Command/Response responsibilities, it is also used to *send* event data onto the event fabric.

---

1. In the LAT, through the Event Builder. (See [1].)
Any information to be sent to other nodes of a fabric is first prepared in user memory and then queued to the LCB, which in turn, DMA’s this information into an internal buffer for later execution. A block diagram of the LCB is illustrated in Figure 1:

![Figure 1 Block diagram of the LAT Communications Board](image)

### 1.2 FIFOs

Because of the necessity to operate the three different functional Engines (Request, Event, and Transfer) more or less asynchronously with respect to each other, all communication between the various engines is buffered through a series of FIFOs. In some instances a FIFO is also used as a barrier between that portion of the LCB which runs at the system clock rate (20 MHZ) and that portion which runs at the PCI clock rate (33 MHZ). The FIFOs of the LCB can be categorized three ways:

- **queues:** This FIFO summarizes to the user any information DMA’d from the LCB into user PCI memory. This information includes arrived events and the results from using the Request Engine.

- **buffers:** These FIFOs hold the serial data which are transmitted to, or received from, a fabric. There data are transferred via the Transfer engine to and from user PCI memory. These holding buffers allow the process of serialization or deserialization to occur uninterrupted. These FIFOs are internal to the LCB and are not exposed directly to the user.
**pending events and pending results FIFOs:** These FIFOs summarize the information held by the buffers described above. For example, each entry in the pending event FIFO specifies one event packet. The data of this packet is contained in the Event buffer FIFO described above. The user has no direct access to these FIFOs. This FIFO summarizes information held by the Results buffer FIFO described above. Each entry in this FIFO specifies one set of results. The user has no direct access to this FIFO.

<table>
<thead>
<tr>
<th>Table 4 Attributes of the FIFOs used by the LCB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIFO name</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>REQUEST_QUEUE</td>
</tr>
<tr>
<td>RESULT_QUEUE</td>
</tr>
<tr>
<td>EVENT_QUEUE</td>
</tr>
<tr>
<td>REQUEST_BUFFER</td>
</tr>
<tr>
<td>RESULT_BUFFER</td>
</tr>
<tr>
<td>EVENT_BUFFER</td>
</tr>
<tr>
<td>PENDING_REQUESTS</td>
</tr>
<tr>
<td>PENDING_RESULTS</td>
</tr>
<tr>
<td>PENDING_EVENTS</td>
</tr>
</tbody>
</table>

1. In **bits**.
2. In **entries**.

### 1.3 Front-End Interface

#### 1.3.1 Command/Response

To be written.

#### 1.3.1.1 Sysclk signals

The LCB receives its clock externally. The clock operates at a nominal frequency of 20 MHZ. The signal which carries the clock is called SYSCLK. The SYSCLK wires are always connected to LVDS receivers. The SYCLK signals are redundant with only one of the two signals used at
any one time. Which particular signal (*primary* or *redundant*) is in use is determined by the value of the register described in Section 3.2.8.

1.3.1.2 Reset signals

Depending on whether the LCB is used as a commander or responder (see xxx), it may either *generate* a LAT reset signal or it may *respond* to a LAT reset request. The incoming signal is called \texttt{RESET\_IN} and outgoing signal \texttt{RESET\_OUT}. The \texttt{RESET\_IN} wires are always connected to LVDS receivers, and the \texttt{RESET\_OUT} wires are always connected to LVDS transmitters. Both the \texttt{RESET\_IN} and \texttt{RESET\_OUT} signals are redundant, and only one of the two signal pairs is used at any one time. Which particular pair (*primary* or *redundant*) is currently in use is determined by the value of the register described in Section 3.2.8.
1.3.1.3 Command/Response signals

Both the LATP_IN and LATP_OUT wires carry single cell LATp packets as described in [1]. The LATP_IN wires are always connected to LVDS receivers, and the LATP_OUT wires are always connected to LVDS transmitters. The usage of these wires by the LCB changes as a function of whether the LCB is used as a commander or as a responder (see xxx). If the LCB is used as:

**a commander:** The LATP_OUT wire is assumed to carry *command* packets, which are *transmitted* by the LCB. The LATP_IN wire is assumed to carry *response* packets, which are *received* by the LCB.

**a responder:** The LATP_IN wire is assumed to carry *command* packets, which are *received* by the LCB. The LATP_OUT wire is assumed to carry *response* packets, which are *transmitted* by the LCB.

Both the LATP_IN and LATP_OUT signals are redundant, and only one of the two signal pairs is used at any one time. Which particular pair (*primary* or *redundant*) is in use is determined by the value of the register described in Section 3.2.8.

**Figure 4** Processing of the Command/Response signals
1.3.2 Event

To be written.

1.4 Back-End Interface

To be written.

1.4.1 The Request Engine

To be written.
Figure 6  Relationship between intermediate FIFOs (normal transfer)

Figure 7  Relationship between intermediate FIFOs (aborted transfer)
1.4.2 The Event Engine

To be written.
1.5 Back-End Interface

The state machine represented as the Transfer Engine wakes up in response to one of three signals:

**Transfer request:** The PCI user has written to the REQUEST_QUEUE. (See Section 3.2.4.) The value written "points" to a list of transactions to be executed. The Transfer Engine moves (DMAs) this list from user PCI space to the REQUEST_BUFFER. When the transfer completes, the Transfer Engine writes the length and status of the transfer to the PENDING_REQUESTS FIFO.

**Transfer result:** The state machine represented as the Request Engine has completed the processing of a request-list. The results of this processing are contained in the RESULT_BUFFER. The length and processing status of the results were written to the PENDING_RESULT FIFO by the Request Engine. The Transfer Engine moves the result from the RESULT_BUFFER to a location in user PCI space. This location is given as the first item in the RESULT_BUFFER. When the transfer is complete, the Transfer Engine writes a description (see Section 4.4) to the RESULT_QUEUE and conditionally asserts a PCI interrupt.

**Transfer event:** The state machine represented as the Event Engine has received at least one of the packets corresponding to an incoming event. This packet is contained in the EVENT_BUFFER. A description of this packet was written to the PENDING_EVENT FIFO by the Request Engine. The Transfer Engine must move this packet from the EVENT_BUFFER to a location in user PCI memory space. The Transfer Engine determines this location by allocating from the LCB’s circular buffer. (See Section 3.2.2.) When the transfer is complete, the Transfer Engine writes a description (see Chapter 5) to the EVENT_QUEUE and conditionally asserts a PCI interrupt.

The Transfer Engine processes only one activity at a time. However, because these signals are developed completely independently of each other, they may occur simultaneously with
respect to the activity of the Transfer Engine. Therefore, a relative priority is assigned to each activity, allowing the Transfer Engine to prioritize simultaneous requests. These priorities are enumerated in Table 5, where the smallest value corresponds to the highest priority.

**Table 5** Scheduling priority

<table>
<thead>
<tr>
<th>Activity</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result transfer</td>
<td>1</td>
</tr>
<tr>
<td>Request transfer</td>
<td>2</td>
</tr>
<tr>
<td>Event transfer</td>
<td>3</td>
</tr>
</tbody>
</table>

The synthesis of the three different wake up signals is derived by combinatorial logic and the flags of the various FIFOs as illustrated in Figure 10. A description of the processing of the Transfer Engine in response to receiving each of these signals is given below.

![Diagram](image)

**Figure 10** Developing the signals used to activate the Transfer Engine

### 1.5.1 Transfer Request

**Triggered by:** The PCI user writes a *request-descriptor* to the REQUEST_QUEUE. (See Section 3.2.4.) This condition is summarized by the signal which specifies the queue is *not empty*. Note: If the queue is *full*, the write fails with a TARGET_ABORT error.
Constraints: The REQUEST_BUFFER must be at least three-quarters empty, and the PENDING_REQUESTS FIFO must not be full. The inhibit request transfer field of the CSR register must be clear. (See Section 3.2.1.)

Action: When activated:
— Read the REQUEST_QUEUE.
— Extract from the request-descriptor the request-list address and length.
— Write the request-address to the REQUEST_BUFFER.
— Construct the DMA request. Source address and transfer length are derived from the just read request-descriptor. The destination address is the REQUEST_BUFFER.
— Start the DMA and wait for its completion.
— Check for REQUEST_BUFFER overflow. This translates to a BUFFER_FULL error, to be set in the status descriptor.
— Write the status descriptor to the PENDING_REQUESTS FIFO. The structure of this descriptor is illustrated in Figure 11.

Figure 11 Structure of any one item in the pending requests FIFO

length: The size of the request-list corresponding to the specified request-descriptor. This list resides in the LCB’s REQUEST_BUFFER. The structure of a request-list is described in Section 4.2. The Transfer Engine has transferred this list from user memory into the REQUEST_BUFFER. The Request Engine will transfer this list out of the buffer. The length of this list is specified in units of 32-bit words. Note that the request-list is always preceded by its corresponding request-address. The Request Engine has the responsibility to copy this address to the RESULT_BUFFER as illustrated in Figure 7.

error: The value of field depends on the state of the flush field described below. If the flush field is clear, this field has a value of zero. If the flush field is set, this field enumerates the transfer error which caused the Transfer Engine to abort the transfer of the request-list into the REQUEST_BUFFER. The possible values of this field are specified in xxx. The Request Engine has the responsibility to copy this field to the PENDING_RESULTS FIFO as illustrated in Figure 6.

flush: This field, if set, indicates to the Request Engine that the request-list corresponding to the descriptor is incomplete. An incomplete list occurs when the transfer (by the Transfer Engine) from user memory to the RESULT_BUFFER fails. The reason the transfer failed is contained in the error field described above. The length field specifies how much of the of the list was transferred before
failure. The Request Engine has the responsibility, when it finds the flush field set, to read and throw away the incomplete list. Note that even in the case of a flush request, a request-address will precede the buffer to be flushed. This is illustrated in Figure 7.

1.5.2 Transfer Result

**Triggered by:** The Request Engine writes to the PENDING_RESULTS FIFO. (See Section 1.2.) This condition is summarized by the signal which specifies the FIFO is not empty.

**Constraints:** The RESULT_QUEUE must not be full. The inhibit result transfer field of the CSR register must be clear. (See Section 3.2.1.)

