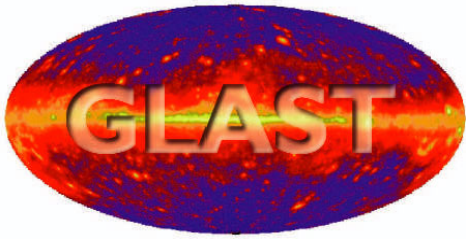

LAT Flight Software

LAT Communications Board Driver

Number: LAT-TD-01380
Subsystem: Data Acquisition/Flight Software
Supersedes: None
Type: Technical Document
Author: Curt Brune
Created: 17 March 2004
Updated: 21 April 2004
Printed: 22 September 2004

This document provides detailed descriptions of the software architecture and interfaces for the LCB Driver (LCBD). The hardware interface driver is described first, followed by a discussion of the external interfaces to the software driver.

Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision

Document Approval

Prepared By:			
	Curt Brune	LAT Flight Software	Date
Approved By:			
	G.Haller	LAT Electronics Manager	Date
Approved By:			
	J.J.Russell	LAT Flight Software Manager	Date

Record of Changes

Version	Date	Title Or Brief Description	Entered By
0.1	11 July 2002	Draft Release	Curt Brune
0.2	18 July 2002	Moved LAT System Initialization to Separate Document	Curt Brune
0.3	25 July 2002	Added Command and Response Chapter	Curt Brune
0.4	21 January 2003	Fleshed out software architecture	Curt Brune
0.5	17 March 2003	Updated ISR driver section and hardware section, incorporating feedback from the FSW February Workshop	Curt Brune
0.6	17 March 2004	Updated all sections to reflect the current state of the driver, now that the hardware has stabilized	Curt Brune

Contents

Figures

Tables

0 Introduction

This document describes the software architecture and interfaces of the LCB Driver (LCBD). The LCBD consists of a low-level hardware driver for the cPCI LAT Communications Board [huffer1] and a high-level external Application Programming Interface (API). The low-level hardware driver is used internally by LCBD to implement the external APIs. User software will only access the LCB through the external APIs.

0.0 LCB Primer

The LAT Communications Board (LCB), a cPCI module residing in every cPCI crate within the LAT, provides communication within the LAT. A LCB can communicate directly with other LCBs (in other crates) on the event fabric or with other LATp nodes on the LAT's command/response fabric [huffer2].

The LCB manages traffic on the Command/Response fabric and the Event Data fabric. With the Command/Response fabric a node can issue a command to another LATp node and receive the solicited response. For example, this is used to read a register on a remote electronics module. The key point is the response is solicited.

Event data, however, arrives unsolicited, i.e. whenever it is ready. A special case of "event data" is the unsolicited LCB-to-LCB communication. In both cases a handler is notified and processes the unsolicited data in a timely manner.

The LCB hardware design calls for two FPGAs that communicate via intermediate FIFOs. The PCI FPGA controls the PCI bus and provides the interface to PCI registers. The LATp FPGA controls LATp specific I/O features, e.g. sending commands, receiving response and receiving event data. The LATp FPGA also contains a small number of 32-bit registers.

0.1 LCBD Overview

The LCBD provides high-level external APIs to the user for managing the LCB:

- Initialization, Configuration and Statistics
- Command/Response Data
- Unsolicited Event Data
- Unsolicited LCB-to-LCB Data
- Polled mode operations during EPU boot

These high-level APIs will remain constant even as the underlying hardware matures from the prototype to the final production version.

The LCBD also includes a low-level hardware interface to the LCB. This interface, however, is not exposed directly to the end user. It is used internally by LCBD to implement the external APIs.

The hardware interface provides basic initialization and configuration services, including simple access to the LCB's PCI register model.

1 Hardware Interface Driver

The hardware interface driver supports low-level services of the LCB, including:

- Access to the LCB PCI registers (PCI FPGA).
- Access to the LCB LATp registers (LATp FPGA).
- Initialization and configuration routines.
- LCB Statistics.

The hardware interface is intended to be a private interface used by LCBD to support the high-level public interface. However, the hardware engineer designing and implementing the LCB may also find the hardware interface useful for debugging purposes.

A diagram of the cPCI crate and PCI bus is shown in Figure 1-1. This diagram shows the relationship between the single board computer (SBC), the hardware interface, the PCI bus and the LCB.

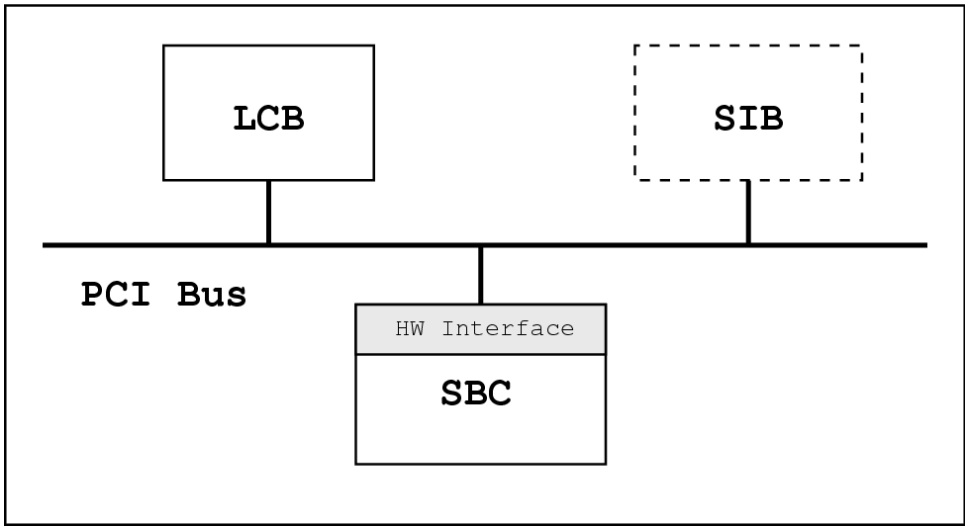

Figure 1-1 PCI Bus and cPCI Modules

Hard copies of this document are for REFERENCE ONLY and should not be considered the latest revision

1.0 Register Model

The LCB has three types of registers: PCI Configuration registers and PCI Memory registers provided by the PCI FPGA, and LAT registers provided by the LATp FPGA. See [huffer1] for a complete list of these registers. The LCBD hardware interface provides read/write access to these registers.

1.0.0 PCI Configuration Space Registers

The PCI configuration space registers are used to identify and locate the LCB on the PCI bus. Once the LCB is located (by finding locating the VendorID/DeviceID) the configuration space registers are programmed to configure the base address of the PCI Memory Space.

The PCI configuration space registers are manipulated on long word(32 bit), word(16 bit) or byte(8 bit) boundaries at byte offsets from the beginning on the configuration space. Word in/out routines are shown below:

```
#include “LCB/LCB_pci.h”

LCB_pci_cfg_inWord ( const LCB  lcb, int offset, unsigned short *val);
LCB_pci_cfg_outWord( const LCB  lcb, int offset, unsigned short  val);
LCB_pci_cfg_inLong ( const LCB  lcb, int offset, unsigned int   *val);
LCB_pci_cfg_outLong( const LCB  lcb, int offset, unsigned int    val);
```

1.0.1 PCI Memory Space Registers

The PCI Memory space registers are all 32-bit registers. These registers control various aspects of the LCB, including LATp parameters, test features, sending commands, receiving responses and event data taking.

All registers have 32-bit read/write interface functions.

1.0.1.0 Control and Status Register (CSR)

The 32-bit Read/Write functions for the CSR are shown below. LCB_REG_CSR, a bit field defined in LCB/LCB_reg.h, defines the format of the CSR.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_reg.h”

LCB_io_CSR_read ( const LCB  lcb, unsigned int *val);
LCB_io_CSR_write( const LCB  lcb, unsigned int  val);
```

1.0.1.1 EVENTS_BASE and EVENTS_FREE Registers

The EVENTS_BASE and EVENTS_FREE registers manage the circular buffer used for event data. The EVENTS_BASE register defines the origin of the circular buffer, while the EVENTS_FREE register maintains the "read pointer" for the circular buffer. The "write pointer" is maintained internally by the LCB.

LCB_REG_Event_Base and LCB_REG_Event_Free bit fields defined in LCB/LCB_reg.h, define the format of these registers.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_reg.h”

LCB_io_EVENTS_BASE_read ( const LCB  lcb, unsigned int *val);
LCB_io_EVENTS_BASE_write(       LCB  lcb, unsigned int  val);
LCB_io_EVENTS_FREE_read ( const LCB  lcb, unsigned int *val);
LCB_io_EVENTS_FREE_write( const LCB  lcb, unsigned int  val);
```

1.0.1.2 REQUEST_QUEUE Register

The REQUEST_QUEUE is a write-only register that is written with a 32-bit request descriptor to initiate the transmission of a command list.

LCB_reqDesc, a bit field defined in LCB/LCB_latp.h, defines the format of a request descriptor.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_latp.h”

LCB_io_REQUESTQ_write(const LCB  lcb, unsigned int val);
```

1.0.1.3 RESULT_QUEUE Register

The RESULT_QUEUE is a read-only register that is read to retrieve a 32-bit result descriptor, which describes a result list.

LCB_rstDesc, a bit field defined in LCB/LCB_latp.h, defines the format of a result descriptor.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_latp.h”

LCB_io_RESULTQ_read( const LCB  lcb, unsigned int *val);
```

1.0.1.4 EVENT_QUEUE Register

The EVENT_QUEUE is a read-only register that is read to retrieve a 32-bit event descriptor, which describes a single event.

LCB_evtDesc, a bit field defined in LCB/LCB_latp.h, defines the format of an event descriptor.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_latp.h”

LCB_io_EVENTQ_read( const LCB  lcb, unsigned int *val);
```

1.0.1.5 INTERRUPT_ENABLE Register

The `INTERRUPT_ENABLE` register controls the LCB’s ability to raise interrupts. `LCB_REG_irqEnb`, a bit field defined in `LCB/LCB_reg.h`, defines the format of this register.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_reg.h”

LCB_io_IRQ_read ( const LCB  lcb, unsigned int *val);
LCB_io_IRQ_write( const LCB  lcb, unsigned int val);
```

1.0.1.6 DEBUG Register

The `DEBUG` register is a read only register that exposes some of the LCB’s internal state for debugging purposes, including

- Current Event Circular Buffer High Water Marks
- Current Event Queue High Water Marks
- Current Event Write Pointer

`LCB_REG_Debug`, a bit field defined in `LCB/LCB_reg.h`, defines the format of this register.

```
#include “LCB/LCB_io.h”
#include “LCB/LCB_reg.h”

LCB_io_DEBUG_read( const LCB  lcb, unsigned int *val);
```

1.0.2 LAT Registers

The LAT Registers reside in the LATp FPGA. These registers are accessed using LIOX command items, which require all the machinery necessary for creating and sending LIOX commands. See section 2.2.2 for more details on the machinery.

