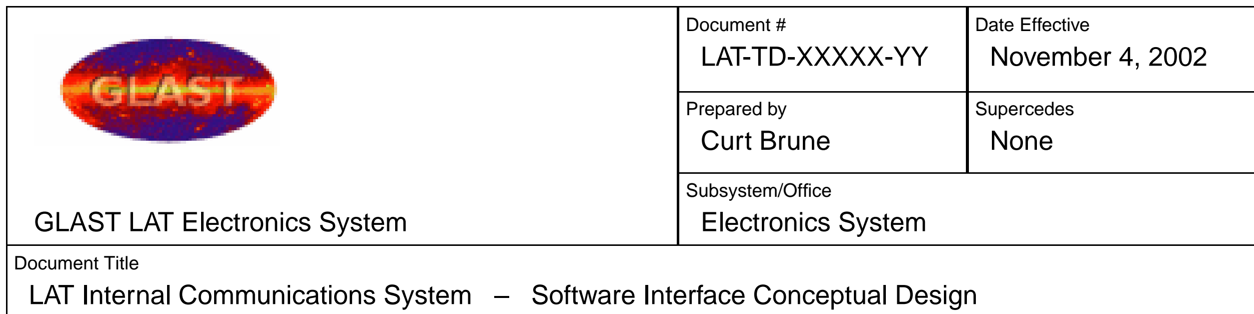

GLAST LAT Electronics System	Document # LAT-TD-XXXXX-YY	Date Effective November 4, 2002
	Prepared by Curt Brune	Supercedes None
	Subsystem/Office Electronics System	
Document Title LAT Internal Communications System – Software Interface Conceptual Design		

LAT Internal Communications System

Software Interface Conceptual Design

LAT Internal Communications System: Software Interface Conceptual Design

by Curt Brune

Published November 4, 2002

This document provides an overview and conceptual design of the software interface for the LAT Internal Communications System (`LICS`).

This document is also available in the following formats:

- PDF
- One page html
- Text

Revision History

Version	Date	Comment
0.3	July 25, 2002	Added Command and Response Chapter.
0.2	July 18, 2002	Moved LAT System Initialization to Separate Document.
0.1	July 11, 2002	Initial draft.

Table of Contents

Introduction ix

1. Definitions ix
2. Assumptions ix

1. LICS Initialization 1

1.1. Orienteering – Where am I and Who's on Top? 1
1.1.1. SBC Enumeration 1
1.1.2. Memory Maps 1

2. Commands and Results 3

2.1. Asynchronous Command and Response Interface (ACRI) 3
2.1.1. Preparing for Commanding 4
2.1.2. Adding Commands to the Command List 6
2.1.3. Executing a Command List 7
2.1.4. Processing Results 8
2.2. Synchronous Command and Response Interface (SCRI) 10

3. Event Processing 13

3.1. LCB Event Processing Model 13
3.2. Hardware Initialization 13
3.3. LEPI Initialization 14
3.3.1. Error Handler 14
3.3.2. Event Handler 14
3.4. Event Processing 15
3.4.1. Error Handling 15
3.4.2. Event Handler 15

A. Result Processing Flow Diagram 17

Background Documentation 19

List of Tables

2-1. LIOX List Memory Attributes 4

List of Figures

1. Layout of the PCI bus in a single cPCI crate. ix
2-1. Overview of LIOX data structure. 4
A-1. Result Processing Flow Diagram 17

List of Examples

2-1. Reading Calorimeter DAC 6
2-2. Reading a Calorimeter DAC – Continued 9
2-3. Read Calorimeter DAC – Synchronous Interface 11

Introduction

The LAT Internal Communications System (LICS) provides communications between the modules within the LAT's cPCI crates and the nodes of the LAT's command and event fabrics. This document describes the software interface of the LICS as seen by flight software developers.

1. Definitions

The following definitions will apply throughout this document.