**Action:** When activated:

- Read one entry from the PENDING_RESULTS FIFO. The structure of any one entry of this FIFO is illustrated in Figure 12.
- Read one item from the RESULT_BUFFER.
- Test whether the flush field of the entry is set. If set, go to the alternate action described below.
- Construct the DMA request. The source address is the RESULT_BUFFER. The destination address corresponds to the item just read from the RESULT_BUFFER. The transfer length is extracted from the just read item from the PENDING_RESULTS FIFO.
- Start the DMA and wait for its completion.
- Check for RESULT_BUFFER underflow. This translates to a BUFFER_EMPTY error to be set in the result-descriptor.
- Check the DMA completion status. If the DMA failed, flush the data in the RESULT_BUFFER which remains to be transferred.
- Construct the result-descriptor. The address field corresponds to the destination address of the DMA transfer. The direction field is clear. The status field is derived from the status of the just completed DMA operation.
- Write the result-descriptor to the RESULT_QUEUE. (See Section 3.2.5.) If the act of writing to this queue causes the queue to go from empty to non-empty, assert an interrupt. (Interrupts are described in Section 1.5.4.)

**Alternate action:** When the flush field of the entry just read from the PENDING_RESULTS FIFO is set:

- Construct the result-descriptor. The address field corresponds to item just read from the RESULT_BUFFER. The direction field is set. The status field is derived from the error field of the item just read from the PENDING_RESULTS FIFO.
— Write the result-descriptor to the RESULT_QUEUE. If the act of writing to this queue causes the queue to go from empty to non-empty, assert an interrupt. (Interrupts are described in Section 1.5.4.)

![Figure 12 Structure of any one item in the pending results FIFO](image)

- **length:** The size of the result-list corresponding to the descriptor. This list resides in the LCB’s RESULT_BUFFER. The structure of a result-list is described in Section 4.4. The Request Engine has the responsibility to construct and write this list to the RESULT_BUFFER. The Transfer Engine is responsible for transferring this list from the RESULT_BUFFER into user memory. The length of this list is specified in units of 32-bits words. Note that the result-list is always preceded by its corresponding result-address. The Request Engine has the responsibility to copy this address from its corresponding request-list as illustrated in Figure 6.

- **error:** The value of field depends on the state of the flush field described below. If the flush field is clear, this field has a value of zero. If the flush field is set, this field enumerates the transfer error which caused the Transfer Engine to abort the transfer of the corresponding request-list into the REQUEST_BUFFER. The possible values of this field are specified in xxx. The Request Engine has the responsibility to copy this field from the PENDING_REQUESTS FIFO as illustrated in Figure 7.

- **flush:** This field, if set, indicates to the Transfer Engine that the request-list corresponding to the descriptor was flushed by the Request Engine. (See Section 1.4.) In such a case only the corresponding result-address will be on the RESULT_BUFFER and the length field will have a value of zero. This is illustrated in Figure 7.

### 1.5.3 Transfer Event

**Triggered by:** The Event Engine writes to the PENDING_EVENTS FIFO. (See Section 1.2.) This condition is summarized by the signal which specifies the FIFO is not empty.

**Constraints:** The EVENT_QUEUE must not be full. The inhibit event transfer field of the CSR register must be clear. (See Section 3.2.1.)

**Action:** When activated:

— Read one entry from the PENDING_EVENTS FIFO. The structure of any one entry of this FIFO is illustrated in Figure 13.
— Determine the event packet size. Extract the requested transfer length from the just read item. Add eight (8) to this value to account for the obligatory user header prefixed to each event. (See Chapter 5.) Retrieve the current value of the WRITE pointer of the LCB’s circular buffer. This will be the event address. Take the difference between event address and the READ pointer. This is the amount of memory remaining in the circular buffer. Compare the remaining amount to the event packet size. If the event packet size is greater then the remaining amount, go to the alternate action described below.

— Construct the DMA request. The destination address is derived by summing the event address plus four (as the transfer begins just past the user header) with the circular buffer base. (See Section 3.2.2.) The source address is the EVENT_BUFFER. The transfer length is the event packet size less the four (4) words allocated to the header.

— Start the DMA and wait for its completion.

— Check for EVENT_BUFFER underflow. This translates to a BUFFER_EMPTY error to be set in the event-descriptor.

— Check the DMA completion status. If the DMA failed, flush the data in the EVENT_BUFFER which remains to transfer.

— Construct the event-descriptor. This structure is described in Chapter 5. The offset field will contain the event address. The length field is equal to the actual number of words transferred plus the eight (8) word prefix. Nominally (an event with no transfer errors), this value is simply the event packet size. The error field has two components: the receive status and the transfer status. The transfer status is a copy of the status returned by the just completed DMA operation (and potentially, BUFFER_EMPTY) and the receive status is a copy of the receive error field of the item just read from the PENDING_EVENTS FIFO.

— Update the circular buffer’s WRITE pointer. This update must take into account the eight words allocated to the header. Additionally, the update must also take into account only those words actually transferred, which may be different than the requested transfer size, as, for example, in the case of a DMA error.

— Write the event-descriptor to the EVENT_QUEUE. Conditionally, assert an interrupt depending on the state of the EVENT_QUEUE and the circular buffer, and how the user has configured the interrupt conditions for the LCB. (Interrupts are described in Section 1.5.4.)

Alternate action: When the size of the packet is larger than the amount of memory available in the circular buffer:

— Flush the entire packet in the EVENT_BUFFER.
— Construct the event-descriptor. The offset field is set to zero. The length field is set to the requested event packet size. The error field has two components: the receive status and the transfer status. The transfer status is set to a value of INSFMEM (7) and the receive status is a copy of the receive error field.

— Write the event-descriptor to the EVENT_QUEUE. Conditionally, assert an interrupt depending on the state of the EVENT_QUEUE and the circular buffer, and how the user has configured the interrupt conditions for the LCB. (Interrupts are described in Section 1.5.4.)

![Figure 13 The pending event descriptor](image)

**length:** The size of any one packet\(^1\) of an event. The packet’s payload resides in the LCB’s EVENT_BUFFER. The Event Engine will construct and write a packet’s payload into the EVENT_BUFFER. The Transfer Engine moves the packet payload from the EVENT_BUFFER into user memory. The length of a packet is specified in units of 32-bit words.

**receive error:** An enumeration summarizing whether the Event Engine discovered an error when receiving the incoming event packet. If this field is zero, the engine received the packet successfully. If the field is non-zero, its value enumerates why the transfer failed. The reasons are enumerated and described in Section 5.3.1.

### 1.5.4 Interrupts

Functionally, the LCB is designed to generate an interrupt whenever either a result or an event is ready to be presented to the user. The LCB uses signals from four different sources in determining if one of these conditions are meet. These sources and their respective signals include:

i. The RESULT_QUEUE is not empty.

ii. The event circular buffer is full.

iii. The EVENT_QUEUE exceeds some predefined high-water mark.

iv. The System Clock (SYSCLK) changes state from either: off to on, or on to off.

The event queue defines four different high-water levels:

---

1. Actually, its payload.
The queue is not empty.
ii. The occupancy of queue exceeds 25%.
iii. The occupancy of the queue exceeds 50%.
iv. The occupancy of the queue exceeds 75%.

The user has the ability to select which one of the four high-water marks will be used as input in generating an interrupt. The interface for this selection is contained in the CSR register described in Section 3.2.1. The three different signals (properly selected, in the case of the event buffer and queue) are then simply ORed together to form the actual interrupt signal. This signal is then used to drive two sinks:

- The PCI interrupt “A” line
- A user defined I/O line

All the logic necessary to form and process an interrupt is summarized in Figure 14. Note that once asserted, the interrupt signal remains asserted until the condition that caused the assertion is removed. For example, if the interrupt was asserted because the result queue transits from empty to not empty, an interrupt will remain asserted while the queue remains not empty.

![Figure 14 - Developing the signals used to generate an interrupt](image-url)
Chapter 2

Front-End Interface

2.1 Introduction

This chapter describes the registers used to configure and maintain the state of the Front-End Interface of the LCB. Front-End registers are all thirty-two bits wide and the fields of these registers fit into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated greyed 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 greyed-out with their 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).

Registers of this interface are accessed by the Back-End (PCI) Interface using the request/result paradigm described in Section 4.3.2 and Section 4.6.3. The CSR register described in Section 2.2 may also be accessed through the command/response fabric as described in the following section.
2.1.1 Accessing the Front-End CSR through the Command/Response fabric

Any responder, in order to participate on the command/response fabric, must allow for its LATp address and its current path (Primary/Redundant) to be set by the fabric commander. As the LATp address is specified by the board ID field of the Front-End CSR (see Section 2.2), this implies external access of this register by a commander. A commander access this register by issuing a command/response packet with the structure illustrated in Figure 15.

The LCB listens for this particular command packet on both its primary and redundant path, independent of which path is currently selected (see Section 3.2.8). Whichever path the command is received on becomes the new currently selected path. For example, if the LCB is currently operating on the primary path and receives this command packet on the redundant path, the current path becomes the redundant path. If the LCB receives this command packet on the same path as is currently selected, the current path is left unchanged.

where:

**LATp header:** The respond field must be *clear*. The destination address must be the broadcast address (0x3F) and the protocol field must be three (3).

**register value:** The value to be written to the Front-End CSR. The fields which may be changed by this command packet are specified by Section 2.2. Note: Only those fields specified by the *field mask* will be changed. See below.

**field mask:** A 32-bit mask in which the value of each one-bit field is used to determine whether or not the corresponding field in the CSR will be changed. If a field is *set*, the corresponding field in the CSR is changed. The new value of the field is found in the corresponding bit field of the *register mask* (see above). If a field is *clear*, the corresponding field in the CSR is left unchanged.

![Figure 15 Command Packet used to access Front-End CSR](image)
2.2 Control and Status Register (CSR)

Note: This register should not be confused with the registers of similar name, which occupy PCI configuration and memory space.

This register is used to configure and monitor the state of the LCB’s Front-End Interface. This register may be accessed from PCI using the LCB’s transaction engine as described in Section 4.3.2. If the LCB is a slave on the fabric, this register may be accessed from the Command/Response fabric by a command, whose format and structure is described in Section 2.1.1. The fields of this register are illustrated in Figure 16. Note, that any field marked user defined has no predefined function and its usage is determined by application. Such fields may be both read and written.

Note: This register does not follow the usual rules on reset. There are three specific cases:

- power-on reset.

and on reset have an initial value of zero (0).

![Figure 16 Front-End CSR Register](image)

**event enable**: This field enables or disables the reception of event data into the LCB. (See Chapter 5.) If the field is set, unsolicited event packets will be accepted. If the field is clear, any unsolicited event packets, if sent to the LCB, will be discarded.