The LATp registers take two 32-bit input parameters, a value and a mask. The set bits in the mask specify which bits of the destination register are writable, while the value specifies what to the set the writable bits to. This allows the user to set specific bits of a register atomically, without first reading the register, modifying the value and writing it back. A mask of `0x0` can be used to read a register.

The LATp register access also returns two values – the current value of the register after the write and the previous value of the register before the write.

Since accessing these registers requires a lot of machinery two interfaces are provided – both interfaces are synchronous, with one using all the LIOX machinery including interrupts, while the

other interface uses polling instead of interrupts. There are times during initialization when you need to write a LATp register, but interrupts are not yet enabled.

1.0.2.0 Control and Status Register (CSR)

The LATp CSR register configures and monitor several aspects of the LAT communications, including the LCB LATp address, byte-wide event data, commander/slave, event path select and parity testing functions.

```
#include “LCB/LIOX_sync.h”

LIOX_sync_CSR( LIOXs             slh,
               unsigned int  new_val,
               unsigned int     mask,
               unsigned int *cur_val,
               unsigned int *old_val);

LIOX_psync_CSR( LIOXs             slh,
                unsigned int  new_val,
                unsigned int     mask,
                unsigned int *cur_val,
                unsigned int *old_val);
```

1.0.2.1 FIFO Faults Register

The FIFO faults register latches any errors encountered by the LCB when accessing its internal FIFOs. When this register is written the value written is ignored and any latch status is cleared.

```
#include “LCB/LCB_sync.h”

LIOX_sync_FIFO_FAULTS(LIOXs             slh,
                      unsigned int  new_val,
                      unsigned int     mask,
                      unsigned int *cur_val,
                      unsigned int *old_val);
```

1.1 Initialization and Configuration

Before using the LCB it must first be configured for a particular task. Before configuring, however, the board must first be initialized.

Initialization refers to the set of required operations necessary to put the LCB into a functioning, well known state. These operations include:

- Handle Allocation
- Board Discovery

- Default Configuration

1.1.0 Handle Allocation

The LCBD hardware interface is managed using an opaque handle, whose type is pointer to `struct _LCB`. All of the hardware interface functions use the members of `struct _LCB` to access the underlying hardware.

Before initializing the LCB, memory must first be allocated for the opaque handle. The `LCB_sizeOf()` function returns the size of `struct _LCB`, allowing the higher-level interface to manage memory allocation.

```
unsigned int LCB_sizeOf(void);
```

A simple-minded example using `malloc()` is shown below:

```
typedef struct _LCB* LCB;

LCB lcb;
lcb = (LCB)malloc(LCB_sizeOf());
```

1.1.1 LCB Initialization

LCB initialization refers to the process of detecting the LCB within the cPCI crate and configuration. Once detected various parameters about the LCB are stored in the opaque handle previously allocated. Examples of parameters include the memory-mapped locations of the PCI Memory Space and interrupt parameters. These parameters are used by subsequent calls to the hardware interface in order to access registers.

The `LCB_create()` function attempts to detect a LCB. If successful this routine stores the associated parameters in the LCB handle.

```
unsigned int LCB_create(LCB lcb);
```

Now the LCB handle is ready to use.

1.1.1.0 Default Configuration

The hardware interface is also responsible for leaving the LCB in a default, well-known state after initialization. The default state requires programming of PCI Memory Space registers and the LAT CSR register.

1.1.1.1 PCI Memory Space Registers

Setting the default configuration requires setting the PCI CSR (Section 1.0.1.0), INTERRUPT_ENABLE (Section 1.0.1.5) and EVENT_BASE (Section 1.0.1.1) registers.

For the CSR the settings are:

- Command Path A Selected
- Inhibit testing bits clear
- Interrupt conditions (event queue) non-empty
- Interrupt conditions (event buffer) ¾ full

Interrupts are disabled using the INTERRUPT_ENABLE register.

The location of the event circular buffer is written into the EVENT_BASE register.

1.1.1.2 LATp CSR Register

During initialization the LATp CSR is programmed using the polled mode (Section 1.0.2.0). The important settings are:

- Commander/Responder setting
- Events disabled
- Event path A selected
- Use odd LATp header parity
- Use odd LATp payload parity
- LATp physical address
- LATp Byte wide enable

1.1.1.3 Installing FORK queues

As described further in Section 2 the queuing of commands and the reception of results and events are decoupled from the hardware using FORK message queues. During system initialization these queues are installed by the system startup.

1.1.1.3.1 Command Request Queue

The LCB can only handle two outstanding requests at a time. The command request queue buffers user command requests and sends them out at a rate the LCB can handle. The command request queue is installed using:

```
LIOX_sysInit( LCB               lcb,
              FORK_fcb      *req_fcb,
              TASK_attr    *req_attr,
              FORK_msg_sys     *pMsg,
              unsigned int      nMsg);
```

The FORK_fcb, TASK_attr and FORK_msg_sys arguments allows the system to specify several parameters of the command queue:

- VxWorks task priority

- VxWorks task name
- Depth of the message queue (how many commands can back up)

The request mechanism only uses one queue from the FORK object.

1.1.1.3.2 ISR Dispatch Queue

See section 4.1 for a proposal to change this interface.

The VxWorks interrupt handler simply disables interrupts and queues a message to the ISR Dispatch Queue.

The ISR dispatch FORK queue is installed using:

```
LCB_isr_fork_set( LCB  lcb, FORK_fcb *fcb)
```

This function uses queue 0 from the `fcb` parameter for the ISR dispatch queue. The LCB driver reserves the queue returned by `FORK_que_get(fcb, 0).` The FCB could contain additional queues for synchronizing access to the event handler tasks.

Note: The FCB is expected to use system messages.

The user has complete responsibility for allocating memory for the FORK queue and the system messages. An example of a single FORK queue with 10 system messages in shown below:

```
TASK_attr          t_attr;

t_attr.options    = 0;
t_attr.stack_addr = 0;
t_attr.stack_size = 0;
t_attr.name       = "tLCBbh";
t_attr.priority   = 10;

pFCB    = (FORK_fcb *)MBA_alloc(FORK_fcb_sizeof(1));
pSysMsg = (FORK_msg_sys *) MBA_alloc( sizeof(FORK_msg_sys) * 10);

// create FORK queue
status = FORK_create(pFCB,           // fork control block
                     &t_attr,        // use task attributes
                     NULL,           // default callback
                     lcb,            // callback parm
                     NULL,           // timeout callback parm
                     TOC_FOREVER,    // timeout
                     1,              // define one interal queue
                     NULL,           // no queue cfg for single queue
                     FORK_K_TYPE_PENDING,
                     // blocking type when using system msg,
                     pSysMsg,        // source of system msg.
                     10);            // count of system msg.
```

1.1.2 Enabling Interrupts

After the FORK message queues are installed the LCB is ready to have interrupts enabled. The following routine enables interrupts:

```
LCB_isr_start ( LCB  lcb);
```

This function prepares the LCB for receiving solicited and unsolicited response data. After this function is called the LCB is ***live*** and the LCBD ISR is enabled.

1.2 LCB Statistics

The LCBD maintains several different counters for various events that occur at the driver level. For each type of counter the LCBD provides an interface for retrieving the counter and for clearing the counter.

1.2.0 Interrupt Statistics

The LCBD maintains a count of the number of interrupts received. The interface to this counter is shown below:

```
int LCB_stat_irq_cnt_get(const LCB *lcb, unsigned int *count);

int LCB_stat_irq_cnt_clr(LCB *lcb);
```

1.2.0.0 Error Statistics

The LCBD maintains PCI and LATp error counters for both cmd/rsp and event data.

1.2.0.0.1 LCB PCI Import Errors

The following errors occur when the LCB imports data from the single board computer across the PCI bus.

```
enum _LCB_PCI_IN_ERR {
  LCB_PCI_IN_ERR_MASTER_ABORT = 1,
  LCB_PCI_IN_ERR_PCI_PARITY   = 2,
  LCB_PCI_IN_ERR_TARGET_ABORT = 3,
  LCB_PCI_IN_ERR_BUFFER_FULL  = 4,
  LCB_PCI_IN_ERR_Q_EMPTY      = 7,
  LCB_PCI_IN_ERR_LAST         = 8,
};
typedef enum _LCB_PCI_IN_ERR LCB_PCI_IN_ERR;
```

1.2.0.0.2 LCB PCI Export Errors

These errors occur when the LCB exports data to the single board computer across the PCI bus.

```
enum _LCB_PCI_OUT_ERR {
  LCB_PCI_OUT_ERR_MASTER_ABORT = 1,
  LCB_PCI_OUT_ERR_PCI_PARITY   = 2,
  LCB_PCI_OUT_ERR_TARGET_ABORT = 3,
  LCB_PCI_OUT_ERR_STOP         = 4,
  LCB_PCI_OUT_ERR_BUFFER_EMPTY = 5,
  LCB_PCI_OUT_ERR_INSFMEM      = 6,
  LCB_PCI_OUT_ERR_Q_EMPTY      = 7,
  LCB_PCI_OUT_ERR_LAST         = 8,
 };
typedef enum _LCB_PCI_OUT_ERR LCB_PCI_OUT_ERR;
```

1.2.0.0.3 LCB LATp Receive Errors

These are errors that the LCB detects when accepting data from a LATp fabric. That is these errors occur when the LCB reads response data or event data.

```
enum _LCB_RECV_ERR {
  LCB_RECV_ERR_HDR_PARITY        = 1,
  LCB_RECV_ERR_DATA_PARITY       = 2,
  LCB_RECV_ERR_TRUNCATE          = 3,
  LCB_RECV_ERR_TRANSMIT_UNDERRUN = 4,
  LCB_RECV_ERR_TIMEOUT           = 5,
 };
typedef enum _LCB_RECV_ERR LCB_RECV_ERR;
```

1.2.0.0.4 Command Result PCI Errors

The interface for retrieving and clearing these errors is

```
LCB_stat_rst_err_get( const LCB       lcb,
                      int       direction,
                      LCB_XFER_ERR errNum,
                      unsigned int   *cnt);

LCB_stat_rst_err_clr( LCB             lcb,
                      int       direction,
                      LCB_XFER_ERR errNum);
```

The direction parameter selects the transfer direction of the error. 0 is for LCB to SBC DMA errors (result list DMA) and 1 is for SBC to LCB DMA errors (command list DMA). The `errNum` parameter is a `union` of `LCB_PCI_IN_ERR` and `LCB_PCI_OUT_ERR` depending on the `direction`.