LICS

LAT Internal Communications System

LSI

LAT System Initialization

SIB

Spacecraft Interface Board

LCB

PCI LAT Communications Board

SBC

Single Board Computer, generically referring to either a Director SBC or a Actor SBC (see below).

Director SBC

Used as a synonym for master SBC, but avoiding all the other meanings of the word master. Think Stanley Kubrick and "2001: A Space Odyssey". See definition of Actor SBC.

Actor SBC

Used as a synonym for slave SBC, but avoiding all the other meanings of the word slave. Think "Kevin Spacey". See definition of Director.

2. Assumptions

This document assumes that all modules in the cPCI crate are properly initialized as described in [8].

The LAT Internal Communications System (LICS) is compiled code for the VxWorks operating system that resides on each of the SBCs in the system.

A simple diagram of this setup is shown below.

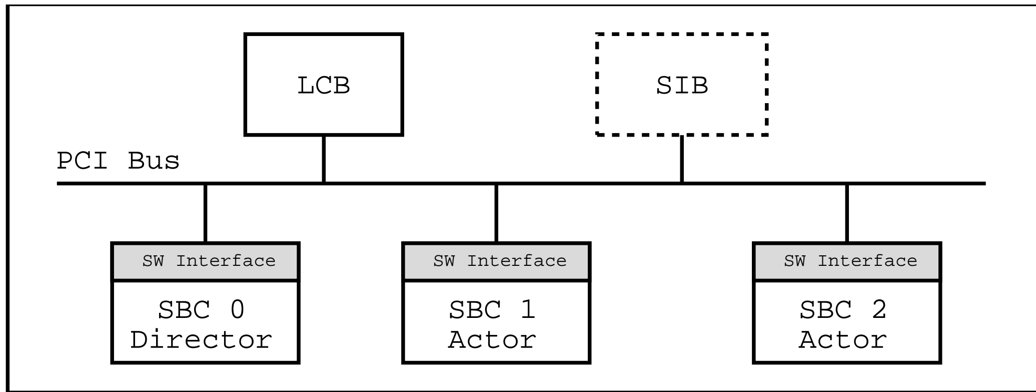

Figure 1. Layout of the PCI bus in a single cPCI crate.

Chapter 1. LICS Initialization

This section describes the initialization of the LICS, including both an overview of the configuration and an API for configuration.

1.1. Orienteering – Where am I and Who's on Top?

The LCB provides access to the LAT Internal Communications System (LICS); for up to three SBCs. The access is provided via a memory-mapped interface, where each SBC accesses a certain portion of the LCB's memory space. For the system to function correctly a SBC must access that LCB memory which pertains to it, without accessing memory allocated for another SBC.

The LCB exposes memory-mapped registers for low-level initialization and and confifuration, which only a *single* SBC, the Director SBC, should ever access. As described in [8] the Director SBC is the SBC that resides in the system slot of the cPCI crate.

1.1.1. SBC Enumeration

In the LICS the SBCs must be enumerated unambiguously. This is acccomplished by each SBC inspecting the cPCI bus to determine which *physical* slot it resides in.

The SBCs can also determine the PCI slot positions of every *other* SBC in the crate by leveraging the fact that all the SBCs are homogeneous in hardware . Since every SBC has the same PCI host-bridge chip with the same PCI DeviceId and VendorId a SBC can locate every other SBC.

Once all the SBCs are located they can enumerate themselvs from 0 to 2 starting with the Director SBC at 0 and increasing with increasing PCI slot number.

This enumerates the SBCs from 0 to 2 regardless of which PCI slot the SBC occupies (except for the Director SBC which must reside in the cPCI system slot).

1.1.2. Memory Maps

The LICS defines three different classes of PCI memory. Two of the memory classes map to physical memory on the LCB, while the third class of memory maps to physical memory on each SBC.

A SBC will use its enumeration to setup it memory maps and avoid conflicting with the other SBCs.

The LCB exposes three sets of registers for event data and command/response data, one set for each of the three SBCs. For event data the registers are EVENTS_BASEn and EVENTS_FREEn, while for command/response the registers are EXPORTn and RESULTSn with n ranging from 0 through 2 for all sets.