**clock on trailing edge (incoming)**: This field determines whether the LCB clocks its incoming signals on either the rising or trailing edge of SYSCLK. If the field is clear, the LCB registers incoming signals on the clock’s leading edge. If this field is set, the LCB registers all incoming signals on the clock’s trailing edge. Normally, and by convention the LCB registers all incoming signals on the clock’s rising edge. In
some instances, for example, where the length of cable between clock source and the LCB’s incoming signals is sufficient to skew these signals, using the leading edge may be inappropriate. In such a case, one may compensate for this skew by setting this field.

**clock on trailing edge (outgoing):** This field determines whether the LCB clocks its outgoing signals on either the rising or trailing edge of SYSCLK. If the field is clear, the LCB registers outgoing signals on the clock’s leading edge. If this field is set, the LCB registers all outgoing signals on the clock’s trailing edge. Normally, and by convention the LCB registers all outgoing signals on the clock’s rising edge. In some instances, for example, where the length of cable between clock source and the LCB’s outgoing signals is sufficient to skew these signals, using the leading edge may be inappropriate. In such a case, one may compensate for this skew by setting this field.

**use even header parity:** This field determines whether header packet parity (see [1]) on outgoing commands or events should be odd or even. The LATp protocol specifies that any computed parity should be odd. If this field is set, header parity will be even. This field is designed to test parity error checking at a packet destination; consequently, the field should never be set in normal operation.

**use even payload parity:** This field determines whether packet cell parity (see [1]) on outgoing commands or events should be odd or even. The LATp protocol specifies that any computed parity should be odd. If this field is set, cell parity will be even. This field is designed to test parity error checking at a packet destination; consequently, the field should never be set in normal operation.

**assert back-pressure:** This field is used to artificially generate event fabric back-pressure. Asserting this field causes the LCB to generate back-pressure to the Event-Builder after receiving the header of the next properly addressed command cell on the event fabric. That is, on the arrival of an event packet. Once this event occurs, the field is cleared. The back-pressure remains asserted until the end of the current incoming packet is reached, i.e., until the end of the cell which is not immediately followed by another data cell. This field is a test-feature and is used to test the Event-Builder’s response to back-pressure.

**board ID:** This field contains the LCB’s node number. This value will be used to construct the LATp source address field of each packet transmitted by the LCB. (See Section 4.3.1.1.) This value must be unique for all boards on both the command and event fabrics. As this board (from the perspective of LATp) is a master (see [1]), its high-order bit is always set. Consequently, only five bits of this six-bit field are writable, with the high-order bit read-only and always returning the value one (1). Note: This field persists over an LCB reset. That is, it will not be set to zero on reset. In order to reset this field the LCB must be power cycled. This behaviour is independent of how the LCB was reset, either through a PCI_RESET on the bus, or accessing the

**byte-wide LATp:** This field determines whether the incoming events are decoded using bit-wide or byte-wide LATp. (See [1].) If this field is clear, the LCB expects that all incoming event packets are produced with bit-wide LATp. If this field is set, the LCB expects that all incoming event packets are produced with byte-LATp.
**crate reset:** Generates a Power-On-Reset (POR) signal which is carried on the cPCI backplane. On a PMC version of the LCB, asserting this field has no effect. Normally this signal is used to reset the crate’s processor and consequently, the crate itself. When this field is asserted (set), the POR signal is generated. The POR signal remains asserted for 256 (decimal) clock tics, where, one clock tic is equal to 30.72 micro-seconds (approximately 7.5 milli-seconds).

**event transmission paused:** Monitors the transitory state of the flow-control line used for *out-going* data on the event fabric. While this field is set, flow-control is being asserted and out-going events cannot be sent by the LCB. This field being permanently asserted, normally indicates a problem with the Event Builder. Note: This field is *read-only*.

**Front-End version:** The version number of the FPGA which implements the functionality of the LCB’s Back-End. Note: This field is *read-only*.

### 2.3 FIFO faults register

This register latches any errors encountered by the LCB when accessing its internal FIFOs. On read access, this corresponds to the appropriate FIFO being *empty* when read. For a write operation, this corresponds to the FIFO being *full* when written. For any one of these internal FIFOs, the LCB records whether a read or write fault occurred in the structure illustrated by Figure 17, where any one of the two fields, if set, constitutes a fault. This register may be accessed from PCI using the LCB’s transaction engine as described in Section 4.3.2.

![Figure 17 Structure of a FIFO fault](image)

**Figure 17** Structure of a FIFO fault
The `FIFO_FAULTS` register contains five of these structures corresponding to the five different internal FIFOs as illustrated in Figure 18. Writing to this register will reset its contents (to zero), independent of the value requested to be written.

Note that the design of the LCB is such that these FIFOs should never fault. Consequently, any fields of this register found set constitute a serious, non-recoverable error and point to a hardware design error and thus, should, in practice, never occur.

**2.4 Event Statistics**

Event packets received by the LCB follow the LATp protocol described in [1]. The protocol requires the LCB to maintain a set of well-defined statistics. These statistics are contained in the register described below in Figure 19.

The accumulated statistics may be reset to zero by writing to this register. The value to be written is ignored.

**packets received:** The number of packets successfully received by the LCB. Success is defined as a packet which had no header parity errors and whose address either matches the LCB’s own, broadcast address, or round-robin address. Note: this includes any packets which contain data cell parity errors (see below). This counter is saturating, which implies that when the counter overflows, it stops incrementing.
**data cell parity error:** This field summarizes any cell parity errors discovered by the LCB when receiving event packets. If this field is *set*, the LCB has discovered at least one data cell parity error since the field was reset. Note: packets with data cell parity errors will still increment the *packets received* counter.

**header parity error:** This field summarizes any packet header parity errors discovered by the LCB when receiving event packets. Such packets are discarded by the LCB. If this field is *set*, the LCB has discovered at least one packet header parity error since the field was reset. Note: packets with a header parity error do not increment the packets received counter (see above).

**invalid address:** This field summarizes any packets received by the LCB which have correct header parity, but whose destination address does not match either the LCB’s own, broadcast address, or round-robin address. Such packets are discarded by the LCB. If this field is *set*, the LCB has discovered at least one packet incorrectly routed since the field was reset. Note: packets with an incorrect destination address do not increment the packets received counter (see above).
Chapter 3

PCI Interface

This chapter describes the PCI addressable storage of the LCB. The LCB contains addressable storage in two different PCI spaces:

- Configuration
- Memory

The module does not define any storage in I/O space. For Configuration and memory space, storage is represented by registers. Registers are all thirty-two bits wide. Fields of a register fit into one of three classes:

**Not defined:** Undefined fields are identified as Must Be Zero (MBZ) and are illustrated greyed 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 with their 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).
### 3.1 Configuration space

The PCI specification requires a 64-byte configuration header to define various generic attributes of the LCB. This configuration header is illustrated and described in Figure 20. Many of the values of the configuration header are driven by the so-called “CorePCI Customization constants.” (See [2].) The values for these constants are enumerated in Table 6.

![Figure 20: PCI configuration header for LCB](image)

**Vendor ID:** Determined by the CorePCI constant USER_VENDOR_ID described within Table 6.

**Device ID:** Determined by the CorePCI constant USER_DEVICE_ID described within Table 6.

**Command:** See the register definition 3.1.1.

**Status:** See the register definition 3.1.2.

**Revision ID:** See the register definition 3.1.3.

**Class Code:** The class code is made up of three eight-bit fields. These fields are determined (from low-order field to high-order field) from the CorePCI constants USER_PROGRAM_IF, USER_SUB_CLASS, and USER_BASE_CLASS described within Table 6.

**Cache Line Size:** To be defined.

**Latency Timer:** To be defined.
**Header Type:** Set to a value zero (0), specifying that the LCB has a standard Device Configuration header.

**BIST:** Set to a value of zero (0), specifying the LCB does not support Built-in Self Test.

**Base Address Register 0:** The LCB implements 16 kilobytes of memory space. This register specifies the starting PCI address and size of this space. For a discussion of how this space is used, see Section 3.2.4. Note that appropriate constants in Table 6 are used to set the size of this space.

**Subsystem Vendor ID:** Determined by the CorePCI constant USER_SUBVENDOR_ID described within Table 6.

**Subsystem ID:** Determined by the CorePCI constant USER_SUBSYSTEM_ID described within Table 6.

**Capabilities Pointer:** Set to zero (0).

**Interrupt Line Register:** The LCB can generate a PCI interrupt request. This register is simply the R/W scratch register demanded by the PCI bus specification.

**Interrupt Pin:** The LCB can generate a PCI interrupt request. This register has a value of 01, indicating it uses a single interrupt and the interrupt line is INTAn.

**Min_Gnt:** To be defined.

**Max_Lat:** To be defined.

**Interrupt Control/Status Register:** See the register definition 3.1.3.
### Table 6: CorePCI customization constants for LCB

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_DEVICE_ID</td>
<td>0845²</td>
</tr>
<tr>
<td>USER_VENDOR_ID</td>
<td>11AA³</td>
</tr>
<tr>
<td>USER_REVISION_ID</td>
<td>see 3.1.3</td>
</tr>
<tr>
<td>USER_BASE_CLASS</td>
<td>07⁴</td>
</tr>
<tr>
<td>USER_SUB_CLASS</td>
<td>80</td>
</tr>
<tr>
<td>USER_PROGRAM_1F</td>
<td>00</td>
</tr>
<tr>
<td>USER_SUBSYSTEM_ID</td>
<td>00⁵</td>
</tr>
<tr>
<td>USER_SUBVENDOR_ID</td>
<td>00</td>
</tr>
<tr>
<td>BIT_64</td>
<td>00</td>
</tr>
<tr>
<td>DMA_CNT_EN</td>
<td>01</td>
</tr>
<tr>
<td>DMA_IN_IO</td>
<td>N/A</td>
</tr>
<tr>
<td>MADDR_WIDTH</td>
<td>0F⁶</td>
</tr>
<tr>
<td>BAR1_ENABLE</td>
<td>00⁷</td>
</tr>
<tr>
<td>BAR1_IO_MEMORY</td>
<td>00</td>
</tr>
<tr>
<td>BAR1_ADDR_WIDTH</td>
<td>00</td>
</tr>
<tr>
<td>BAR1_PREFETCH</td>
<td>N/A</td>
</tr>
<tr>
<td>HOT_SWAP_EN</td>
<td>00</td>
</tr>
</tbody>
</table>

1. Expressed in units of hexadecimal.
2. Allocated and assigned by ACTEL.
3. Corresponds to ACTEL’s vendor ID.
4. Simple Communications Controller.
5. Specifies that this optional register (and the USER_SUBVENDOR_ID register) is not used.
6. Corresponds to 16 kilobytes of memory space
7. Specifies that BAR1 register is not used
3.1.1 Command Register

![Figure 21 Command Register of the LCB’s PCI Configuration Header](image)

3.1.2 Status Register

Fields which are read/write can *only* be cleared. To clear a read/write field, the field must be *asserted*.