1.2.0.0.5 Event Reception Errors

The interface for retrieving and clearing event PCI errors is

```
LCB_stat_evt_xfer_err_get( const LCB          lcb,
                           LCB_PCI_OUT_ERR errNum,
                           unsigned int      *cnt);

LCB_stat_evt_xfer_err_clr( LCB                lcb,
                           LCB_PCI_OUT_ERR errNum);
```

Event PCI errors only occur when the LCB is sending data to the SBC.

The interface for retrieving and clearing LATp event errors is

```
LCB_stat_evt_recv_err_get( const LCB       lcb,
                           LCB_RECV_ERR errNum,
                           unsigned int   *cnt);

LCB_stat_evt_recv_err_clr( LCB             lcb,
                           LCB_RECV_ERR errNum);
```

1.2.0.1 LATp I/O Statistics

LATp defines 16-bit transmitter statistics and receiver statistics for all nodes on a fabric [huffer2]. The LCBD maintains these statistics for the LCB's communications on the command/response fabric and the event fabric. Functions for retrieving and clearing these statistics are declared below.

```
LCB_stat_latp_get ( const LCB      lcb,
                    LATp_Fabric fabric,
                    LATp_RX        *rx,
                    LATp_TX        *tx);

LCB_stat_latp_clr ( LCB            lcb,
                    LATp_Fabric fabric);
```

The fabric parameter is an enumerated type shown below

```
typedef enum _LATp_Fabric {
  LATP_FABRIC_CMD    = 0,
  LATP_FABRIC_EVENT  = 1,
} LATp_Fabric;
```

2 Interrupt Mode Driver

The Figure 2-1below shows the software architecture for the interrupt mode LCB driver.

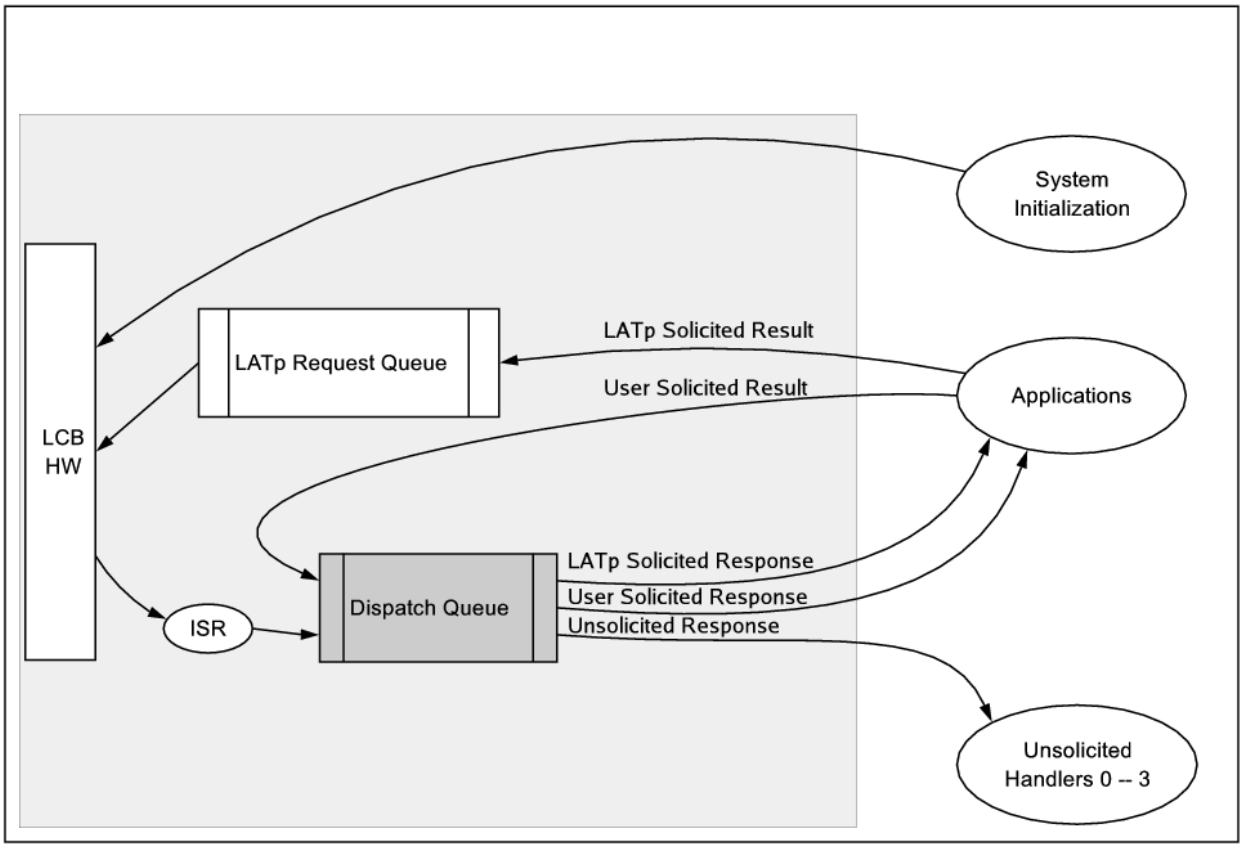

Figure 2-1 Interrupt Mode Driver Architecture

We will refer back to this diagram throughout this chapter.

2.0 Driver Libraries

The LCBD interface presents a public, stable interface to multiple users. All user applications communicate via the LCBD interface. The LCBD interface, in turn depends on the low level hardware interface (Chapter 1) and on several utility services provided by PBS, a common LAT Flight Software utility library.

The Figure 2-2 below shows the interrupt mode LCB driver library and its dependencies. Note that the user applications only interact with the public interface of the LCBD.

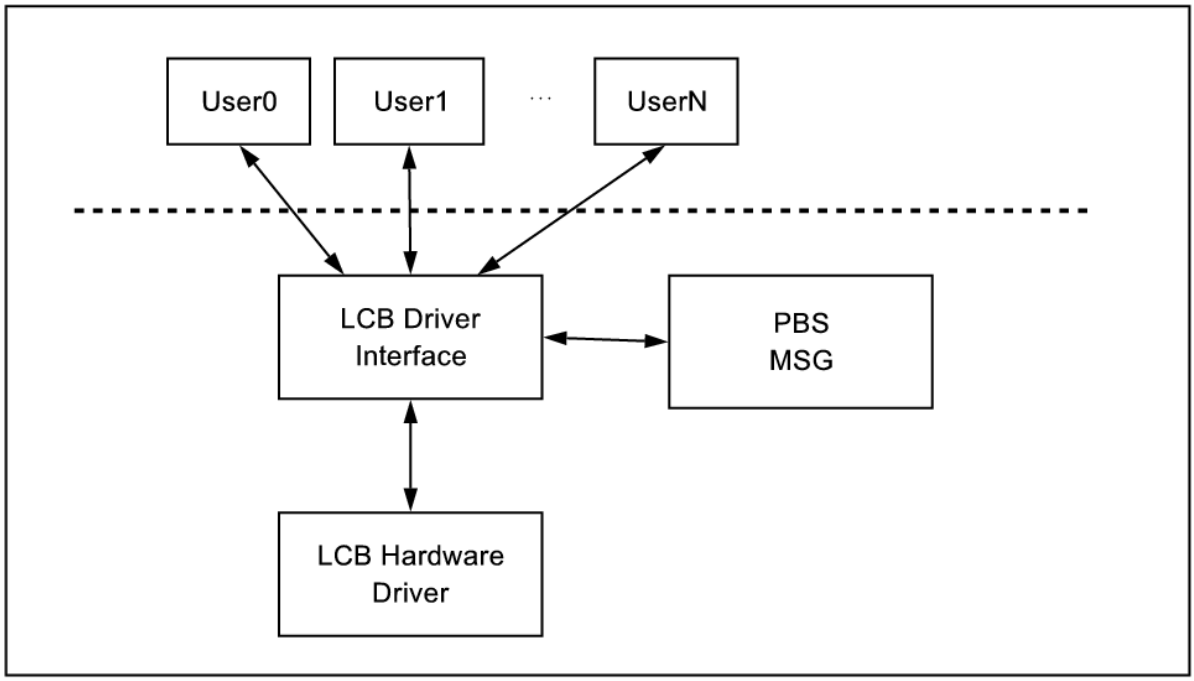

Figure 2-2 Interrupt Mode Driver Libraries

The LCBD uses the following services from PBS:

- **FORK Queues –** The LCBD relies on message queues to buffer communications between the hardware and the user applications. This includes messages sent from the user application (commands) and messages sent from the LCB (responses or unsolicited data).
- **RW –** The LCBD needs interlocked access to a few specific resources.
- **Memory Allocators –** The LCBD provides an interface allowing users to allocate command lists and result lists with the required memory alignment.
- **SPIN –** The polled mode driver uses the busy wait spin routines to pass time.

2.1 Overview of Operation

The LCBD provides the following services:

- Sending packets on the LATp command fabric.
- Receiving and dispatching solicited packets from the LATp response fabric, i.e. receive the responses from commands.
- Injecting timed markers into the results FIFO.

- Resetting the LATp command fabric.
- Reading/Writing LCB LATp registers.
- Sending unsolicited packets on the LATp event fabric, the so-called LCB-to-LCB data.
- Receiving and dispatching unsolicited packets from the LATp event fabric, including event data and LCB-to-LCB data.
- Sending and dispatching user solicited commands for synchronization purposes.

2.2 Command/Response Interface

This section describes the sending of commands and the processing of results with the LCBD. Commands are bit sequences sent down to the front end electronics (register reads, register writes and dataless commands). Every command generates a result - register writes and dataless commands generate a simple "command successfully transmitted" result, while register read commands return the register value as a result.

Some definitions used within this section:

Command Item or simply ***Command***

A single command sent to the front end electronics.

Command List

An ordered list of command items containing one or more command items.

Result Item or simply ***Result***

A single result from the LCB in reply to a command item.

Result List

An ordered list of result items from the LCB in reply to a command list.

LAT I/O Transaction (LIOX)

An indivisible unit of work consisting of a command list and its associated result list.