A SBC will use its enumeration (see Section 1.1.1) as an index into these register sets, thus avoiding any conflicts.

Chapter 2. Commands and Results

This section describes the sending of commands and the recieving of results within the `LICS`. Commands are bit sequences sent down to the front end electronics (register reads, register writes and dataless commands). Every command generates a result within the `LICS` – 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 with in this section[1]

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.

I/O Transaction

An indivisable 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 the `LCB` sends a notification message to the driver. This arrangement allows for asynchronous commanding, i.e. a user can queue a command list to the `LCB`, continue to do other work and then be notified asynchronously that the queued commands are completely processed.

For the case when a user only wants to send a *single* command it is purposed to also offer a *synchronous* interface to the `LICS`. With this interface a user would queue a *single* command to the `LCB` and wait for the command to complete. At this time it appears straight forward to build the synchronous interface on top of the asynchronous interface.

2.1. Asynchronous Command and Response Interface (`ACRI`)

The `ACRI` offers a powerful way for users to queue command lists to the `LCB` for processing. While the `LCB` is busy processing the commands list and recieving result items the user can continue to do other work. The `LICS` notifies the user after the final command item and result item are processed. After notification the user would proceed to inspect the result list.

The `LICS` provides a data structure for organizing the command list and the result list called the LAT I/O Transaction (`LIOX`). Users communicate with the `LICS` via an opaque handle to a `LIOX` structure.

1. All definitions, package names, function names and terminology are subject to change at the whim of the author.

A diagram showing the relationship between the command list, result list and LIOX structure is shown below in Figure 2-1.

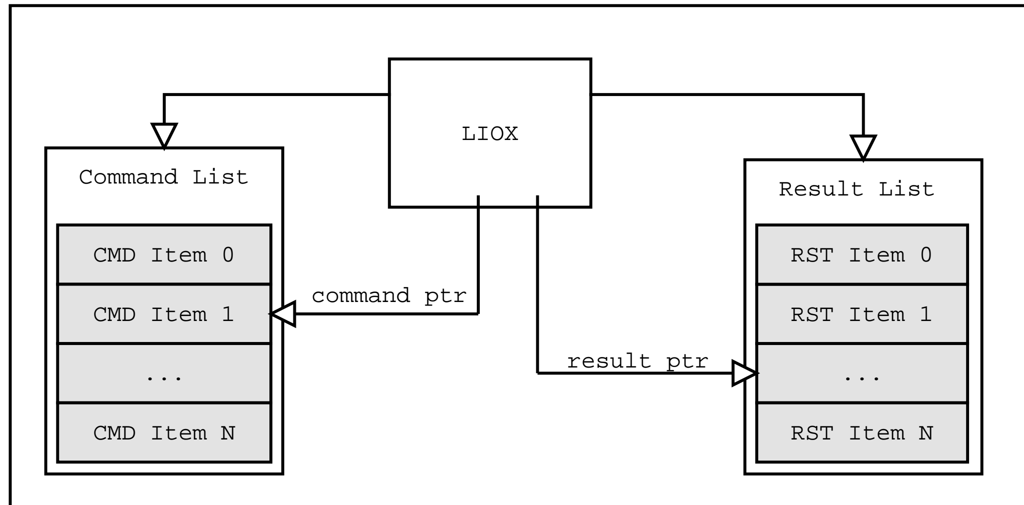

Figure 2-1. Overview of LIOX data structure.

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 in a file system – 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.

2.1.1. Preparing for Commanding

Before creating and sending command lists via the LICS 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 by the LICS when signaled by the LCB that a result list is ready.
- Initialize the LIOX by passing in pointers for the command list memory, for the result list memory and for the result processing function.

2.1.1.1. Memory Allocation

Memory allocation is left entirely to the user, i.e. the LICS never allocates memory of its own. The LICS uses memory previously allocated by the user.