![Figure 22 Status Register of the LCB’s PCI Configuration Header](image)

3.1.3 Revision ID register

The structure of the CorePCI constant `USER_REVISION_ID` enumerated within Table 6 is illustrated in Figure 23. The *type* field should be an enumeration of the environment the
board will be used in, for example, one value for the engineering version and a different value for a flight qualified board.

![Figure 23 Revision ID Register of the LCB’s PCI Configuration Header](image)

### 3.2 Memory space

The LCB’s registers for both functional configuration and control are located in PCI memory space. The base address used to locate these registers is specified by the BAR1 register of the PCI Configuration Header discussed in Section 3.1. The list of registers and their respective byte offsets from this base address are given by Table 7:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Offset^1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>00</td>
</tr>
<tr>
<td>EVENTS_BASE</td>
<td>04</td>
</tr>
<tr>
<td>EVENTS_FREE</td>
<td>08</td>
</tr>
<tr>
<td>REQUEST_QUEUE</td>
<td>0C</td>
</tr>
<tr>
<td>RESULT_QUEUE</td>
<td>10</td>
</tr>
<tr>
<td>EVENT_QUEUE</td>
<td>14</td>
</tr>
<tr>
<td>INTERRUPT_STATE</td>
<td>18</td>
</tr>
<tr>
<td>FABRIC_SELECT</td>
<td>1C</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

---

^1. Expressed as a byte offset from the LCB’s memory base in units of hexadecimal
3.2.1 Control and Status Register (CSR)

Note: This register should not be confused with the registers of similar name which occupy PCI configuration space and the Front-End Interface.

![Figure 24 Command and Status Register (CSR)]

**inhibit request transfer:** This field is for testability only and in normal operation should always be clear. If set, the Transfer Engine will not process requests (originating from the user) to transfer a request-list. (See Section 1.5.1.) This function allows testing of flow-control between Front-End and Back-End interfaces.

**inhibit result transfer:** This field is for testability only and in normal operation should always be clear. If set, the Transfer Engine will not process requests (originating from the Request engine) to transfer a result-list to user PCI memory. (See Section 1.5.2.) This function allows testing of flow-control between Front-End and Back-End interfaces.

**inhibit event transfer:** This field is for testability only and in normal operation should always be clear. If set, the Transfer engine will not process requests (originating from the Event Engine) to transfer event data to user PCI memory. (See Section 1.5.3.) This function allows testing of flow-control between Front-End and Back-End interfaces.

**interrupt condition (event queue):** This field enumerates which of four possible conditions occurring within the EVENT_QUEUE will cause an external interrupt to be asserted. These conditions and the field values necessary to enable the corresponding condition are enumerated in Table 8. Once active, an interrupt will continue to be asserted until the condition which triggered the interrupt is removed. A more detailed discussion of the role of interrupts within the LCB is found in Section 1.5.4.

**SYSCLK:** This field specifies whether or not the System Clock (nominally 20 MHz) is present within the LCB. The LCB Front-End Interface cannot function without the presence of SYSCLK. If the field is set, the clock is present. If the field is clear, the clock is not present. Note: This field is Read-Only.
reset: This field is used to reset the LCB. *Asserting* this field will initiate the reset. The field will always read back *clear*. A reset initiated through this register is identical to the reset initiated by issuing a `PCI_RESET` signal on the bus, with the exception that the ACTEL PCI core is itself, not reset.

### Table 8 Interrupt conditions for event queue

<table>
<thead>
<tr>
<th>Field value</th>
<th>LCB asserts interrupt while event queue:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>At least 75% occupied</td>
</tr>
<tr>
<td>1</td>
<td>At least 50% occupied</td>
</tr>
<tr>
<td>2</td>
<td>At least 25% occupied</td>
</tr>
<tr>
<td>3</td>
<td>Not empty</td>
</tr>
</tbody>
</table>

#### 3.2.2 Circular buffer

All event data are transferred by the LCB (through the Transfer Engine, see Section 1.5) into external user PCI memory. As events arrive unsolicited and are transferred asynchronously with respect to their processing by the user, both LCB and user must agree on a memory management model. The model consists of a simple *circular* buffer, from which the LCB allocates memory and to which the user frees memory. That is, the LCB maintains the circular buffer’s *write* pointer and the user maintains its *read* pointer. The circular buffer is a fixed 512 kilobytes long and may be located on any 1 megabyte PCI memory boundary. The base address of a circular buffer is determined by the value of the `EVENTS_BASE` register.

“Pointers” are expressed as 32-bit word *offsets* from the circular buffer’s base. In order to simplify the memory manager implementation and prevent split buffers, an additional amount of “guard” memory beyond the 512 kilobyte limit is also required. The size of this guard area is equal to the largest allowed transfer, which is 4 kilobytes. Therefore, the total amount of user memory reserved for the circular buffer must be 512 + 4 kilobytes, or 516 kilobytes. The basic arrangement of the circular buffer is illustrated in Figure 25.
Chapter 3 PCI Interface

The configuration and control of the circular buffer is distributed over three registers:

i. A base register (see Figure 26) used to locate the starting address of the circular buffer. This register may be used to specify any one (1) megabyte aligned boundary of the 32-bit PCI memory address space.

ii. An allocation register. This register maintains the write pointer. This register is internal to the LCB and is not directly exposed to the user. On event arrival, the LCB uses this register to assign a buffer for the arrived event. The address of this buffer is communicated to the user through the EVENT_QUEUE as described in Chapter 4.

iii. A deallocation register. (See Figure 27.) This register maintains the read pointer. It is the responsibility of the user to update the read pointer as appropriate and thus “free” an event.

Further discussion of the usage of the LCB’s circular buffers is the subject of Chapter 4.

**Figure 25** LCB Circular Buffer

**Figure 26** EVENTS_BASE register

**base address:** This field specifies the base PCI memory address of the circular buffer. In actual fact, only the upper-order 12 bits of the address, as the circular buffer, must be aligned on a 1-megabyte boundary, and therefore, the low-order 20 bits of the address are an implied zero. The value of this register may only be changed if three constraints are meet:
— The \textit{enable events} field of the CSR register must be \textit{clear}. (See Figure 24.)
— The (internal) \textit{pending events} FIFO must be \textit{empty}. (See Figure 1.)
— The EVENT\_QUEUE must be \textit{empty}. (See Figure 1.)

If any of these constraints are violated, the LCB will terminate user access with a TARGET\_ABORT error.

### 3.2.3 Circular Buffer Deallocation register

![Figure 27] EVENTS\_FREE register

\textbf{read offset:} This field specifies the current value of the circular buffer’s read pointer. (See Section 3.2.2.) The read pointer is expressed as a 32-bit word offset from the base of the circular buffer. For a discussion on how this register must be updated see Chapter 4.

### 3.2.4 Request queue

The REQUEST\_QUEUE buffers (in strict FIFO order) any transaction \textit{request} queued by the user to the LCB. (See Chapter 4 and the register described in Section 3.2.5.) PCI access to this queue is determined by the rules specified in Table 9:

<table>
<thead>
<tr>
<th>Access request</th>
<th>Queue state</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>\textit{not full}</td>
<td>Returns the value \texttt{00000000} (hexadecimal)</td>
</tr>
<tr>
<td>READ</td>
<td>\textit{full}</td>
<td>Returns the value \texttt{FFFFFFFF} (hexadecimal)</td>
</tr>
<tr>
<td>WRITE</td>
<td>\textit{not empty}</td>
<td>Enqueues a \textit{request-descriptor} (see Section 4.1)</td>
</tr>
<tr>
<td>WRITE</td>
<td>\textit{full}</td>
<td>Generates a \textit{TARGET_ABORT} error</td>
</tr>
</tbody>
</table>

### 3.2.5 Result queue

The RESULT\_QUEUE buffers (in strict FIFO order) the \textit{result} of any transaction queued by the user to the LCB. (See Chapter 4 and the register described in Section 3.2.4.) PCI access to this queue is determined by the rules specified in Table 10:
3.2.6 Event queue

The EVENT_QUEUE buffers (in strict FIFO order) descriptions corresponding to any incoming events received by the LCB. (See Chapter 5.) PCI access to this queue is determined by the rules specified in Table 11:

<table>
<thead>
<tr>
<th>Access request</th>
<th>Queue state</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>not empty</td>
<td>Dequeues and returns a result-descriptor (see Section 4.4)</td>
</tr>
<tr>
<td>READ</td>
<td>empty</td>
<td>Returns the value FFFFFFF7 (hexadecimal)</td>
</tr>
<tr>
<td>WRITE</td>
<td>don’t care</td>
<td>Generates a TARGET_ABORT error</td>
</tr>
</tbody>
</table>

### Table 10  PCI access rules for the RESULT_QUEUE

<table>
<thead>
<tr>
<th>Access request</th>
<th>Queue state</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>not empty</td>
<td>Dequeues and returns a result-descriptor (see Section 4.4)</td>
</tr>
<tr>
<td>READ</td>
<td>empty</td>
<td>Returns the value 38000000 (hexadecimal)</td>
</tr>
<tr>
<td>WRITE</td>
<td>don’t care</td>
<td>Generates a TARGET_ABORT error</td>
</tr>
</tbody>
</table>

3.2.7 Interrupt State

This register is used to control and monitor the various conditions under which the LCB can assert an interrupt. These conditions are discussed in detail within Section 1.5.4. In general, for each kind of interrupt the register satisfies three functions:

i. The ability to enable or disable the interrupt. In order to deal with each kind of interrupt in an orthogonal fashion, this function is implemented as a set/reset register, with separate fields for enable and disable. On write, unless the specified field value is explicitly set, the value of the corresponding field is left unchanged.

ii. To monitor whether or not the interrupt is currently asserted.

iii. To de-assert or acknowledge the interrupt. In many cases, de-asserting the interrupt is accomplished by removing the condition which caused the interrupt. For those interrupts for which this is not the case, the interrupt can be acknowledged by explicitly setting an appropriate field.
enable result interrupt: This field allows the LCB to assert an interrupt when its result queue changes from empty to non-empty. When the value of the field is TRUE (1), the interrupt is enabled. When the value of the field is FALSE (0) the interrupt is disabled. Writing a TRUE (1) value to this field will enable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To disable the interrupt see the “disable result interrupt” field below.