The LCB has the capability of bundling several command items together and sending them sequentially as a command list. After all the commands are processed and all the responses are received the LCB fires an interrupt that the driver traps. The driver then notifies the user.

This arrangement allows for asynchronous commanding, e.g. a user can queue a command list to the LCB, continue to do other work and then be notified asynchronously that the results are ready for processing.

For the case when a user only wants to send a single command the LCBD also provides a synchronous interface. With this interface a user queues a single command to the LCB and waits for the command to complete.

2.2.0 LIOX Logical Node Address

All of the LIOX commanding functions that access LATp nodes require a *logical* node address parameter of the enumerated type `LIOX_addr`. This parameter is a logical constant representing a LATp node within the LAT, allowing the user to reference a particular node by name. The logical node address isolates the user from needing to know the *physical* LATp address of a node.

The logical LIOX addresses are here

```
enum __LIOX_addr {
  LIOX_addr_TEM_0     = 0x0,
  LIOX_addr_TEM_1     = 0x1,
  LIOX_addr_TEM_2     = 0x2,
  LIOX_addr_TEM_3     = 0x3,
  LIOX_addr_TEM_4     = 0x4,
  LIOX_addr_TEM_5     = 0x5,
  LIOX_addr_TEM_6     = 0x6,
  LIOX_addr_TEM_7     = 0x7,
  LIOX_addr_TEM_8     = 0x8,
  LIOX_addr_TEM_9     = 0x9,
  LIOX_addr_TEM_10    = 0xA,
  LIOX_addr_TEM_11    = 0xB,
  LIOX_addr_TEM_12    = 0xC,
  LIOX_addr_TEM_13    = 0xD,
  LIOX_addr_TEM_14    = 0xE,
  LIOX_addr_TEM_15    = 0xF,
  LIOX_addr_GEM       = 0x10,
  LIOX_addr_AEM       = 0x11,
  LIOX_addr_EBM       = 0x12,
  LIOX_addr_PDU_0     = 0x13,
  LIOX_addr_PDU_1     = 0x14,
  LIOX_addr_CRU       = 0x1E,
  LIOX_addr_SLV_BCAST = 0x1F,
  LIOX_addr_SIU_EXT   = 0x21,
  LIOX_addr_SIU_0     = 0x22,
  LIOX_addr_SIU_1     = 0x23,
  LIOX_addr_EPU_0     = 0x24,
  LIOX_addr_EPU_1     = 0x25,
  LIOX_addr_EPU_2     = 0x26,
  LIOX_addr_MST_BCAST = 0x1F,
  LIOX_addr_NULL
};

typedef enum __LIOX_addr LIOX_addr;
```

Before `LCBD` sends a command to the LCB it translates the logical address to the corresponding physical address using a look-up table. The look-up table is populated once during system initialization. The following functions are used to populate and query the look-up table.

```
LIOX_addr_logic_set  ( LIOX_addr      logic_addr,
                       unsigned short  phys_addr);

LIOX_addr_logic_get  ( unsigned short  phys_addr,
                       LIOX_addr     *logic_addr);

LIOX_addr_phys_get   ( LIOX_addr      logic_addr,
                       unsigned short *phys_addr);
```

2.2.1 Asynchronous Command and Response Interface

The asynchronous interface allows users to queue command lists to the LCB for processing. While the LCB is busy processing the command list and receiving result items the user can continue to do other work. The LCBD notifies the user after the final command item and result item are processed. After notification the user proceeds to inspect the result list.

The LCBD provides a data structure for organizing the command list and the result list called the LAT I/O Transaction (LIOX). Users communicate with the LCBD via an opaque handle to a LIOX structure. A diagram showing the relationship between the command list, result list and LIOX structure is shown below in Figure 2-3.

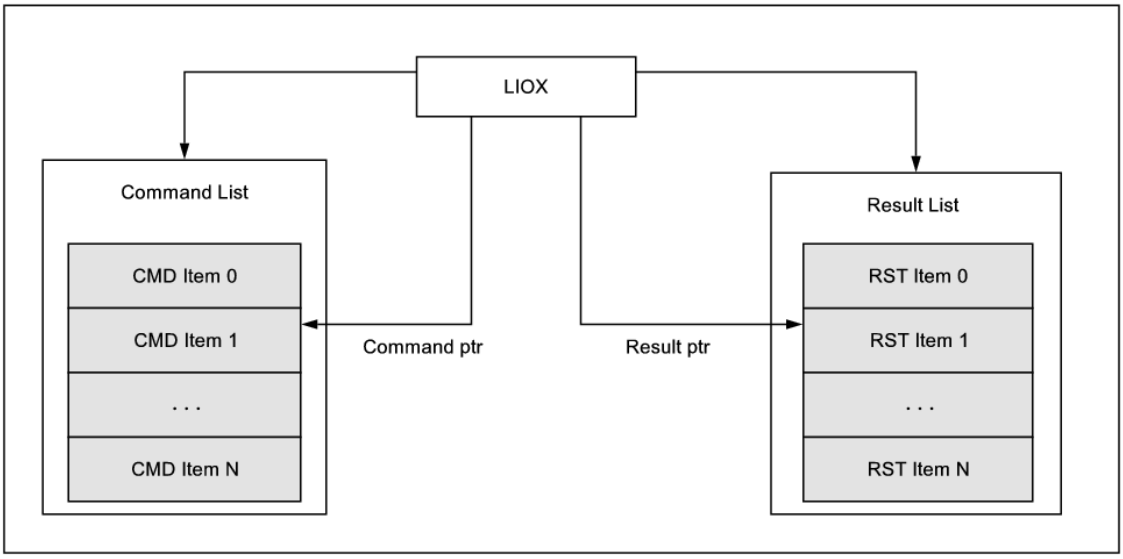

Figure 2-3 Overview of LIOX data structure

In addition to organizing memory the LIOX also maintains an internal command pointer used when filling the command list. The command pointer behaves much like a file pointer - it always points to the memory location where the next command will be inserted. As commands are added to the command list the command pointer is updated to point to the next available command slot.

The LIOX also performs bounds checking, i.e. it will not allow more commands to be added than will fit into the available memory for the command list.

2.2.1.0 Preparing for Commanding

Before creating and sending command lists via the LCBD a LIOX handle must first be properly initialized. This entails the following activities:

- Allocate memory for the LIOX structure, the command list and the result list.
- Implement a user-defined result processing function called during result dispatch.
- Initialize the LIOX by passing in pointers for the command list memory, for the result list memory and for the result processing function.

Once created a LIOX handle can be in one of the following states:

LIOX State	Description
LIOX_READY	Allocated, ready for transmission
LIOX_PEND	Command list sent, waiting for result
LIOX_RECEIVED	Result received

Table 2-1 LIOX State

The allowable state transitions are shown below in Figure 2-4. These states are discussed in the following sections.

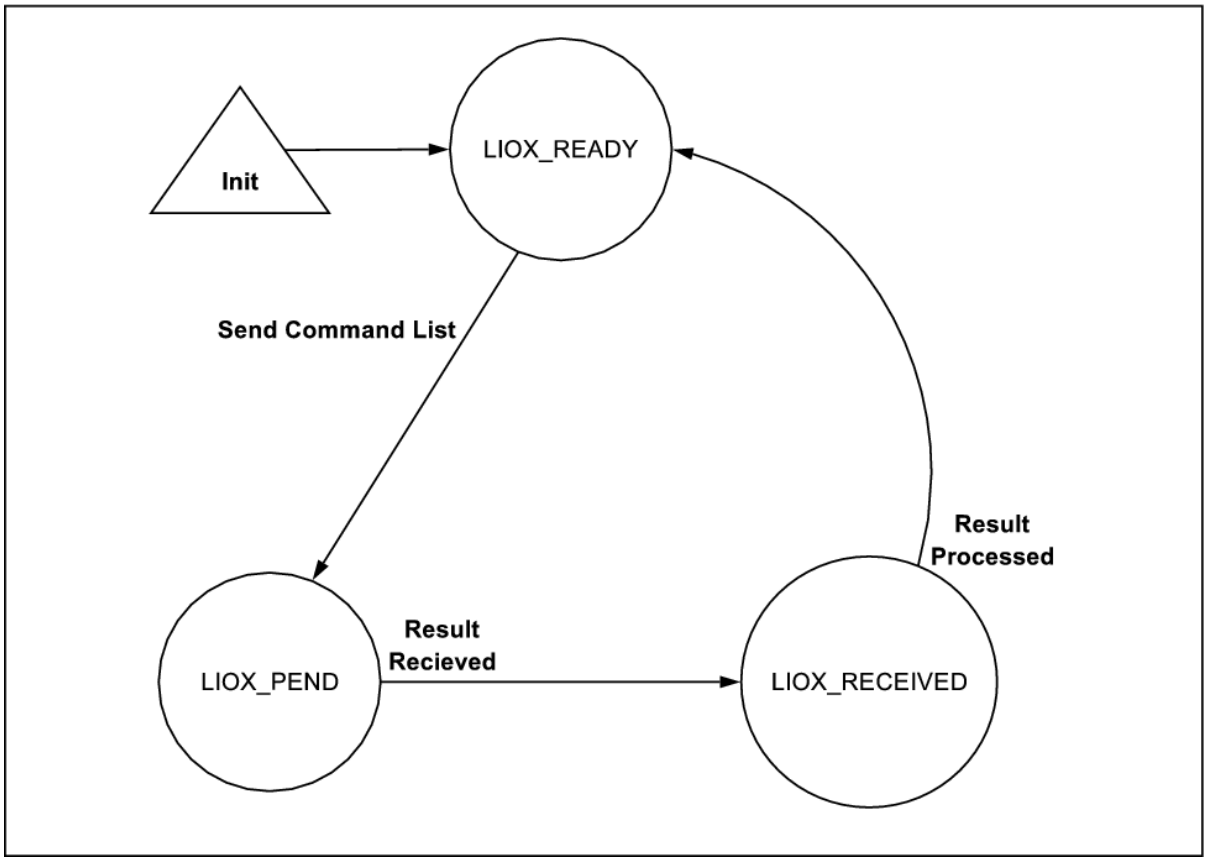

Figure 2-4 LIOX State Transitions

2.2.1.0.1 Memory Allocation

Memory allocation is left entirely to the user, i.e. the LCBD never allocates memory of its own. Memory must be allocated for the LIOX handle, the command list and the result list.

The LCB places constraints on the memory allocated for command lists and result lists. In particular the memory for the lists must be visible to the PCI bus and have the size and alignment attributes shown in Table 2-2.