The LCB places constraints on the memory allocated for command lists and result lists. In particular the memory for the lists must be visable to the PCI bus and have the size and alignment attributes shown in Table 2-1.

Draft for internal use only

Memory Space	LCB DMA	Size	Alignment
Command List	Read	Up to 4092 Bytes	512 Byte
Result List	Write	Up to 4084 Bytes	8 Byte

Table 2-1. LIOX List Memory Attributes

It may be worthwhile for the LAT system software to offer memory allocators for different alignment requirements.

2.1.1.2. User Defined Result Processing Function

The user defined result processing function is called by the LICS after a command list is sent and the corresponding result list is ready for processing. From the user's point of view this function is called asynchronously, notifying the user that a result list is ready for processing.

The LIOX maintains a pointer to the result processing function.

For more information on result processing and the result processing function see Section 2.1.4.

2.1.1.3. Initializing a LIOX Handle

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

```
/* define result processing function */
static int myProcessResultFunc(...);

/* allocate memory for command list, result list and LIOX struct */
void        *pCmdList; /* command list */
void        *pRstList; /* result list */
void        *pLIOX;
LIOXHandle  lh;

/* allocate memory for the command list - suitably aligned */
pCmdList = mallocCommandList();

/* allocate memory for the result list  - suitably aligned */
pRstList = mallocResultList();

/* allocate memory for the LIOX structure */
pLIOX = mallocLIOX( liox_SizeOfIOX());

/* create opaque handle to LIOX
lh = liox_Init( pLIOX, pCmdList, pRstList, myProcessResultFunc);

/* LIOXHandle lh is now initialized and ready for use */
```

> **Note:** The liox_Init() function may also take the sizes of the command list and result list as arguments. This would allow the LICS to perform bounds checking as commands are added to the command list.

After the LIOX handle is initialized the internal command pointer points to the first available command slot in the command list. A mechanism exists for reseting the command pointer so that a LIOX handle can be *re-filled* with a new list of commands.

2.1.2. 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". 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 11 types of addressable objects exist within the LAT – this number is expected to grow to about 20 types of objects. 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)

For each object type there exists three "Command Functions" – register load, register read and dataless command. That is a total of 33 functions for 11 object types. A rough estimate projects 50-60 functions (about 20 object types) in the final system.

Example 2-1. Reading Calorimeter DAC

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 liox_qGCFEload.

```
int   liox_qGCFEread(   LIOXHandle          lh,
                        short            temId,
                        short           gcccId,
                        short           gcrcId,
                        short           gcfeId,
                        short            regId );
```

Calling `liox_qGCFEread` 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;
LIOXHandle      lh; /* previously initialized */

/* queue the read command to the command list */
errVal = liox_qGCFEread( 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.

2.1.3. Executing a Command List

Once the command list of a `LIOX` handle is loaded the list is ready for execution. Queueing the command list to `LCB` is trivial from the user's perspective. A simple execute function, like the one below, is all that is needed.

```
int   liox_qCmdList(   LIOXHandle      lh );
```

Behind the scenes, however, the `LICS` performs various bookkeeping operations when queueing the command list. Refer to [1] for more details on queueing 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 `LICS` – together the length and address are called the `export descriptor`. The length of the list is calculated from the internal command pointer.

After queueing the `export descriptor` the `LCB` 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 for the `SBC` and fires an interrupt.

The `LICS` traps the interrupt and reads the `result descriptor` from the results FIFO. The `LICS` uses the `result descriptor` to locate the appropriate `LIOX` handle. From the `LIOX` handle the `LICS` calls the user supplied result processing call-back function.

[Note to self: what might the return code from the user supplied call-back be used for ?]

2.1.4. Processing Results

Once a result list is ready the user needs a way to process the result list and decode the result items within the 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 both load commands and dataless commands, which by their nature have no response data. This result type contains a 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 value read from the register.

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

2.1.4.1. Navigating Result Lists

The `LICS` provides a basic mechanism for walking or iterating over the result list, item by item. The `LIOX` handle maintains a `result pointer` (see Figure 2-1) that points to the next available result item.