enable event interrupt: This field allows the LCB to assert an interrupt when one or more events are available for processing. The definition of availability is discussed in Section 1.5.4. When the value of the field is TRUE (1), the event interrupt is enabled. When the value of the field is FALSE (0) the interrupt is disabled. Writing a TRUE (1) value to this field will enable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To disable the interrupt see the “disable event interrupt” field below.

enable SYSCLK “on” interrupt: This field allows the LCB to assert an interrupt when the system clock (SYSCLK) which was absent, becomes present. When the value of the field is TRUE (1), the interrupt is enabled. When the value of the field is FALSE (0) the interrupt is disabled. Writing a TRUE (1) value to this field will enable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To disable the interrupt see the “disable SYSCLK “on” interrupt” field below.
enable SYSCLK “off” interrupt: This field allows the LCB to assert an interrupt when the system clock (SYSCLK) which was present, becomes absent. When the value of the field is TRUE (1), the interrupt is enabled. When the value of the field is FALSE (0) the interrupt is disabled. Writing a TRUE (1) value to this field will enable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To disable the interrupt see the “enable SYSCLK “off” interrupt” field below.

disable result interrupt: This field disables the interrupt associated with the result queue changing from empty to non-empty. Reading the value of this field always returns FALSE (0). Writing a TRUE (1) value to this field will disable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To enable the interrupt see the “enable result interrupt” field above.

disable event interrupt: This field disables the interrupt asserted whenever one or more events are available for processing. The definition of availability is discussed in Section 1.5.4. Reading the value of this field always returns FALSE (0). Writing a TRUE (1) value to this field will disable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To enable the interrupt see the “enable event interrupt” field above.

disable SYSCLK “on” interrupt: This field disables the interrupt associated with the system clock (SYSCLK) changing from absent to present. Reading the value of this field always returns FALSE (0). Writing a TRUE (1) value to this field will disable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To enable the interrupt see the “enable SYSCLK “on” interrupt” field above.

disable SYSCLK “off” interrupt: This field disables the interrupt associated with the system clock (SYSCLK) changing from present to absent. Reading the value of this field always returns FALSE (0). Writing a TRUE (1) value to this field will disable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To enable the interrupt see the “enable SYSCLK “off” interrupt” field above.

result interrupt pending: This field reflects the current state of the interrupt asserted whenever one or more events are available for processing. The definition of availability is discussed in Section 1.5.4. When the value of this field is set, the interrupt is pending. When the value of the field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable result interrupt” and the “disable result interrupt” fields described above. Note: This field is Read-Only. The interrupt condition is removed whenever there are no longer events available to process. For example, by reading the result queue until it is empty.

event interrupt pending: This field reflects the current state of the interrupt associated with the event queue changing from empty to non-empty. When the value of this field is set, the interrupt is pending. When the value of the field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable event interrupt” and the “disable event interrupt” fields described above. Note: This field is Read-Only. The interrupt condition is removed by reading the event queue until it high water mark condition is no longer true and the circular buffer is no longer full.
**SYSCLK “on” interrupt pending:** This field reflects the current state of the interrupt associated with the system clock changing from absent to present. When the value of this field is set, the interrupt is pending. When the value of this field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable SYSCLK on interrupt” and the “disable SYSCLK on interrupt” fields described above. To de-assert (acknowledge) the interrupt, the field must be written with a TRUE (1) value.

**SYSCLK “off” interrupt pending:** This field reflects the current state of the interrupt associated with the system clock changing from present to absent. When the value of this field is set, the interrupt is pending. When the value of this field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable SYSCLK off interrupt” and the “disable SYSCLK off interrupt” fields described above. To de-assert (acknowledge) the interrupt, the field must be written with a TRUE (1) value.

**enable event buffer interrupt:** This field allows the LCB to assert an interrupt when the event circular buffer is full (see xxx for more information). When the value of the field is TRUE (1), the interrupt is enabled. When the value of the field is FALSE (0) the interrupt is disabled. Writing a TRUE (1) value to this field will enable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To disable the interrupt see the “disable event buffer interrupt” field below.

**disable event buffer interrupt:** This field disables the interrupt associated with the event circular buffer going full. Reading the value of this field always returns FALSE (0). Writing a TRUE (1) value to this field will disable the interrupt. Writing a FALSE (0) value leaves the field unchanged. To enable the interrupt see the “enable event buffer interrupt” field above.

**event buffer interrupt pending:** This field reflects the current state of the interrupt associated with the event buffer going full. When the value of this field is set, the interrupt is pending. When the value of the field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable event buffer interrupt” and the “disable event buffer interrupt” fields described above. Note: This field is Read-Only. The interrupt condition is removed by returning enough memory to the circular buffer, such that the circular buffer is no longer full.

**event queue interrupt pending:** This field reflects the current state of the interrupt associated with the event queue going full. When the value of this field is set, the interrupt is pending. When the value of the field is clear, the interrupt is not pending. To either enable or disable this interrupt, see both the “enable event queue interrupt” and the “disable event queue interrupt” fields described above. (see high water mark)

### 3.2.8 Fabric Select

Both fabrics (command/response and event) used by the LCB to send and receive information are duplicated to mitigate against single-point failure. One pair is referred to as the primary fabric pair and the other as the redundant fabric pair. The LCB has the option of attaching to either the primary or the redundant pair. This one-bit register reflects which pair is currently used by the LCB. If the field of this register is set, the redundant pair is used by the LCB. If this
field is clear, the primary pair is used by LCB. The LCB provides two different mechanisms to select which pair:

i. Writing to the register described in this section. If this register is set, the redundant pair will be selected. If this register is cleared, the primary pair will be selected.

ii. Accessing the Front-End CSR through the command/response fabric (see Section 2.1.1). If the CSR is accessed through the redundant fabric, the redundant pair will be selected. If the CSR is accessed through the primary fabric, the primary pair will be selected.

Independent of which mechanism is used, this register will reflect which pair (primary or redundant) is currently selected.

![Fabric Select Register](image)

**Figure 29** Fabric Select Register
Chapter 4
Transactions

One of the two principal functions of the LCB is to, under user control, transmit or export data packets to the various nodes of the LAT’s command or event fabric. As is standard practice for the LCB, any transaction initiated by the user of the LCB will generate a result, and because transactions involving the events are data, is said to solicit results.

A destination node may be a electronics module on the command fabric (for example, one or more TEMs), or it may be an LCB on the event fabric. The LCB allows the user to bundle transactions together and to thus queue a set of transactions to the LCB for later execution. A bundled set of requests is called a request-list. Note that the LCB allows for more than one request-list to be queued simultaneously. There are four fundamental structures involved in using the request capability of the LCB:

- **request-list**: Principally a specification of the packets to be transmitted by the LCB. This specification includes, for each packet: its destination, payload, whether or not it requires a response, and if so, a timeout period. As each packet will cause the LCB to emit a result, a request-list also specifies the address of a buffer to receive the result-list. (See below.) The request-list is discussed in detail in Section 4.2.

- **request-descriptor**: Specifies the address and size of a request-list. By writing this structure to the REQUEST_QUEUE, a request-list is queued to the LCB for execution. The request-descriptor is discussed in more detail in Section 4.1.

- **result-list**: As the LCB processes each item of a request-list, it generates a result for that transaction. The set of these results are contained in a result-list. When the LCB completes processing all of the items of a request-list, the result-list is transferred to an address specified within the request-list. The result-list is described in more detail in Section 4.4.

- **result-descriptor**: The user waits for request completion by waiting for activity on the RESULT_QUEUE. An entry in this queue will contain the address of the result-list which corresponds to the completed request-list. (See the result-list above.) In short, to use the request capability of the LCB, the user prepares (in the memory of their own CPU) a list of the requested transactions and communicates this list to the LCB by writing to the REQUEST_QUEUE. The user then “waits” for the LCB to process each transaction of the list. Meanwhile, the LCB reads the
REQUEST_QUEUE, transfers the list into an internal buffer and begins to process the list. As the LCB retires each item, it writes (into an internal buffer) the result of processing that item. At some point, the LCB completes processing the entire request-list and then DMA s a corresponding result-list into the user’s memory. This list has a result for each item of the request-list. The LCB communicates this list to the user by writing a result description to the RESULTS FIFO. The user then reads this FIFO to locate the result-list. This process is illustrated in Figure 30:

**Figure 30** Relationship between a transaction request and its result

### 4.1 Request-descriptors

The request-descriptor is used to specify a request-list to be queued for execution by the LCB. This specification includes the list’s address and length. The structure of the request-descriptor is illustrated in Figure 31. The user queues a request-list to the LCB by writing its corresponding descriptor to the LCB’s REQUEST_QUEUE.
length:  The size of the request-list (in 8-byte units). The location of the request-list is specified by the address field described below. Four bytes are implicit in the length, as any request-list must contain at least the address of the result-list. For example, a value of one (1) corresponds to a request-list size of 12 (4 + 8) bytes, or a value of seven (7) would correspond to 60 (4 + 56) bytes. As this field has a width of 9 bits, the largest request-list is 4092 (4+ 4088) bytes.

address:  The PCI address specifying the location of the request-list. In actual fact, only the upper-order 23 bits of the address, as all request-lists, must be aligned on a 512-byte boundary, and therefore, the low-order 9 bits of the address are an implied zero. The size of this list pointed to by this address is specified by the length field described above.

4.2 Request-lists

The specification of the transactions to be performed by the Transaction Engine is contained in a request-list. This list is provided by the user in PCI accessible memory. The request-list has two primary components:

- The address of a buffer into which the LCB will write the results of executing the list. The discussion of the information and structure of these results is contained in Section 4.4.
- A list specifying the packets to be sent by the LCB. An item of this list is described in Section 4.6.

The structure of a request-list is illustrated in Figure 32:
result-list address: A PCI address (in user memory target space) which specifies the location of a buffer to contain the result-list. In actual fact, only the upper-order 27 bits of the address, as any result, must be aligned on an thirty-two byte boundary, and therefore, the low-order five bits of the address are an implied zero. The structure and format of the result-list is discussed in Section 4.4.

request-items: A variable length list of request-items. The structure of any one item of this list is described in Section 4.3. The total size of this list is specified by the length field of the request-descriptor described in Figure 31.

4.3 Request-items

Each item on a request-list specifies one particular type of transaction requested to the LCB. While the detailed structure of an item varies as a function of the type of request, each request-item has the same generic structure illustrated in Figure 33:

opcode: A small integer which specifies the transaction to be performed by the Request Engine. All transactions can be partitioned into two sets: those transactions which result in the transmission of a LATP packet and those which do not. This division is reflected in the low-order bit of opcode. This field is called the inhibit transmit field. If this field is clear, the transaction includes a LATp packet to be transmitted;
if this field is set, the transaction does not involve the transmission of a packet. The set of possible transactions are enumerated in tables 12 and 13. Note that both the length field and the interpretation of the parameters field will vary as a function of the opcode.