Memory Space	LCB DMA	Size	Alignment
LIOX Handle	N/A	< 100 Bytes, fixed	None
Command List	Read	Up to 4092 Bytes	512 Byte
Result List	Write	Up to 4084 Bytes	8 Byte

Table 2-2 Memory alignment requirements

The LCBD provides the following memory allocators for command and result list memory.

```
    LIOX_cl *LIOX_cl_alloc( unsigned int nBytes)

    LIOX_rl *LIOX_rl_alloc( unsigned int nBytes)
```

All of the above objects could be pre-allocated to form object pools. The user could request an object from the object pool as needed.

The number of allocated lists is a function of the number of:

- Concurrently executing command lists, e.g. background housekeeping during other commanding.
- Command lists previously prepared, e.g. command lists containing default/baseline configuration commands.

After a result is received and the result is processed the user can return the objects to the appropriate object pool. Objects can be safely de-allocated according to the following table:

Object	LIOX States OK to Deallocate
LIOX Handle	LIOX_READY, LIOX_RECEIVED
Command List	LIOX_READY, LIOX_RECEIVED
Result List	LIOX_READY, LIOX_RECEIVED

Table 2-3 De-allocating Memory

It is unsafe to deallocate memory when the LIOX state is LIOX_PEND. The LCB will DMA the result into the result list whenever the result arrives. Once a LIOX is initiated the LCB "owns" the result list and command list memory until the transaction completes.

This means we cannot safely deallocate the memory for the result list or the command list until the transaction completes - if the transaction never completes we lose the memory associated with both lists and the memory for the LIOX handle.

2.2.1.0.2 Initializing a LIOX Handle

Initializing a LIOX handle requires the following steps:

- Allocate memory for the LIOX handle.
- Allocate memory for the command list.

- Declare the response handling function and callback parameter.
- Allocate memory for the result list (optional).

Allocating memory for the result list may be deferred until the command list is completely loaded, at which time the total amount of memory required for the result list is known. If, however, the LIOX will be used for synchronous operations (see Section 2.4.2) the result memory must be supplied at initialization time.

The function prototypes for the response handling function and the initialization function appear below:

```
typedef int (*LIOX_rst_cb)( LIOX             lh,
                            unsigned int status,
                            LIOX_cb_prm usrParm);

unsigned int LIOX_init( LCB             lcb,
                        LIOX             lh,
                        unsigned int cmdLen,
                        LIOX_cl         *cl,
                        unsigned int rstLen,
                        LIOX_rl         *rl,
                        LIOX_rst_cb      cb,
                        LIOX_cb_prm usrParm);
```

Currently there is no method to determine the size of the command list beforehand.

The following pseudo-code illustrates the initialization of a LIOX Handle.

```
// declare result processing callback and extra data
static   int myProcessResultFunc( LIOX lh, unsigned int status,
                                  LIOX_cb_prm usrParm);
unsigned int myExtraData;

// allocate memory for command list and LIOX struct
LIOX_cl         *pCmdList; // command list
LIOX                  lh; // LIOX    handle
unsigned  short    cmdLen; // command list length

// allocate memory for the command list -- suitably aligned
cmdLen = 4080;
pCmdList = LIOX_cl_alloc( cmdLen);

// allocate memory for the LIOX structure
lh = (LIOX)MBA_alloc LIOX_SizeOfIOX());

// create opaque handle to LIOX -- note the result list length
// is zero and the result list pointer is NULL.  These will be
// provided later when the command list is queued.
LIOX_Init( lcb, lh,
           cmdLen, pCmdList,
           0, NULL,
           myProcessResultFunc,
           myExtraData);

// LIOX Handle lh is now initialized and ready for filling
```

The allocation of the result list memory is deferred until after the command list is completely filled. At that time the required size of the result list is completely determined and the exact amount of memory can be allocated. See Section 2.2.1.2.

After the LIOX handle is initialized the internal command pointer points to the first available command slot in the command list. To reset the command pointer so that a LIOX handle can be re-filled with a new list of commands use the following function.

```
unsigned int LIOX_cl_Rewind( LIOX lh);
```

2.2.1.1 Adding Commands to the Command List

Commands are added to the command list by passing the associated LIOX handle as an argument to the family of "LAT Command Functions" which are contained in the DEM and DAB packages. The "LAT Command Functions" provide a basic interface for read/write and dataless commands to the various addressable objects within the LAT.

As of this writing 15 types of addressable objects exist within the LAT. Each object contains registers, which are also addressable. The following object types exist today:

- TEM Common Controller
- TEM Trigger Interface Controller (GTIC)

- Tracker Cable Controller (GTCC)
- Tracker Readout Controller (GTRC)
- Tracker Front End Controller (GTFE)
- Calorimeter Cable Controller (GCCC)
- Calorimeter Readout Controller (GCRC)
- Calorimeter Front End Controller (GCFE)
- AEM Common Controller
- ACD Readout Controller (GARC)
- ACD Front End Controller (GAFE)
- CRU Command Response Unit
- EBM Event Builder Module
- GEM Global Trigger
- PDU Power Distribution Unit

Most object types have three "Command Functions" - register load, register read and dataless command. That is a total of 45 functions for 15 object types.

In addition to the above command/response items are the following special command items:

- Command Fabric Reset
- Injected Time Markers
- Sending Look At Me (LAM) on the command fabric
- Accessing the LATp CSR and FIFO_FAULTS register
- Sending bulk data on the event fabric

2.2.1.1.1 Normal Command Items

The following illustrates the queuing of a normal command item used to read and write registers.

As an example consider the hypothetical case of reading a DAC on a calorimeter front end chip (GCFE). The reading of a DAC register corresponds to a register read "Command Function". To uniquely address a DAC register on a specific GCFE requires a TEM address, a GCCC address, a GCRC address, a GCFE address and the DAC register number. The GCFE read "Command Function" requires all these parameters as arguments. Below is the function prototype for `TEM_GCFE_read(),` part of the DEM package.

```
    TEM_GCFE_read( LIOX                  lh,
                   LIOX_addr          temAddr,
                   unsigned short   gcccAddr,
                   unsigned short   gcrcAddr,
                   unsigned short   gcfeAddr,
                   unsigned short      regId);
```

Calling `TEM_GCFE_read` with appropriate arguments would add a GCFE read command to the command list of the LIOX handle. The LIOX handle would update its internal command pointer to point at the next available command slot in the list. An example is shown below:

```
    unsigned int    errVal;
    LIOX      lh; /* previously initialized */

    /* queue the read command to the command list */
    errVal = TEM_GCFE_read( lh, temId, gcccId,
                            gcrcId, gcfeId, DACregId);
```

This process is repeated, adding commands to the LIOX handle until the entire command sequence is loaded or the LIOX handle runs out of memory. Next the command list is queued to the LCB for execution.

2.2.1.1.2 Command Fabric Reset

A special command item exists that instructs the LCB to assert the hardware reset line of the command fabric. This resets all the nodes of the command fabric.

The following function places a LAT Reset request on the current command list.

```
    LIOX_ci_qLATreset (LIOX lh, unsigned int timeout);
```

2.2.1.1.3 Injected Time Markers

Quoting [huffer1]:

> An application may need to specify an arbitrary amount of idle time between two export items, or to inject an accurate time marker in the RESULTS FIFO. Both of these requirements must be satisfied without actually transmitting a packet on a fabric ... A result is generated at a time determined by the stall/timeout field.

Queuing an injected marker command item takes the following form:

```
    int LIOX_ci_qMarker(LIOX lh, unsigned int timeout);
```

The stall/timeout is in units of sysclks, i.e. 50 nano-second increments.

This command generates a result item when completed.

2.2.1.1.4 LATp CSR and FIFO_FAULT Register Access

The LATp FPGA has two registers that the user can access using the following two functions.

```
    LIOX_ci_qCSR      (LIOX                  lh,
                       unsigned int         val,
                       unsigned int        mask,
                       unsigned int     timeout);

    LIOX_ci_qFIFO_FAULTS(LIOX                 lh,
                         unsigned int        val,
                         unsigned int        mask,
                         unsigned int     timeout);
```

The `val` parameter specifies a value to load into the register, while the `mask` parameter specifies which bits of the register are writable during the load operation. A user can therefore update specific bits of the registers in an atomic operation.

The result of this command contains the value of the register before the load operation and the value of the register after the load. To read the CSR use a mask of zero which will not update any bits of the register.

The FIFO_FAULTS register contains latched FIFO status information – write operation clear this register.

2.2.1.1.5 Look At Me (LAM) Command

To instruct a LATp node to switch from the primary to the redundant command path a Look At Me (LAM) command is sent by the LCB using the following function.

```
    LIOX_ci_qLAM      (LIOX                  lh,
                       LIOX_addr            addr,
                       unsigned int     timeout);
```

2.2.1.1.6 Sending Bulk Data on the Event Fabric

While the sending of bulk data occurs on the event fabric, the mechanism for queuing a bulk transfer looks like queuing a command item. The difference is the command item payload is filled with the bulk data to be sent.

The following routine is used to queue an event command item:

```
    LIOX_ci_qEvent    (LIOX                  lh,
                       LIOX_ci               *ci,
                       LIOX_addr            addr,
                       unsigned int     timeout);
```

Multiple bulk transfers may be queued in the same command list.

This command generates a result item when completed.

The following code fragment demonstrates how to queue a command item with bulk data:

```
    LIOX_ci        *ci;
    unsigned short *payload;

    status = LIOX_ci_Alloc( lh, evtSize, &ci);
    if ( _msg_failure(status)) {
      printf("Unabled to allocate %d words from lh\n", evtSize);
      return -1;
    }

    // set LATp protocol
    LIOX_ci_SetProto( ci, 0x3);

    // get a pointer to the command item payload
    // so you can fill it in
    LIOX_ci_GetPayload( ci, &payload);

    // fill the payload with some data
    for ( i = 0; i < ((evtSize * 2)-1); i++) {
      payload[i] = 0x5000 + i;
    }

    // send it
    status = LIOX_ci_qEvent( lh, ci, destAddr, 0x30);
```

2.2.1.2 Executing a Command List

Once the command list of a LIOX handle is loaded the list is ready for execution. At this time the amount of result list memory required is completely determined by the command items. The result list memory can now be allocated and assigned to the LIOX handle.