The `LICS` provides the following functions for result list navigation:

```
ResultItem* liox_nextResultItem(   LIOXHandle     lh );
int          liox_rewindResults(   LIOXHandle     lh );
```

The `liox_nextResultItem()` function returns a pointer to the next `ResultItem` and updates the `result pointer` for the `LIOX`. When no more result items exist the functions returns `NULL`. The `liox_rewindResults()` resets the `results pointer` to the beginning of the result list.

A user can iterate over a result list as follows:

```
ResultItem *pItem = NULL;
LIOXHandle lh;

while ( (pItem = liox_nextResultItem( lh))) {

    processItem( pItem);
}
```

In the above example the `while` loop terminates when `liox_nextResultItem` returns `NULL`.

2.1.4.2. 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.

2.1.4.2.1. Common Result Item Functions

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

Both result types have a `timestamp` field (24 bits), an `error` field (16 bits) and a `result type` bit. The following functions retrieve these fields from a `ResultItem` pointer.

```
unsigned int liox_riGetTimestamp(   ResultItem     *pItem );
unsigned int     liox_riGetError(   ResultItem     *pItem );
int             liox_riHasPayload(   ResultItem     *pItem );
```

The `liox_riHasPayload()` function test the `result type` bit and returns non-zero if that bit is set. This indicates that the result item is a response to a read command and has a payload.

2.1.4.2.2. 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:

```
unsigned short   liox_riGetHeader(   ResultItem     *pItem );
unsigned short* liox_riGetPayload(   ResultItem     *pItem );
```

In addition to the basic `liox_riGetPayload()` function the following helper functions might also be added for specific payload types.

```
unsigned short         liox_riGetPayload16(   ResultItem     *pItem );
unsigned int           liox_riGetPayload32(   ResultItem     *pItem );
unsigned long long liox_riGetPayloadTKR64(   ResultItem     *pItem );
```

The `liox_riGetPayload16()` and `liox_riGetPayload32()` return the first 16 bits and 32 bits of the payload respectively. Most register reads fall into these two cases.

The `liox_riGetPayloadTKR64()` function is a special decoder for tracker register reads (`GTRC` and `GTFE` registers are 64 bits wide). The raw data from a tracker register read contains extra protocol bits every 16 bits – this function removes the protocol bits and returns only the 64 bit register value.

Other special decoders might be needed to facilitate decoding of special registers. The environmental registers of the `GTIC` is one example.

Example 2-2. Reading a Calorimeter DAC – Continued

Continuing the example started in Example 2-1 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 `liox_qGCFEread()` command.

```
int resultProcCallback( LIOXHandle lh) {

  ResultItem          *pItem = NULL;
  unsigned int        timeStamp;
  unsigned int        errorValue;
  unsigned short int  registerValue;

  /* fetch the first result item from the LIOX handle */
  pItem = liox_nextResultItem( lh);

  if ( pItem != NULL) {

    /* get the time stamp value */
    timeStamp = liox_riGetTimestamp( pItem);

    /* get the error value */
    error = liox_riGetError( pItem);

    /* test if this item has a result payload */
    if ( liox_riHasPayload( pItem)) {

      /* by design this item is a 16 bit GCFE register */
      registerValue = liox_riGetPayload16( pItem);

      /* do something useful with register value */

    }
  }

  return OK;

}
```

2.2. Synchronous Command and Response Interface (SCRI)

The SCRI 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 could provide backward compatibility with the GNAT interface that

the I&T test stands are currently using.

Architecturely the synchronous interface would be built on top of the asynchronous interface, hiding the details of the asynchronous interface from the user.

The `SCRI` interface is much simpler than the `ACRI`. With this 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 lists and properly initialize a `LIOX` structure.

The synchronous versions of the "LAT Command Functions" have similar function names and signatures as the asynchronous versions. The synchronous function name uses the letter `s` where the asynchronous function name has the letter `q`.