**length:** The size of the parameter list. (See the parameters field.) The size is expressed in units of 32-bit words. A length of zero implies the transaction has no parameters. The maximum size of a parameter list 4092 bytes. This size is specified by a field value of 1023 (decimal).

**parameters:** a list of the parameters associated with the operation to be performed. The interpretation of this list depends on the value of the opcode field. The size of the parameter list is determined by the length field.

**stall/timeout:** Specifies a time interval, in units of system clock-ticks. The minimum time interval is zero. As the system clock runs at 20 MHZ, a value of one corresponds to an interval of 50 nanoseconds and the maximum value of 65535, corresponds to approximately 3.28 milliseconds. This field has two interpretations depending on whether or not the respond field of the payload header is set. If the respond field is clear, this field corresponds to the length of time to wait before executing any subsequent item. If this is the last item of a list, the interval specifies the minimum amount of time to wait before another list (if queued) can begin execution. If the respond field of the header is set, this field corresponds to the length of time to wait for a response to be returned from the destination node specified in the header template. If a response is not returned within this time, a result with the timeout field of the error word set is generated. The timer begins once the entire packet has been transmitted.

### 4.3.1 Requests which involve packet transmission

The structure of the opcodes used to specify those transactions which involve packet transmission is illustrated in Figure 34. Note that the low-order bit of the opcode is always clear.

![Figure 34](image)

**event:** If this field is set, the packet to be transmitted on the event fabric. If this field is clear, the packet is introduced on the Command/Response fabric.
respond: This field determines whether the packet’s sender expects a packet to be sent in response to receiving this packet. If the field is set, a response is expected. If the field is clear, no response is expected.

**Table 12** Opcodes for requests which involve packet transmission

<table>
<thead>
<tr>
<th>Opcode¹</th>
<th>Transaction</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>Transmit event</td>
<td>non-zero</td>
</tr>
<tr>
<td>0000</td>
<td>Transmit command</td>
<td>four (4)</td>
</tr>
<tr>
<td>0100</td>
<td>Transmit command and wait for response</td>
<td>four (4)</td>
</tr>
</tbody>
</table>

¹. Binary

### 4.3.1.1 Constructing the header for transmitted packets

Whenever the Transfer Engine must transmit a packet, it must also construct a legitimate LATp control header for that packet. The Transfer Engine derives the information necessary to construct the header as follows:

i. The **respond** field from the **opcode** field (see Figure 34) found in the corresponding **request-list** item.

ii. The **destination address** field from the template header (see Figure 35) found in the corresponding **request-list** item.

iii. The **protocol** field from the template header (see Figure 35) found in the corresponding **request-list** item.

iv. The 6-bit **source address** from the value found in the **board ID** field of the Front-End CSR register described in Figure 24.

v. Header parity using the value of the **header parity** field of the Front-End CSR register described in Figure 24.

The template header’s structure is illustrated in Figure 35:

![Figure 35 Structure of a template header](image)

**protocol:** This field and its values are used by the user to enumerate how the data of the transmitted packet is encoded. Its values are neither set or used by either the LCB or LATp.

**destination address:** The address of the node to receive the transmitted packet. Legitimate values for this field are discussed in [1].
4.3.1.2 Transmit event request

The structure of a request to send a packet to a node on the Event Fabric is illustrated in Figure 36. The destination address is determined by the header template field.

![Figure 36 Structure of “Transmit event” request](image)

- **length**: The size of the payload (including the payload header) of the packet to be transmitted. The payload size is expressed in units of 32-bit words; however, LATp packets may only be transmitted in units of cells. One cell is equal to 16 bytes or 4 words. Therefore, the lower-two bits of the length field must be zero. The minimum payload that may be transmitted is one cell. This length is specified by a field value of four. The maximum payload that may be transmitted is 255 cells (4080 bytes). This length is specified by a field value of 1020 (decimal).

- **header template**: A template, or mask for the first 16 bits of the payload. The LATp protocol assumes the first 16-bits of any payload follow the structure described in [1]. Many, if not most, of the fields of this structure are assigned by the LCB itself; however, some header fields are controlled by the user. This field specifies those fields of the header under the control of the user. A description of how the LCB constructs the header based on this template is found in Section 4.3.1.1.

- **payload**: The parameters for this transaction consist of the contents of the packet to be transmitted. The first 16-bits are reserved and should be MBZ. The second 16-bits of the payload are used to form the payload header. (See the header template field described above.) The remaining contents of this field are opaque to the LCB. The size of the payload is determined by the length field.

4.3.1.3 Transmit command request

The structure of a request to send a packet to a node on the Command/Response fabric is illustrated in Figure 37. The destination address is determined by the header template field.

Note that all command packets are one cell long. Consequently, the length field must have a value of four.
header template: A template, or mask, for the first 16 bits of the payload. The LATp protocol assumes the first 16-bits of any payload follow the structure described in [1]. Many, if not most, of the fields of this structure are assigned by the LCB itself; however, some header fields are controlled by the user. This field specifies those fields of the header under the control of the user. A description of how the LCB constructs the header based on this template is found in Section 4.3.1.1.

payload: The parameters for this transaction consist of the contents of the packet to be transmitted. The first 16-bits of the payload are assumed to be used to form the payload header. (See the header template field described above.) The contents of the remainder of this field are opaque to the LCB.

4.3.1.4 Transmit command, wait on response request

The structure of a request to send a packet to a node on the Command/Response fabric and wait for a response is illustrated in Figure 38. The destination address is determined by the header template field. Note that all command packets are one cell long. Consequently, the length field must have a value of four. The opcode links the sending of a packet with a response by setting the respond field. The LCB will then time-out the response, with a value determined by the stall/timeout field.
**header template:** A template, or mask, for the first 16 bits of the payload. The LATp protocol assumes the first 16-bits (in transmission order) of any payload follow the structure described in [1]. Many, if not most, of the fields of this structure are assigned by the LCB itself; however, some header fields are controlled by the user. This field specifies those fields of the header under the control of the user. A description of how the LCB constructs the header based on this template is found in Section 4.3.1.1.

**payload:** The parameters for this transaction consist of the contents of the packet to be transmitted. The first 16-bits of the payload are assumed to be used to form the payload header. (See the header template field described above.) The contents of the remainder of this field are opaque to the LCB.

### 4.3.2 Requests which do not involve packet transmission

The structure of the opcodes used to specify those transactions which do not involve packet transmission is illustrated in Figure 39. Note that the low-order bit of the **opcode** is always set.

**Figure 39** Structure of opcodes that do not involve packet transmission

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Transaction</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Mark time</td>
<td>zero</td>
</tr>
<tr>
<td>0011</td>
<td>Reset LAT</td>
<td>zero</td>
</tr>
<tr>
<td>0101</td>
<td>Access the Command and Status Register (CSR)</td>
<td>two (2)</td>
</tr>
<tr>
<td>0111</td>
<td>Access the FIFO faults register</td>
<td>two (2)</td>
</tr>
<tr>
<td>1001</td>
<td>Access the Event Statistics register</td>
<td>two (2)</td>
</tr>
</tbody>
</table>

1. Binary
4.3.2.1 Accessing Front-End registers

The majority of opcodes not involving packet transmission allow the user to access (read, potentially modify, and write) the set of registers contained within the Front-End interface. Requests involving register access require two parameters:

**value:** The value used to update the current content of the specified register. This parameter is treated as thirty-two one-bit fields. The value of any one field (zero or one) will replace the corresponding field in the register, if and only if the corresponding field in the *field select* parameter is set. (See below.)

**field select:** This parameter is used to determine which of the thirty-two one-bit fields of the register will be replaced with the corresponding field in the *value* parameter described above. If a field is set, the corresponding field of the register is updated with the value of the corresponding field of the *value* parameter. If the field is clear, the corresponding field of the register is left unchanged.

The *result* for an access request (see, for example, Section 4.3.2.4) returns two parameters: the value of the register before the access and the value of the register after the access. For example, to simply read a register, the *field select* parameter would be set to a value of zero. Or, to set the field of a register at bit offset three, and only that field, the *field select* parameter would have a contain eight (8) and so would the *value* parameter. However, to clear the same field, the *field select* parameter would remain the same, but the *value* parameter would now be zero.

4.3.2.2 Mark time request

The transaction specified by this *opcode* directs the Request Engine to perform a NOP. This transaction will not result in a packet being transmitted. The structure of this transaction is illustrated in Figure 40.

![Figure 40 Structure of “Mark time” request](image)

4.3.2.3 Reset LAT request

The transaction specified by this *opcode* directs the Request Engine to assert a LAT reset. This transaction will not result in a packet being transmitted. The structure of this transaction is illustrated in Figure 41.
4.3.2.4 Access CSR request

The transaction specified by this opcode is used to access (read and potentially modify) the Front-End Command and Status Register. This function and structure of this register is described in Section 2.2. This request requires two parameters: the potential new value of the register and a mask to selectively specify which fields within the register should be modified. (See Section 4.3.2.1.) This transaction will not result in a packet being transmitted. The structure of this transaction is illustrated in Figure 42.

4.3.2.5 Access FIFO faults register request

The transaction specified by this opcode is used to access (read and potentially modify) the Front-End FIFO faults register. This function and structure of this register is described in Section 2.3. This request requires two parameters: the potential new value of the register and a mask to selectively specify which fields within the register should be modified. (See Section 4.3.2.1.) This transaction will not result in a packet being transmitted. The structure of this transaction is illustrated in Figure 43.
4.3.2.6 Access Event Statistics register request

The transaction specified by this opcode is used to Access (read and potentially modify) the Event Statistics register. This function and structure of this register is described in Section 2.4. Like all other internal register requests, this request requires two parameters: the potential new value of the register and a mask to selectively specify which fields within the register should be modified. (See Section 4.3.2.1.). This transaction will not result in a packet being transmitted. The structure of this transaction is illustrated in Figure 44.