The following function is used to determine the size of a result list based on the contents of the command list. This size can be passed to `LIOX_rl_alloc()` to allocate memory for the result list.

```
    LIOX_rl_Size( LIOX lh, unsigned int *bytes)
```

Once allocated the result list memory can be attached to the LIOX handle using

```
    LIOX_rl_SetBuffer    (LIOX                lh,
                          unsigned int      nBytes,
                          LIOX_rl              *rl);
```

Finally the command list is queued to the LCB using

```
    LIOX_cl_Queue     (LIOX                 lh);
```

The following example shows how to set the result list buffer and send the request list.

```
unsigned int bytes;
LIOX_rl      *rl;

// Fetch size of result list
LIOX_rl_Size( lh, &bytes);

// allocate memory and attach it to LIOX handle
rl = (LIOX_rl *)LIOX_rl_alloc( bytes);
LIOX_rl_SetBuffer( lh, bytes, rl);

// send the list
LIOX_cl_Queue( lh);
```

Behind the scenes the LCBD performs various bookkeeping operations when queuing the command list. Refer to [huffer1] for more details on queuing command lists (referred to as export lists).

The address of the command list and the length of the command list are queued to the LCB by the LCBD. Together the length and address are called the export descriptor. The length of the list is calculated from the internal command pointer.

Once queued the state of the LIOX transitions from LIOX_READY to LIOX_PEND. Refer to Figure 2-4.

After queuing the export descriptor the LCB hardware takes over. The LCB DMAs the command list from the SBC's memory to the LCB's memory and sends the commands out on the command/response fabric of the LAT one by one.

As the results come into the LCB it stores the result items locally. After the final result comes in the LCB DMAs all the result items to the result list associated with the command list that initiated the commanding sequence. It next puts a result descriptor in the result FIFO and fires an interrupt (if the FIFO transitions from empty to non-empty).

The LCBD traps the interrupt and reads the result descriptor from the results FIFO. From the result descriptor the LCBD determines the associated LIOX handle and calls the user supplied callback routine for that handle.

2.2.1.3 Processing Results

Once called back the user navigates and processes the result list. When processing is complete the user may free the LIOX handle and the associated command list and response list.

The LCB provides two types of result items, both of which are fixed in length. The first type of result item is a simple "transmission verification" result used for command items, which by their nature have no response data. This result type contains transmission timestamp and error information.

The second type of result item is in response to a read command. In addition to the "transmission verification" data of the simple result type, the response type result item also has a payload containing protocol header information and the response returned from the target.

The next sections describe methods provided by the LCBD for navigating the results list and for decoding result items.

2.2.1.3.1 Navigating Result Lists

This section assumes that a result list has been successfully dispatched to the user application.

The LCBD provides a basic mechanism for walking or iterating over the result list, item by item. The LIOX handle maintains a result pointer (see Figure 2-3) that points to the next available result item.

The LCBD provides the following functions for result list navigation:

```
LIOX_rl_Next         (LIOX                  lh,
                      LIOX_ri             **ri);

LIOX_rl_Rewind       (LIOX                  lh);
```

The `LIOX_rl_Next()` function returns a pointer to the next result item in the `ri` parameter and updates the result pointer for the LIOX. When no more result items exist the function returns `NULL` in the `ri` parameter. The `LIOX_rl_Rewind()` resets the results pointer to the beginning of the result list.

A user can iterate over a result list as follows:

```
int myRspCallBack(
                  LIOX lh,
                  unsigned int status,
                  LIOX_cb_prm usrParm) {

 LIOX_ri *pItem = NULL;

 If ( status == LCB_OK) {
   // process result items
   LIOX_rl_Next( lh, &pItem);
   while ( pItem) {
     status = processItem( pItem, usrParm);
     LIOX_rl_Next( lh, &pItem);
   }
 }

 // At this point it is the user's choice to free the LIOX
 // and associated command list memory and result list memory.
 // You may want to keep it around for later execution.

 return status;
}
```

In the above example the while loop terminates when `LIOX_rl_Next`() sets `pItem` to `NULL.`

2.2.1.3.2 Decoding Result Items

As alluded to earlier two types of result items exist. The two types have some data fields in common while the "response" result type has more information and requires special decoding.

2.2.1.3.2.1 Common Result Item Functions

This section discusses functions used to access the common data fields for both result types.

Both result types have an opcode field (4 bits), a timestamp field (24 bits) and an error field (16 bits). The following functions retrieve these fields from a result item pointer.

```
LIOX_ri_GetOpcode    (LIOX_ri              *ri,
                      unsigned int         *op);

LIOX_ri_GetTimestamp (LIOX_ri              *ri,
                      unsigned int         *ts);

LIOX_ri_GetError     (LIOX_ri              *ri,
                      unsigned int        *err);
```

The opcode is used to determine the type of result item, either simple or response.

2.2.1.3.2.2 Response Only Result Item Functions

Result items that contain payloads in response to read commands require extra decoding. All response result items contain two fields: a LATp Header and a data payload.

The LATp header is a 16 bit field, while the data payload is a 112 bit field (7 16-bit integers).

The following functions are available for decoding response result items:

```
LIOX_ri_GetHeader    (LIOX_ri              *ri,
                      LATp_CellHeader      *ch);

LIOX_ri_GetPayload   (LIOX_ri              *ri,
                      unsigned short **payload);
```

In addition to the basic `LIOX_ri_GetPayload`() function the following helper functions exist for common payload lengths.

```
LIOX_ri_GetPayload16 (LIOX_ri              *ri,
                      unsigned short  *payload);

LIOX_ri_GetPayload32 (LIOX_ri              *ri,
                      unsigned int    *payload);

LIOX_ri_GetRegPayload(LIOX_ri              *ri,
                      unsigned int   *prev_val,
                      unsigned int    *cur_val);
```

`LIOX_ri_GetPayload16`() and `LIOX_ri_GetPayload32`() return the first 16 bits and 32 bits of the payload respectively. Most register reads fall into these two cases.

`LIOX_ri_GetRegPayload`() decodes the result for LATp CSR and FIFO_FAULT register accesses.

Other special decoders exist in the DEM and DAB packages for decoding irregular payloads including calorimeter registers, tracker registers, GTIC environmental monitoring, AEM environmental monitoring and ACD registers.

Continuing the example started in Section 2.2.1.1.1 we will decode the 16 bit GCFE DAC value within our result list callback function. Assume the result list only contains a single result item, the result item that corresponds to the GCFE read command.

```
int myRspCallBack(
                  LIOX lh,
                  unsigned int status,
                  LIOX_cb_prm usrParm) {

unsigned int        status = LCB_OK;
LIOX_ri             *pItem = NULL;
unsigned int        timeStamp;
unsigned int        errorValue;
unsigned int        opcode;
unsigned short      val;

/* fetch the first result item from the LIOX handle */
LIOX_rl_Next( lh, &pItem);

if ( pItem != NULL) {

  // get the time stamp value
  LIOX_ri_GetTimestamp( pItem, &timeStamp);

  // get the error value
  LIOX_ri_GetError( pItem, &error);

  // test if this item has a result payload -- in this contrived
  // example we already know it does have a payload.
  LIOX_ri_GetOpcode( pItem, &opcode);
  if ( opcode == LIOX_OP_CMD_RSP) {

    // By design this item is a 16 bit GCFE register
    LIOX_ri_GetPayload16( pItem, &val);

    // do something useful with register value

  }
}

return status;

}
```

2.2.2 Synchronous Command and Response Interface

The synchronous interface is used when the user wants to send a single command and is willing to block waiting for the result to come back. This interface is used to provide backward compatibility with the GNAT interface that the I&T test stands are currently using.

The synchronous interface is built on top of the asynchronous interface, hiding the details of the asynchronous interface from the user. Instead of using a LIOX handle a Synchronous LIOX (LIOXs) handle is used instead.

2.2.2.0 Synchronous LIOX Handle

The following functions are used to create a synchronous LIOX handle.

```
LIOX_sync_sizeOf       (                   void);

LIOX_sync_init         (LCB                 lcb,
                        LIOXs               slh,
                        LIOX_cl             *cl,
                        LIOX_rl             *rl);
```

`LIOX_sync_sizeOf`() returns the size of the LIOXs handle so the user is control of memory allocation. The following is an example of creating and initializing a LIOXs handle.

```
LIOXs    slh;
LIOX_rl  *rl;
LIOX_cl  *cl;

// allocate memory for synchronous LIOX handle
slh = MBA_alloc(LIOX_sync_sizeOf());

// allocate memory for command list
cl = (LIOX_cl *)LIOX_cl_alloc( 80);

// allocate memory for result list
rl = (LIOX_rl *)LIOX_rl_alloc( 80);

// initialize handle
LIOX_sync_init( lcb, slh, cl, rl);
```

The synchronous interface is much simpler than the asynchronous interface. With the synchronous interface the user only needs to call the synchronous version of a "LAT Command Function". However, the user must still allocate memory for the command and result

2.2.2.1 Synchronous Commands

The synchronous versions of the "LAT Command Functions" have similar function names and signatures as the asynchronous versions. The synchronous function names contain the string “sync”.

2.2.2.1.1 Normal Synchronous Command

Below is the function prototype for the synchronous version of the GCFE read register command discussed previously in Section 2.2.1.1.1. This function reads a register on a particular GCFE.

```
TEM_GCFE_read_sync (LIOXs                slh,
                    LIOX_addr          temAddr,
                    unsigned short    gcccAddr,
                    unsigned short    gcrcAddr,
                    unsigned short    gcfeAddr,
                    unsigned short       regId,
                    unsigned short        *val);
```

With the synchronous interface the entire process of sending a command, waiting for a result and decoding a result is reduced to a single synchronous function call. Reading a calorimeter register is implemented as follows:

```
unsigned int        errVal;
unsigned short int  regVal;

/* read the register value – blocking */
errVal = TEM_GCFE_read_sync( slh, temId, gcccId, gcrcId,
                             gcfeId, DACregId, &regVal);

/* do something useful with register value */
```

2.2.2.1.2 Synchronous Command Fabric Reset

The following function synchronously sends a LAT Command Fabric reset.

```
LIOX_sync_LAT_RESET    (LIOXs              slh);
```

2.2.2.1.3 Synchronous LATp CSR and FIFO_FAULT Register Access

The following functions synchronously access the LATp CSR and FIFO_FAULT registers.

```
LIOX_sync_CSR          (LIOXs              slh,
                        unsigned int    new_val,
                        unsigned int       mask,
                        unsigned int   *cur_val,
                        unsigned int   *old_val);

LIOX_sync_FIFO_FAULTS  (LIOXs              slh,
                        unsigned int    new_val,
                        unsigned int       mask,
                        unsigned int   *cur_val,
                        unsigned int   *old_val);
```

Upon return the `cur_val` and `old_val` parameters are filled in with the current and old values of the registers.