Below is the function prototype for the synchronous version of the `GCFE` read register command discussed previously in Example 2-1. This function reads a register on a particular `GCFE`.

```
int   liox_sGCFEread(   LIOXHandle          lh,
                        short            temId,
                        short           gcccId,
                        short           gcrcId,
                        short           gcfeId,
                        short            regId,
                        short*           value );
```

Example 2-3. Read Calorimeter DAC – Synchronous Interface

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 */
errVal = liox_sGCFEread( lh, temId, gcccId, gcrcId,
                         gcfeId, DACregId, &regVal);

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

While the `SCRI` is simpler to use it suffers from the *severe* limitions of being synchronous and limited to only a single command.

Chapter 3. Event Processing

This chapter decsribes the LAT Event Processing Interface (LEPI), a software interface within the LICS for processing event data. The LEPI follows the LCB event processing model fully described in [1].

The LEPI relieves the user from knowing the details of how to program the LCB for event processing. The user does, however, need a good understanding of the LCB event processing model, which is briefly reviewed in the next section.

3.1. LCB Event Processing Model

For event data the LCB uses a circular memory buffer that physically resides in the SBC. The starting address of the circular buffer is stored in the EVENT_BASEn register of the LCB. The LCB maintains the buffer's write pointer internally, while the user maintains the buffer's read pointer in the EVENT_FREEn register of the LCB.

Paraphrasing from [1], the life cycle of an event consists of the following steps:

- The LCB decodes an incoming event packet and buffers it in the export unit of the LCB
- The LCB allocates memory for the incoming event from the circular buffer.
- The LCB DMAs the event data into this memory.
- An unsolicated result descriptor, referencing the memory location for the event, is written into the results FIFO for the appropriate SBC.
- The LCB raises an interrupt.
- The LEPI traps the interrupt and reads the result descriptor from the results FIFO.
- The LEPI checks the result descriptor for DMA errors – on errors the LEPI calls a user supplied error handling call-back function.
- The LEPI determines from the result descriptor that the result is an *event* result.
- From the result descriptor the event data is located and processed.
- When event processing is completed the user deallocates the event by updating the circular buffer's read pointer.

3.2. Hardware Initialization

When the LCB initializes it needs the starting address of the event circular buffer. As discussed in [8] the Director SBC of the PCI crate is responsible for programming the EVENT_BASEn registers of the LCB on behalf of all SBCs in the PCI crate.

The power on reset value of the EVENT_FREEn registers will be 0x0, which is not correct for the circular buffer. The Director SBC will write a "good" starting value into the EVENT_FREEn registers, but that value is TBD.

At this point the hardware is ready to take event data. The `LEPI`, however, requires more configuration before processing event data.

3.3. LEPI Initialization

In order for the `LEPI` to dispatch event data and DMA error conditions to a user, the `LEPI` needs a mechanism for notifying the user. The `LEPI` will utilize user supplied call-back functions for error handling and event processing.

3.3.1. Error Handler

Every result has the potential for DMA errors, which is determined by inspecting the result descriptor. During initialization the user provides an error handling call-back function. If an error is detected the `LEPI` calls the user supplied error handler, passing the error code as an argument to the handler.

The `typedef` for an error handling call-back function appears below:

```
typedef int (*LEPI_ERROR_HANDLER_FUNC)( unsigned int errCode);
```

The function prototype of the error handling call-back appears below:

```
int lepi_SetErrorHandler(   LEPI_ERROR_HANDLER_FUNC     errHandler );
```

An example of registering an error handling call-back function is shown below:

```
/* declare call-back routine */
int myErrorHandler( unsigned int errCode);

/* register call-back */
lepi_SetErrorHandler( myErrorHandler);
```

3.3.2. Event Handler

During initialization of the `LEPI` the user will provide an event processing call-back function. This function will be called by the `LEPI` when it determines that event data is ready for processing. The `LEPI` will pass a pointer to the event data into the call-back function as an argument.