4.4 Result-descriptors

The result-descriptor is the structure returned every time the RESULT_QUEUE is read. It contains both a pointer to a result-buffer and detail concerning the status involved in transferring this buffer to user memory. This structure is illustrated in Figure 45. Descriptors are inserted by the LCB in the order that an activity completes. (See Section 1.5 for more details on result scheduling.)
status: An enumeration summarizing whether the transfers involved in the request-list completed successfully. Note that a result involves two transfers: the first transfers a request-list into the LCB from user memory, and the second transfers the corresponding result-list from the LCB into user memory. If this field is zero, the transfer completed successfully. If the field is non-zero, its value enumerates why the transfer failed. Potential reasons are discussed in Section 4.4.1.

direction: This field determines whether the address field (see below) contains either a request or result address. It also determines the possible values of the status field as enumerated in Table 14 and Table 15. If this field is clear, the address field points to a result-list and the status field may have only those values enumerated in Table 14. If the field is set, the address field points to a request-list and the status field may have only those values enumerated in Table 15.

address: The PCI address of either a request-list or result-list. In actual fact, it is the upper-order 28 bits of the address. As all DMA transfers are aligned on at least a sixteen-byte boundary, the low-order four bits of the address are an implied zero. What this address points to is only valid if the transfer status field is zero.

### 4.4.1 Transfer Engine Errors

This section enumerates any possible errors returned by the Transfer Engine when either transferring a request-list to the LCB or transferring a result-list from the LCB to user memory.

An error which occurs in transferring a request list to the LCB is called an in-bound error. If an in-bound error occurs, the direction field of the descriptor is set. These errors are summarized in Table 14. If an in-bound error occurs the address field of the descriptor contains the address of the request list, with one exception: if the error returned is REQUEST_QUEUE_EMPTY the address field will contain the value FFFFFFF (hexadecimal).
An error which occurs in transferring a result list from the LCB is called an out-bound error. If an out-bound error occurs, the direction field of the descriptor is clear. These errors are summarized in Table 15. If an out-bound error occurs the address field of the descriptor contains the address of the result list.

**Table 14** Meaning of result-descriptor status with the direction field indicates an out-bound error

<table>
<thead>
<tr>
<th>Name</th>
<th>Status Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI_MASTER_ABORT</td>
<td>1</td>
</tr>
<tr>
<td>PCI_PARITY_ERROR</td>
<td>2</td>
</tr>
<tr>
<td>PCI_TARGET_ABORT</td>
<td>3</td>
</tr>
<tr>
<td>RESULT_BUFFER_EMPTY</td>
<td>5</td>
</tr>
<tr>
<td>RESULT_QUEUE_EMPTY</td>
<td>7</td>
</tr>
</tbody>
</table>

The description of each one of these errors follows below:

**PCI_MASTER_ABORT:** is the error generated when no PCI target (i.e. slave) responds to the presentation of an destination address at the beginning of a transaction by asserting the DEVSEL signal within the required timeout period (usually 5 clock ticks or so). This is the classic non-existent memory error generated when the DMA address presented to the LCB does not correspond to memory accessible from the bus as currently configured. This error usually is a software error in the generation of a request or result list address, but may also indicate that the hardware windows between the PCI and the system main memory are not properly configured.

**PCI_SLAVE_ABORT:** is a specific response to an individual data cycle request which is generated by a PCI target (i.e. slave). The target generates the abort by initially recognizing the target address (by asserting the DEVSEL signal), but then subsequently de-asserting the DEVSEL signal while asserting the STOP signal. This type of error is typically generated when the combination of the requested cycle type, target address, and target internal state effectively request an operation that is not within the target’s current repertoire of operations. A simple example of such a situation might be that the address of a result list corresponds to memory which is read-only.
**PCI_PARITY_ERROR**: Is generated when the PCI bus PAR signal does not have the same parity as the 32 AD (address/data) bits, plus the 4 C/BE (command/byte-enables) bits that were present two ticks previously. This error could be typically generated when there was a PCI bus parity error while performing the DMA transfer of a request list from user memory to the LCB.

**RESULT_BUFFER_EMPTY**:

**RESULT_QUEUE_EMPTY**:

### 4.4.2 Null descriptors

### 4.5 Result-lists

The execution results of a request-list are contained in an result-list. The buffer for this list was provided by the user in PCI accessible memory and is specified in the corresponding request-list. (See Section 4.2.) The result-list has two primary components:

- The address of the request-list corresponding to the result-list. The discussion of the information and structure of a request-list is found in Section 4.2.

- A list specifying the execution results for each request-item processed by the LCB. An item of this list is described in Section 4.6.

The structure of an result-list is illustrated in Figure 46:

![Figure 46 Result-list](image)

**request-list address**: The PCI address specifying the location of the request-list corresponding to the result. In actual fact, the upper-order 23 bits of the address, as the request-list, is located on a 512-byte boundary, and therefore, the low-order nine bits of the address are an implied zero.

**padding**: A 32-bit word, unused by the LCB, in order to align the DMA operation associated with exporting this list from the LCB. The LCB ignores the value of this field.
result-items: A variable length list of result-items. The structure of any one item of this list is described in Section 4.6.

4.6 Result-items

Once processed, the LCB generates an individual result for each request-item of a request-list. The format, shape, and structure of an result-item depends on its corresponding request-item as illustrated by Figure 47. The size of the parameter block is variable and can be inferred by the opcode.

![Figure 47](image)

**Figure 47** One generic item of a LCB result-list

**timestamp:** The value of the timebase register, sampled when the transaction completed. The timebase is simply a 24-bit register which increments at the system clock rate (20 MHZ). The time at which this register is sampled depends on whether the transaction requested a response (as specified in the template header of Figure 36). If a response was not specified, the timebase is sampled immediately following the transmission of the packet. If a response was requested, the timebase is sampled when the response packet first arrives.

**parameters:** to be written.

**error:** This field is an enumeration of the possible errors found in either transmitting or receiving the packets associated with the transaction. If this field is zero, packet transmission or reception was successful. If the field is non-zero, its value corresponds to the reason packet processing failed. Potential reasons are described in Section 4.6.1.

4.6.1 Request Engine errors
INVALID_HEADER_PARITY: While receiving a LATp packet corresponding to a response, an error in the parity of the packet header was detected. The response may not be valid.

INVALID_DATA_PARITY: While receiving a LATp packet corresponding to a response, an error in the parity of the packet data was detected. The response may not be valid.

TRANSMIT_UNDERRUN: To be defined (FIFO read fault out of the request buffer during transmit?).

TIMEOUT: This error will be returned by only those items in which either the respond field is set, or an event was transmitted (see Section 4.6.2.1). In the case of the respond field being set, the transmitted LATp packet requested a response. A response was not received within the period specified by the request. The payload contents are not valid. In the case of a transmitted event, the transmission was held (pause was asserted) by the corresponding receiver for a time greater then the LCB’s transmit timeout. The transmit timeout is fixed and is equal to a value of xxx.

INVALID_ITEM: This error could be returned for a variety for reasons, including: The opcode number specified is either invalid or unsupported. The length specified is inconsistent with the opcode. The length field is inconsistent with respect to the total length of the list of which this item is a part.

4.6.2 Results which involve packet transmission

4.6.2.1 Transmit event result

This result is returned in response to the request described in Section 4.3.1.2. The result-item for a request to transmit an event returns no parameters as illustrated in Figure 48:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID_HEADER_PARITY</td>
<td>1</td>
</tr>
<tr>
<td>INVALID_DATA_PARITY</td>
<td>2</td>
</tr>
<tr>
<td>TRANSMIT_UNDERRUN</td>
<td>4</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>5</td>
</tr>
<tr>
<td>INVALID_ITEM</td>
<td>6</td>
</tr>
</tbody>
</table>
4.6.2.2 Transmit command result

This result is returned in response to the request described in Section 4.3.1.3. The result-item for this request returns no parameters as illustrated in Figure 49:

![Figure 48 Structure of “Transmit event” result](image)

4.6.2.3 Transmit command, wait on response result

This result is returned in response to the request described in Section 4.3.1.4. As a response was requested, the result also includes four (4) parameters that specify the payload of the response packet, which always has a fixed size of one cell (16 bytes). The structure of this result-item is illustrated in Figure 50:

![Figure 50 Structure of a “Transmit command, wait on response” result](image)

response header: If the response field of the summary is set (see, for example, Figure 47), this field contains the header of the response packet. The LATp protocol assumes the first 16-bits of any response payload follow the structure described in [1].
response: If the *response* field of the summary is set (see Figure 47), this field contains the contents of the packet received. The first 16-bits of the response are the packet header. (See the *response header* field above.) The contents of the remainder of this field are opaque to the LCB. All responses are a fixed 16 bytes in length (including the *response header*).

### 4.6.3 Results which do not involve packet transmission

#### 4.6.3.1 Mark time result

This result is returned in response to the request described in Section 4.3.2.2. The *result-item* for a LAT reset request returns *no* parameters as illustrated in Figure 51:

![Figure 51 Structure of a “Mark time” result](image)

**Figure 51** Structure of a “Mark time” result

#### 4.6.3.2 Reset LAT result

This result is returned in response to the request described in Section 4.3.2.3. The *result-item* for a LAT reset request returns *no* parameters as illustrated in Figure 52:

![Figure 52 Structure of a “LAT reset” request](image)

**Figure 52** Structure of a “LAT reset” request

#### 4.6.3.3 Access CSR result

This result is returned in response to the request described in Section 4.3.2.4. The result returns two parameters (as described in Section 4.3.2.1): the value of the register *before* the access and the value of the register *after* the access. The structure of this *result-item* is illustrated in Figure 53:
4.6.3.4 Access FIFO faults result

This result is returned in response to the request described in Section 4.3.2.5. The result returns two parameters (as described in Section 4.3.2.1): the value of the register before the access and the value of the register after the access. The structure of this result-item is illustrated in Figure 54:

![Figure 54](image)

4.6.3.5 Access Event Statistics result

This result is returned in response to the request described in Section 4.3.2.6. The result returns two parameters (as described in Section 4.3.2.1): the value of the register before the access and the value of the register after the access. The structure of this result-item is illustrated in Figure 55:

![Figure 55](image)
Chapter 5

Events

The LCB has the capability to both transmit and receive events. The subject of transmitting events is covered in the chapter on transaction processing (see Section 4.6.2.1), this chapter will discuss the interface used when receiving events.

Events are received from the fabric and DMA’d directly into user memory. From the viewpoint of the user events are seen to arrive both spontaneously and unsolicited. This requires the LCB to know a-priori where in user memory it should DMA any particular event. Because event arrival is unsolicited, user and LCB must cooperate in the management of the memory used to buffer incoming events. This issue is discussed from two different perspectives in Section 3.2.2 and Section 5.6 of this chapter.