2.2.2.1.4 Synchronous Look At Me (LAM) Command

The following function synchronously sends the Look At Me command.

```
LIOX_sync_LAM          (LIOXs               slh,
                        LIOX_addr          dest);
```

2.3 Unsolicited Data Interfaces

As shown in Figure 2-1 the LCBD routes unsolicited data based upon the LATp protocol. This implies an interface by which the user registers a handler for a particular LATp protocol.

Once the handler is invoked the user needs interfaces for navigating the unsolicited data and for freeing the circular buffer memory used by the unsolicited data.

2.3.0 Registering Call Back Handlers

The LCBD provides an interface for registering handlers based on a specific LATp protocol. The declaration of a call back handler is shown below:

```
typedef void* LCB_cb_prm;

typedef unsigned int (*LCB_evt_cb)( LCB_cb_prm parm,
                                    LCB_evtDesc ed,
                                    LCB_msg *msg);
```

The declaration of the handler registration function is shown below:

```
LCB_bulk_cb_set         ( LCB                lcb,
                          unsigned short    proto,
                          LCB_evt_cb           cb,
                          LCB_cb_prm         prm);
```

The proto parameter takes on the values from 0 to 3, covering the four possible LATp protocols.

2.3.1 Navigating Unsolicited Data

The user needs a way to navigate the unsolicited data inside of the call back handler. From the LCB_evtDesc and LCB_msg parameters the user can extract the following information about the unsolicited data:

- Data Length
- Error Status
- LATp Cell Header
- Pointer to bulk data

2.3.1.0 LCB Event Descriptor

The LCB_evtDesc is a union of an unsigned int and a struct bit-field map of the raw 32-bit descriptor read from the LCB hardware. The bit fields of the descriptor are shown below.

```
#include “LCB/LCB_latp.h”

struct _BFLCB_evtDesc {
  unsigned int    rstatus: 2; // LATp receive status
  unsigned int    xstatus: 3; // PCI  transnsfer status
  unsigned int        len:10; // Packet length, # of 32-bit words
  unsigned int     offset:17; // Circular buffer offset in words
};
```

The possible values for rstatus bit-field are the LCB LATp Receive Errors described previously in Section 1.2.0.0.3.

The possible values for the xstatus bit-field are the LCB PCI Export Errors described previously in Section 1.2.0.0.2.

The offset bit-field is not important to the handler writer since it is a pointer to the actual data and that information is conveyed by the LCB_msg structure discussed next.

2.3.1.1 LCB Message

The LCB_msg structure consists of two sub-structures defined next.

2.3.1.1.1 LCB_msg Private Header

The private message header is the “4 word” gap placed before each unsolicited message in the circular buffer by the LCB hardware. Essentially these 4 words are used by software for internal purposes without the need for software to allocate any memory.

The definition of these four words is shown here.

```
struct _LCB_msg_priv_hdr {
  FORK_msg_hdr          fhdr; // Not used, could be reclaimed
  LCB_mark              mark; // Used to control ownership
  unsigned int        unused; // Pads struct to 16 bytes
};
```

Currently the FORK_msg_hdr (8 bytes) is not used and could be reclaimed for other purposes.

The LCB_mark is written by the driver when the event descriptor is read from the event FIFO. The LCB_mark is used when freeing and LCB message. The bit-fields of the mark are shown here.

```
struct _BF_LCB_mark {
  unsigned int       free: 1; // set when message freed
  unsigned int     unused:21; // not used
  unsigned int        len:10; // Packet length, # of 32-bit words
};

union _LCB_mark {
  unsigned int          ui;
  struct _BF_LCB_mark   bf;
};

typedef union _LCB_mark LCB_mark;
```

The `len` bit-field is a copy of the `len` bit-field from the event descriptor.

2.3.1.1.2 LATp Contribution Header

The LATp contribution header is the first 32-bit word of the DMA data from the LCB. The most significant 16-bits are the LATp Cell Header and the least significant 16-bits are the contribution status from the event builder module.

The definitions of the cell header, contribution status and contribution header are shown next.

```
struct _BFcellHeader {
  unsigned short         rsp:1; // does packet expect a response
  unsigned short        dest:6; // 6 bit LAT destination address
  unsigned short       proto:2; // 2 bit LATp protocol
  unsigned short      source:6; // 6 bit LAT source address
  unsigned short      parity:1; // odd partiy of the header
};
union _cellHeader {
  unsigned short        ui;
  struct _BFcellHeader  bf;
};
typedef union _cellHeader LATp_CellHeader;

struct _BF_LATp_CntrbStatus {
  unsigned short        err:3; // 3 bit contrib error
  unsigned short        seq:5; // 5 bit contrib sequence number
  unsigned short        len:8; // 8 bit length, # of 128-bit cells
};
union _LATp_CntrbStatus {
  unsigned short               ui;
  struct _BF_LATp_CntrbStatus  bf;
};
typedef union _LATp_CntrbStatus LATp_CntrbStatus;

struct _LATp_cntrb_hdr {
  LATp_CellHeader     ch;  // 16-bit LATp cell header word
  LATp_CntrbStatus    cs;  // 16-bit LATp contribution status
};
typedef struct _LATp_cntrb_hdr LATp_cntrb_hdr;
```

2.3.1.1.3 Complete LCB_msg Structure

The entire `LCB_msg` structure is shown next.

```
struct _LCB_msg {
  LCB_msg_priv_hdr      phdr; // Private LCB message header
  LATp_cntrb_hdr        chdr; // LATp contribution header
  unsigned int          data; // First word of data
};
typedef struct _LCB_msg LCB_msg;
```

Given a pointer to `LCB_msg` the user obtains a pointer to the DMA data like this:

```
unsigned int  *pData;

pData = &msg->data;
```

2.3.2 Freeing Unsolicited Data

The unsolicited data arrives in a circular buffer managed by the LCB (write pointer) and the LCBD (read pointer). Within the handler function the user controls when the unsolicited data is freed by calling the LCB_bulk_msg_free() routine shown below.

```
LCB_bulk_msg_free( LCB lcb, LCB_msg *msg);
```

It is important that the user return the memory to the circular buffer as quickly as possible. If the user operation will "take a long time" the user is obligated to make a copy of the unsolicited data and return the unsolicited data memory immediately. It remains to be seen what "a long time" is.

2.3.3 Example Handlers

Below is an example of a quick handler. This example assumes that the user parameter is a pointer to a user-defined structure that contains (among other members) the LCB handle.

```
int my_LCB_HandlerQuick( LCB_cb_prm parm,
                         LCB_evtDesc ed,
                         LCB_msg *msg ) {

USER_STRUCT *pUser  = (USER_STRUCT *)parm;
unsigned int status = LCB_OK;

// check event descriptor status
if (!ed.bf.rstatus && !ed.bf.xstatus) {
  // our processing is quick, so just do it
  status = processData( pUser, msg);
}

// free unsolicited data
LCB_bulk_msg_free ( pUser->m_lcb, msg);

return status;
}
```

Below is an example of a slow handler, which makes a copy of the unsolicited data.

```
int my_LCB_HandlerSlow ( LCB_cb_prm parm,
                         LCB_evtDesc ed,
                         LCB_msg *msg ) {

USER_STRUCT *pUser  = (USER_STRUCT *)parm;
unsigned int status = LCB_OK;

// check event descriptor status
if (!ed.bf.rstatus && !ed.bf.xstatus) {

  // our processing is slow, so make a copy using memcpy.
  // copy data to previously allocated memory, pointed to
  // by the "copy_mem" pointer.  The event descriptor contains
  // the event length in 32-bit words – add four for the private
  // header (4 words) and multiply by 4 to get bytes
  memcpy( copy_mem, (void *)msg, (ed.bf.len + 4) * 4);

  // free the unsolicited data
  LCB_bulk_msg_free( pUser->m_lcb, msg);

  // process the copy
  status = processData( pUser, copy_mem);
}
else {
  // Errors. Just free unsolicited data
  LCB_bulk_msg_free( pUser->m_lcb, msg);
}

return status;
}
```

2.4 User Solicited Commands

User solicited commands are commands that the user wants to insert into the ISR Dispatch Queue (see Figure 2-1). This allows the user interlocked access to handler data structures. This is best explained by a motivating example involving the event filter.

When the event handler is installed a user call back parameter is also specified. Usually this is a pointer to a user defined data structure that the user wants to access during the event handler callback routine. In the case of the event filter this user data structure may contain histogram objects or other statistic counters.

How can the user read these structures outside the context of the callback in a thread safe way? If you are reading (or modifying) these structures you want to be certain the event handler callback is not currently accessing the data.

Recall the FORK queue used for ISR Dispatch (see Section 1.1.1.3.2) is created outside of the LCBD by the system. The LCBD only uses queue 0 of the FORK queue – the FORK queue could have additional queues. By queuing a message (here the “message” is a callback function and the message data) to one of the other queues (or indeed by queuing to queue 0) the user can synchronize access to the shared data.

In fact the LCBD uses this technique internally to read and clear the statistics of Section 1.2 in an interlocked fashion.

In addition the LCBD provides a function to “kick” the ISR dispatch in order to force processing of the event FIFO. Remember the LCB interrupt can be configured to go off at various high water marks of the event FIFO and circular buffer. Imagine at the end of a run the global trigger is shutdown so no more events enter the LCB, but several hundreds of event descriptors remain in the event FIFO since the high water mark has not been hit.

The following function is used to force the event queue processing.

```
LCB_isr_pump( LCB  lcb);
```

3 Polled Mode Driver

The polled mode LCBD is used by the LAT EPU boot application when the services of a RTOS are not available. During EPU boot the polled mode LCBD is limited to sending and receiving traffic on the event data fabric only. The polled mode LCBD will not send or receive data on the command/response fabric.

In the absence of an RTOS the processing of interrupts will be not be possible. Therefore the polled mode LCBD must be driven by the EPU boot application's event poll loop. The polling interface notifies the boot application when a new response is ready for processing.

The sending interface allows the boot application to send bulk data between CPU crates on the event fabric.

3.0 Driver Libraries

The polled mode LCBD interface presents a public, stable interface to a single user, the LAT boot application. The polled mode LCBD interface depends on the LCB hardware library and PBS (see Section 1).

3.1 Driver Interface

The polled mode LCBD provides the following services:

- Driver initialization.
- Dispatching unsolicited LCB-to-LCB data.
- Sending unsolicited packets on the LATp event fabric.