The `typedef` for an event processing call-back function appears below:

```
typedef int (*LEPI_EVENT_PROC_FUNC)( unsigned int *pEvent);
```

The return code from the event processing call-back is used as the "length" to free within the circular buffer. The value of `0` means "do not free anything". See Section 3.4.2 for more about freeing memory from the circular buffer.

The function prototype of the event call-back appears below:

```
int lepi_SetEventProc(   LEPI_EVENT_PROC_FUNC    procFunc );
```

An example of registering an event processing call-back function is shown below:

```
/* declare call-back routine */
int myEventProc( unsigned int *pEvent);

/* register call-back */
lepi_SetEventProc( myEventProc);
```

3.4. Event Processing

Event processing begins when the `LCB` raises the `RESULT_READY` interrupt. At this time the `LEPI` reads a result descriptor from the `RESULTS` FIFO.

From the result descriptor the `LEPI` can determine three important pieces of information:

- Did a DMA error occur ?
- Is the result a solicited command/response result ?
- Is the result unsolicated event data?

Results for solicited command/response is previously discussed in Chapter 2.

The following sections describe error handling and event handling.

3.4.1. Error Handling

If the result descriptor indicates DMA errors the `LEPI` calls a user supplied error handling call-back function, passing in the error code as an argument.

See Section 3.3.1 for more about the error handler.

3.4.2. Event Handler

Event processing occurs within the user’s event processing call-back function. From within this function the user has complete access to the event.

The first 32-bit word of the event result is called the summary word and contains a 16-bit error word and a 16-bit "event length" word. The size of the length word represents the length of the event packet payload plus the summary word.

After processing the event and when the user is *completely* finished with the memory that the event occupies, the `LEPI` needs to deallocate or free that memory in the circular buffer. This is accomplished by "freeing" the length of the event.

To free the event the user has two choices. The first choice is to use the return value from the event processing call-back function as the length of memory to free. Returning a value of `0` means "do not free any memory".

This method allows for a one-to-one correspondence between events and memroy deallocation. Every event is deallocated as it is processed.

The second choice is for the user to pass a length to the `LEPI` `lepi_FreeEvent()` function, whose prototype is shown below. The `lepi_FreeEvent()` function frees the amount of memory indicated by the length parameter.

```
int lepi_FreeEvent(   unsigned int    eventLength );
```

Using the second method allows the user to defer the freeing of the event memory to a later time, *after* the event processing call-back function has returned. E.g. the user could *bundle* several deallocation requests into a single, large deallocation request. This may improve performance over the PCI bus.

3.4.2.1. Example Event Processing Function

This section details an event processing call-back function.

```
...  time passes and myEventProc is called back ...

/* implementation of myEventProc */
int myEventProc( unsigned int *pEvent) {

/* process the event */
/*  the return value of process_event() is the amount of
    circular buffer memory the LEPI will  free. */

return process_event( pEvent);

}
```

In the above example the `process_event()` function controls how much event memory to free. In the simple case process_event() would return the event length and the event would be freed.

A more complex example would be if process_event() returned `0`. In this case process_event() would need to store the event length in memory for later deallocation with the `lepi_FreeEvent()` function.

Appendix A. Result Processing Flow Diagram

A flow chart diagraming the result processing flow.