Abstractly, the LCB is said to receives events from the LAT’s event fabric. In actual fact the LCB may be connected to the event fabric either by connecting directly to a module, or through the LAT’s Event Builder (EBM). Processing of event data is somewhat dependent on how its connected to the event fabric and this subject is discussed in Section 5.5.

5.1 Events and Event Packets

While the LAT’s event fabric produces events, the LCB delivers these events to its user as event packets. An event packet includes not only the event’s data, but also the LATp header associated with the event as well as a user defined prefix area. The structure of a event packet is discussed in Section 5.4. Typically, there is a one to one correspondence between events and event packets. However, given the finite buffering capacity of the LCB, events may span packets, that is, an event may contained in more then one packet. This topic is covered in detail in Section 5.5.
5.2 Overview of packet processing

Each time the LCB DMA’s an event packet into the circular buffer it also enqueues a 32-bit word onto the event queue (see Section 3.2.6). This word is used as both a reference to and a description of the DMA’d event packet. This word is called the event packet’s descriptor. The descriptor contains a pointer to the event packet, and as event packets are variable length, its size. The pointer is represented as an offset from the base of the circular buffer, allowing the pointer to be represented as either a PCI (memory space) or local CPU address. The structure of this descriptor is discussed in Section 5.3. The structure of the event packet referenced by its descriptor is discussed in Section 5.4. The relationship between event queue, event packet, and circular buffer is illustrated within Figure 56.

In order to process an event the user must read the event queue. If the queue is not empty, the read will return a legitimate descriptor. If the queue is empty, a sentinel value will be returned. The LCB delivers an interrupt (the event interrupt) which can be used by the application to determine when to examine the event queue. This interrupt can be configured to occur under a variety of conditions, including when:

- the circular buffer is full
- the event queue is not empty
- the event queue is 25% occupied
- the event queue is 50% occupied
- the event queue is 75% occupied

Configuring for the LCB for the desired interrupt condition is described more fully in Section 1.5.4 and Section 3.2.7.

Mention event enabling here.

One an event is processed, the user discards the event by updating the read pointer of the circular buffer (see Section 3.2.2 and Section 5.6). The new value of the read pointer is formed by summing the event descriptor’s offset and length.
Figure 56  Event packets and the event queue
The event-descriptor is the 32-bit structure returned every time the EVENT_QUEUE is read. Abstractly, the event descriptor provides a description of an arrived event packet. This description includes:

- where the packet is located in the circular buffer.
- the length of the packet.

**Figure 57** Event packets and the event queue

**Figure 58** Event processing flow

### 5.3 Event-descriptors
• whether or not an error occurred in transferring the packet from the fabric to the user.

This structure is illustrated in Figure 59. Descriptors are inserted by the LCB in the order that an events arrive (See Section 1.5).

<table>
<thead>
<tr>
<th>offset</th>
<th>length</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1727</td>
<td>32</td>
</tr>
</tbody>
</table>

**Figure 59** The event descriptor

**offset:** The offset (from the circular buffer base) of the buffer containing the event. The offset is expressed in units of 32-bit words.

**length:** The length of the buffer containing the received event packet. The length is expressed in units of 32-bit words. The length represents the size of the event packet payload corresponding to the received packet, plus the four (4) word user prefix (see Section 5.4).

**error:** This field which determines whether the event was captured by the LCB with or without errors. If this field is non-zero, the event was captured with errors. If the field is zero (0) the event was captured without errors. The interpretation of a non-zero value for this field is discussed in Section 5.3.1

### 5.3.1 Event errors

The **error** field of the event-descriptor (see Section 5.3) if non-zero, indicates that the transfer of the event packet either into the LCB from the fabric, or through the LCB to user memory generated an error. Consequently, the **error** field, as illustrated in Figure 60, is itself divided into two fields: one which specifies a receive error and the other which specifies a transfer error. The set of possible receive errors are enumerated in Table 17. The set of possible transfer errors are enumerated in Table 18. Note that it is possible for an event to have both a transfer and a receive error.

<table>
<thead>
<tr>
<th>transfer status</th>
<th>receive status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 3</td>
</tr>
</tbody>
</table>

**Figure 60** Event errors
5.3.1.1 Receive errors

**INVALID_HEADER_PARITY:** While receiving a packet, the Event Engine detected a parity error in the first 16 bits of the payload (the payload header). The payload contents may not be valid. The remainder of the cells following the control cell (if any) were discarded. The *data* field contents which are present may not be valid.

**INVALID_DATA_PARITY:** While receiving an event packet, the Event Engine detected an error in the parity for one of the cells of the packet. The remainder of the cells following the cell in error (if any) were discarded. The payload contents which *are* present may not be valid.

**PACKET_TRUNCATED:** While receiving an event packet, the Event Engine asserted *pause*. The *length* field reflects only that fraction of the packet received before *pause* was asserted. The remainder of the cells following the truncated cell were either discarded or will follow in a subsequent packet.

5.3.1.2 Transfer Errors

**PCI_MASTER_ABORT:** is the error generated when no PCI target (i.e. slave) responds to the presentation of an destination address at the beginning of a transaction by asserting the DEVSEL signal within the required timeout period (usually 5 clock ticks or so). This is the classic non-existent memory error generated when the DMA address presented to the LCB does not correspond to memory accessible
from the bus as currently configured. This error usually is a software error in the generation of a request or result list address, but may also indicate that the hardware windows between the PCI and the system main memory are not properly configured.

**PCI_SLAVE_ABORT:** Is a specific response to an individual data cycle request which is generated by a PCI target (i.e. slave). The target generates the abort by initially recognizing the target address (by asserting the DEVSEL signal), but then subsequently de-asserting the DEVSEL signal while asserting the STOP signal. This type of error is typically generated when the combination of the requested cycle type, target address, and target internal state effectively request an operation that is not within the target’s current repertoire of operations. A simple example of such a situation might be that the destination address of the DMA’d event corresponds to memory which is read-only.

**PCI_PARITY_ERROR:** Is generated when the PCI bus PAR signal does not have the same parity as the 32 AD (address/data) bits, plus the 4 C/BE (command/byte-enables) bits that were present two ticks previously. This error could be typically generated when there was a PCI bus parity error while performing the DMA transfer of an event from LCB to user memory.

**BUFFER_EMPTY:** The FIFO used as intermediate event buffer (EVENT_BUFFER) became prematurely empty during a DMA which attempted to read from that FIFO.

### 5.4 Event packet structure

The structure of the memory pointed to by a descriptor contains not only the received event packet, but a 4 (32-bit) word prefix. The contents of this prefix are opaque to the transfer of the event packet, that is the LCB neither reads (interprets) or writes the prefix. Therefore, the usage of these four words is entirely up to the application. For example, the application could use the words as forward and backward links in order to put the event packet on an application defined queue. Following the prefix is actual received LATp packet. The first 16-bits of the packet are reserved to the packet transmitter and LCB and contain a sequence number. The usage of this sequence number is described in Section 5.5. The second sixteen bits contain the packet’s LATp header. Note that packets can vary in size. As the value of the length field contained in the descriptor is inclusive of the prefix, the actual length of the packet can be derived by subtracting the length of the fixed size prefix (4) from the descriptor length. The structure of a typical event packet is illustrated in Figure 61.
5.5 Processing truncated events

Typically there is a one-to-one correspondence between event and event packet. However, as the amount of input buffering on the LCB is limited, events can potentially span event packets. An event which cannot be contained in a single packet is called a truncated event. Truncated events are associated with packet descriptors whose receive status is PACKET_TRUNCATED (see Table 17). Whenever the LCB cannot find the resources to consume an event, it asserts pause to the event source (or transmitter)\(^1\). The reaction of a transmitter to pause differs with the type of transmitter, consequently application processing of truncated events may also need to differ with transmitter type. There are two cases:

i. A module which discards on truncation. The TEM is an example of such a module.

ii. A module which re-transmits the remaining fraction of the event as additional packets. The Event Builder (EBM) is example of such a module.

In the first case, truncated events must be simply treated as errors as the entire event is not available to the application. In such a case, due to the expected very small event sizes sourced by this type of transmitter, the frequency of truncated events is expected to be small. The second case is somewhat more complicated as the application must “stitch” together a

---

1. This behaviour is dictated by the LATp flow control model (see xxx).
number of packets in order to re-constitute the event. There are three different cases which depend how many packets the event spans:

**The event fits in a single packet:** This, of course, is the expected and trivial case. The descriptor for the corresponding packet returns a transfer status of `SUCCESS` and the event follows directly after the prefix. The sequence field will contain a value of zero (0). This case is illustrated in Figure 62.

The event spans two packets: The descriptor of the first packet returns a transfer status of `PACKET_TRUNCATED` indicating that one more packet is required to complete the event. The sequence field will have a value of zero (0), indicating it is the first packet of the event. This packet will follow immediately (in time) after the first packet. The first piece of the event follows immediately after the prefix just as in the preceding case. The descriptor of the second packet returns a transfer status of `SUCCESS` indicating this is the last packet of the event. The sequence field will have a value of one (1), indicating it contains the second part of the event’s data. However, the remainder of the event does not follow directly after the prefix, as the header and sequence field for this packet are not part of transferred event. Therefore, the remaining part of the event follows immediately after the sequence and header fields. This case is illustrated in Figure 63.
The event spans two or more packets: The descriptor of the first packet returns a transfer status of `PACKET_TRUNCATED` indicating that one or more packets will follow in order to complete the event. The sequence field has a value of zero (0), indicating this is the first packet required to complete the event. The beginning of the event follows immediately after the prefix, just as in any other case. Subsequent packets of the event follow immediately in time after the first packet. Each one of these packets returns a transfer status of `PACKET_TRUNCATED`, a sequence field whose value is one greater than the previous packet, and whose event data follows after header and sequence fields. Eventually all the data of the event is transferred and the packet which contains the last piece of the event is identified by its successful transfer status. Just as in previous packets, the last event data follows after the header and sequence fields. This case is illustrated in Figure 64 as an event which spans three packets. Note the sequence field of the three packets, which increases monotonically from zero to two.

![Figure 63](image)

Figure 63 Processing an event which spans two packets
Figure 64 Processing an event which spans "n" packets
5.6 Deallocating packets

The user discards events by updating the read pointer of the circular buffer from which the packet was allocated. The new value of the read pointer is the offset of the packet, plus its length.

For example, assume the user obtains a packet by reading the event queue, checks for errors, computes a pointer to the event and processes the result:

Then once processing is complete, the user returns the result to the LCB by:

```c
eventPacket = *EVENT_QUEUE;
process(eventPacket);
*EVENTS_FREE = eventPacket.offset + eventPacket.length;
```