3.1.0 Driver Initialization

Before beginning operation the polled mode LCBD requires several chunks of "scratch pad" memory, which must be provided by the boot application. The required memory sizes and characteristics are shown below:

Memory Space	LCB DMA	Size	Alignment
LCB Handle	N/A	< 100 Bytes, fixed	None
Command List	Read	Up to 4092 Bytes	512 Byte
Result List	Write	Up to 4084 Bytes	16 Byte
Circular Buffer	Write	640KB	1MB

Table 3-1 Polled mode LCBD Memory Attributes

The LCB handle is used to manage the LCB.

A single command list is required to send LCB-to-LCB data on the event fabric.

A single result list is required because the LCB will generate a simple response to the bulk data transfer request.

The circular buffer is required to receive unsolicited data on the event fabric.

The function prototype for initializing the polled mode LCBD is shown below.

```
LCB_poll_init      ( LCB              lcb,
                     LCB_REG_CSR    pciCSR,
                     LIOX_CSR      latpCSR,
                     LIOX_cl           *cl,
                     LIOX_rl           *rl,
                     unsigned int circAddr);
```

The `pciCSR` parameter allows the user to set the initial value of the PCI CSR, which would be used to select the primary/redundant event fabric.

The `latpCSR` parameter allows the user to set the initial value of the LATp CSR, which would be used to set the LCB's address.

This call returns an initialized LCB handle. The `cl` parameter is a pointer to a 4KB memory chunk to use for the command list. The `rl` parameter is a pointer to a 4KB memory chunk to use for the result list. The `circAddr` parameter is the base memory address to use for the circular buffer.

At this point the LCB is completely configured. The reception of unsolicited data is enabled by calling LCB_poll_start() shown below.

```
unsigned int LCB_poll_start(LCB lcb);
```

See Section 3.2.0 for an example of initializing the polled mode driver.

3.1.1 Dispatching Unsolicited Data

This section leverages much of the machinery described previously for unsolicited data handling, see Section 2.3.1. The major difference is that the interrupt handler and callback routines are replaced with a simple polling loop.

The LAT boot application will poll the LCBD looking for new unsolicited data by calling the LCB_poll_query() function.

```
unsigned int LCB_poll_query(LCB         lcb,
                            LCB_msg   **msg,
                            LCB_evtDesc *ed);
```

When new LCB-to-LCB data is available this function returns non-zero and the LCB_msg argument is set to a non-NULL LCB_msg pointer and the event descriptor is filled in. When no data is available, LCB_poll_query() returns zero.

Decoding the LCB_msg pointer proceeds as previously described in Section 2.3.1. The big difference is that the decoding occurs within the polling loop, not in a user callback routine. See Section 3.2.1 for an example of the polled mode interface.

3.1.2 Sending Bulk Data

Sending bulk data on the event fabric is very straightforward. The maximum length of a bulk data item is 4080 bytes. Most of the discussion on sending bulk data using the ISR driver applies to the polled mode driver as well (see Section 2.2.1.1.6). The major difference is that the polled mode LCBD can only queue a single bulk transfer at a time.

To avoid copying data the user first allocates a command item, fills it in and then sends it. The command item is allocated using the following function.

```
LCB_poll_ci_Alloc  ( LCB              lcb,
                     LIOX_ci         **pCI,
                     unsigned int   nWords);
```

The LCB_poll_send() function is used to send bulk data.

```
LCB_poll_send_bulk ( LCB              lcb,
                     LIOX_ci          *pCI,
                     LIOX_addr        addr,
                     unsigned int    proto);
```

See Section 3.2.2 for an example for sending bulk data with the polled mode driver.

3.2 Example Driver

This section covers examples for initializing the polled mode LCBD, for dispatching results in a polled event loop and for sending bulk data.

3.2.0 Initialization

An example of initializing the polled mode LCBD follows:

```
int initLCB(LCB *plcb) {

int                status  =  LCB_OK;
LIOX_cl            *cl;       // command list
LIOX_rl            *rl;       // result  list
unsigned int       baseAddr;  // base address of circular buffer
LCB_REG_CSR        pciCSR;
LIOX_CSR           latpCSR;

// allocate memory for the command list -- suitably aligned
cl = LIOX_cl_alloc( 4096);

// allocate memory for the result  list -- suitably aligned
rl = LIOX_rl_alloc( 4096);

// allocate memory for the LCB handle
*plcb = MBA_alloc( LCB_sizeOf());

// assign base address for circular buffer
baseAddr = LCB_CIRC_BASE_ADDR;

// assign primary cmd path
pciCSR.ui = 0x0;
pciCSR.bf.cmdPath = 0x0;

// assign LCB address, set primary event path
latpCSR.ui = 0x0;
latpCSR.bf.boardID = LIOX_addr_EPU0;
latpCSR.bf.evtPath = 0x0;

// initialize the LCB handle
status = LCB_poll_init( *lcb, pciCSR, latpCSR, cl, rl, baseAddr);
if (_msg_failure(status)) {
  MBA_free(lcb);
  MBA_free(pRstList);
  MBA_free(pCmdList);
}

// the LCB is now initialized, but not enabled

return status;

}
```

3.2.1 Polled Event Loop

This section describes the polled event loop. First the reception of unsolicited data is enabled and then the driver enters an infinite loop waiting for LCB messages. As messages are found they are processed.

```
unsigned int eventLoop( void *usrData) {

unsigned int        status = LCB_OK;
LCB_msg             *msg;
LCB_evtDesc         ed;

// enable the LCB for receiving event data
status = LCB_poll_start( lcb);

if ( _msg_failure(status)) {
  return status;
}

// enter infinite loop
while ( 1) {

  if (LCB_poll_query( lcb, &msg, &ed)) {
    // message found, process it.

    if ( ed.bf.rstatus || ed.bf.xstatus) {
      // record/report error
      status = ed.ui;
    }
    else {
      // proces the message.  The user could also make a copy of
      // the bulk data at this time.
      status = processData( usrData, msg);
    }

    // free unsolicited data
    LCB_bulk_msg_free( msg);

  }

// sleep, or poll other devices, etc...

}

return status;

}
```

3.2.2 Sending Bulk Data

This section describes sending bulk data on the event fabric.

```
unsigned int sendData( LCB *lcb) {

unsigned int   status = LCB_OK;
short          destAddr = LIOX_addr_SIU0  // dest LATp node address
short          proto = 3;       // LATp prototcol to use
LIOX_ci        *pCI;
unsigned short *payload;

// Allocate 4 CELL Message
LCB_poll_ci_alloc( lcb, &pCI, 4 * 4);

// Retreive pointer to command item payload
LIOX_ci_GetPayload( pCI, &payload);

// Fill payload with interesting data
...

// Send message
status = LCB_poll_send_bulk( lcb, pCI, destAddr, proto);

return status;
}
```

4 To Do List

This section outlines changes to the LCB driver that should be made.

4.0 Remove LCB Handle Parameter

The LCB software interface contains functions of the following form:

```
LCB_xxx( LCB lcb, … );
```

As the LCB is a singleton (only one LCB per CPU) the LCB handle parameter is extraneous and should be removed from all the interface routines. The LCB private data could be stored as a static block within the library and all the interface functions could refer to the block via a pointer. This removes the necessity of passing the LCB handle to all the interface functions.

This change, while not complicated, would be a bit tedious to implement as the current interface is in wide spread use.

4.1 LCB_isr_fork_set

This section refers to section 1.1.1.3.2 about the `LCB_isr_fork_set`() function.

The LCB driver does not need access to the entire FORK object, it only needs access to a specific FORK queue. This gives the system more control over the creation of the FORK object. This interface should be changed to the following:

```
LCB_isr_fork_set( FORK_que *que);
```

This change would require a very small amount of coding.

4.2 Assembly of Fragmented LCB Messages

The event builder module has the ability to fragment LATp packets during transmission whenever the receiving LCB asserts the `pause` line. During a transmission if the LCB asserts `pause` the

EBM completes sending the current cell, marks it as truncated and ceases transmission. After `pause` is de-asserted the EBM resumes packet transmission. Each packet fragment has a monotonically increasing sequence number (starting with zero) contained in the 16-bit EBM status word. The last packet is not marked as truncated.

The assembly of these fragments has not been implemented in the `LCBD`. At the moment the problem is pushed out to the event handler. It seems possible to use the 4-word private area to contain link information to aid fragment assembly – a linked list of fragments.

The linked list approach is probably the least traumatic since it gracefully handles the case where the fragments wrap around the end of the circular buffer.

Fragmented packets pose a performance problem for the event filter since the decoding would not be straight forward in this case. For the case of normal event filtering it is presumed that large packets requiring fragmenting would be somewhat rare and could be deferred to a lower priority decoder. Once the complete linked event is ready you could copy the fragments to another location to create one contiguous event and process it from there. After the copy you could free the fragments before processing the copy.

For calibration, however, large packets would be the normal and an alternate strategy might be better.

4.3 LCB Responder / LAM

Recent hardware has the ability to response to the “Look At Me” command, sent by a commander LCB to a responder LCB. This allows the SIU (commander) to program the LATp addresses of the EPUs (responders).

Currently no software interface has been written to send these commands.

4.4 Determining Command List Size

Currently no mechanism exists to pre-determine the size of a command list. For instance you want to create a list containing three items – no interface exists to tell you how big to make the command list to hold the three items.

The problem is mildly complicated because command items are variable size. A reasonably approach would be to assume a maximally size command item and allow the user to request a command list that would hold N maximally sized command items.

This approach could waste a little space (0 to 8 bytes) per command item depending on the actual commands placed on the list.

An alternative would be to always allocate a maximally size command list (4KB, enough to hold 150 maximally sized command items) and use that. If you only wanted to send three items, however, this approach would waste almost the entire 4KB.

5 References

[huffer1] Michael Huffer, LAT Communication Board: LCB Design Specification, LAT-DS-00639.

[huffer2] Michael Huffer, LAT Inter-module Communications: A reference manual,LAT-DS-00606.

[huffer3] Michael Huffer, The Tower Electronics Module (TEM): A Primer,LAT-DS-00605.

[huffer4] Michael Huffer, The ACD Electronics Module (AEM): A Primer,LAT-DS-00639.

[sa99] Tom Shanley and Don Anderson, PCI System Architecture, 4th Edition, ISBN 0-201-30974-2, MindShare, Inc., 1999.

[bae1] BAE SYSTEMS RAD750TM Board: Software User's Manual, BAE Systems, 234A535, 2001.

[bae2] BAE SYSTEMS RAD750TM Board: Hardware User's Manual, BAE Systems, 234A533, 2000.

[sib] GLAST/LAT Spacecraft Interface Board (SIB), Hardware Specification, LAT-SS-01792.