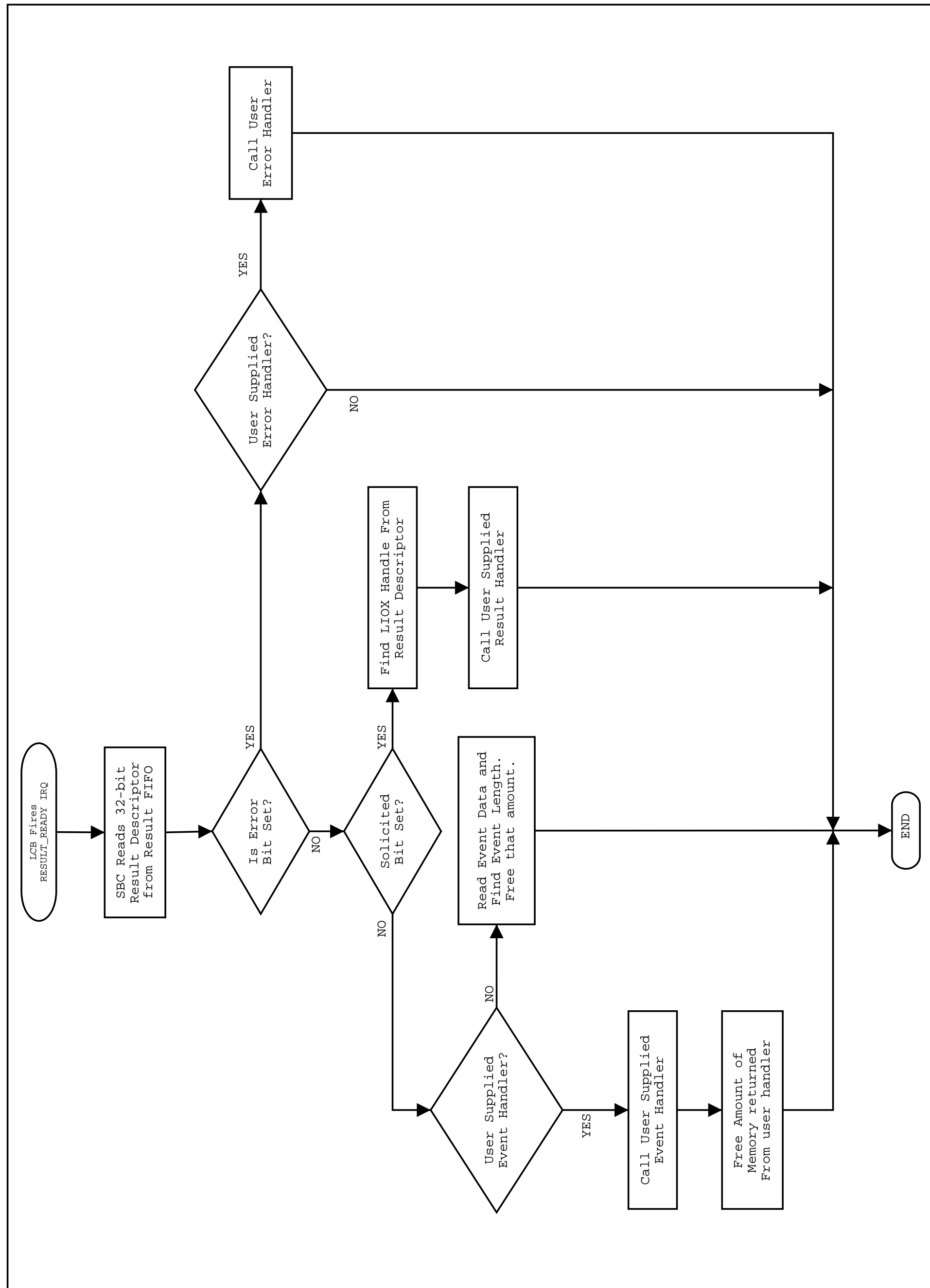

Figure A-1. Result Processing Flow Diagram

Background Documentation

[1] Michael Huffer, *LAT Communication Board: LCB Design Specification*, LAT-DS-00639-D1.

[2] Michael Huffer, *LAT Inter-module Communications: A reference manual*, LAT-DS-00606-D1.

[3] Michael Huffer, *The Tower Electronics Module (TEM): A Primer*, LAT-DS-00605-D1.

[4] Michael Huffer, *The ACD Electronics Module (AEM): A Primer*, LAT-DS-00639-D1.

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

[6] *GLAST LAT Flight PowerPC SBC – To Be Specified*.

[7] *GLAST LAT Spacecraft Interface Board – To Be Specified*.

[8] *LAT System Initialization: A Beginning*.