LAT Flight Software

LAT Charge Injection Calibration Software Design

Number:	V1-0-0
Subsystem:	Data Acquisition/Flight Software
Supersedes:	None
Type:	Document Template
Author:	J. Swain
Created:	February 18, 2005
Updated:	June 29, 2005
Printed:	March 28, 2005

This document presents the design of the LAT Charge Injection Calibration utility (LCI).

Document Approval

Prepared By:	J.Swain	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
0.0	15 February 2005	Draft Release	J. Swain
0.1	22 February 2005	Incorporated comments from JJ and Tony	J. Swain
0.2	25 February 2005	Updated the schedule estimates	J. Swain
0.3	26 February 2005	Modified design after exploring implementation of the LATC calibration mode. Added missing portions of the interface to the cue. Move some of the work from the control function *calibrate* to the *collect* function.	J. Swain
0.4	16 March 2005	Modified the design of the control function following conversation with Tony. Updated the schedule. Modified the XML vocabulary and added INHIBIT keyword for many configuration parameters.	J. Swain
0.5	25 April 2005	Modified the XML vocabulary and added a strobe tag that can be inhibited to send a TAM with TACK only. Added destination to the consignment initiation function – will be helpful for testing. Several revisions of the schedule.	J. Swain
0.6	25 April 2005	Added the requirements from the FSW spec. Added the construction structure. Added the conversation structure.	J. Swain
1.0	June 21, 2005	Extensions to the XML vocabulary required for subsystem calibration scripts managed by I&T. Added support for iterating variables. Split configuration into distinct ACD, CAL and TKR forms, thus a calibration will be one of these three. This has consequences for the catechise and compact portions of the design.	

Contents

0 Commencement

0.0 Scope

This document provides a detailed description of the design of the LAT Charge Injection Calibration utility (LCI). The initial draft shall provide sufficient detail to be used in the construction of a rudimentary completion schedule, and will form the basis of the subsequent implementation process. As the software proceeds towards release the design will be revised and the document updated accordingly. Users, as opposed to developers, of the LCI package should consult the *LCI User Guide*.

0.1 References

1. LAT-TD-00606 *LAT Inter-module Communications: A reference manual*.
2. LAT-TD-01380 *LAT Communication Board Driver: Software architecture and interfaces*.
3. LAT-TD-01547 *The Command/Response Unit: Programming ICD specification*.
4. LAT-TD-01546 *The Event Builder Module: Design specification*.
5. LAT-TD-01545 *The GLT Electronics Module: Programming ICD specification*.
6. LAT-TD-00639 *The ACD Electronics Module (AEM): A primer*.
7. LAT-SS-00363 *ACD-LAT ICD: Mechanical, thermal and electrical*.
8. LAT-SS-00363 *LAT Dataflow Specification: ACD-AEM interface*.
9. LAT-TF-00605 *The Tower Electronics Module (TEM): Programming ICD specification*.
10. *The GLAST acronym list*.

0.2 Request for Comments

If you have comments or corrections, please email jswain@slac.stanford.edu.

0.3 Acronyms

LAT – Large Area Telescope

SIU – Spacecraft Interface Unit

PDU – Power Distribution Unit

EPU – Event Processing Unit

TEM – Tower Electronics Module

CRU – Command Response Unit

ACD – Anti-Coincidence Detector

DAQ – Data AcQuisition

DAB – DAQ Board

LATp – LAT (communications) protocol

EEPROM – Electronic Erasable Programmable Read Only Memory

LCB – LAT Communications Board

GEM – Global trigger Electronics Module

EBM – Event Builder Module

AEM – ACD Electronics Module

FREE – FRont End Electronics

P/R – Primary/Redundant

C/R – Command/Response

0.4 FSW Specifications

The LAT Charge Injection Calibration software is designed to satisfy the following requirements from the FSW specification.

0.4.0 5.3.12 Charge Injection Calibration

This section describes charge injection functions, initiated upon command, which acquire data to support instrument calibration. The gathered data are sent to the solid state recorder on the spacecraft for downlink to the ground and contains the information required by subsystem engineers and ground software to perform calibration activities. Instrument configuration data (e.g., association of DAC value to event data) are included per paragraph 5.3.17.4. [9] provides details of commands and [30] details of configuration files that drive the charge injection functions.

Note that other functions that support instrument calibration, but do not involve charge injection, may appear in the Event Monitoring or Diagnostics sections.

0.4.0.0 5.3.12.0 TKR TOT

The analysis of the following data is performed on the ground to obtain TOT gain calibration. TOT gain will be used to determine the absolute scale of the calibration DAC.

0.4.0.0.1 5.3.12.0.1 TOT Measurements

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall collect such data as is necessary to calculate the average TOT values for each channel for a given DAC setting.

0.4.0.0.2 5.3.12.0.2 TOT Parameters Stored in File

[1] (3.1.3.2.2), [4] (5.4.14)

The number of points, their calibration DAC values, and the number of samples for each data point shall be given by a file or files that can be updated via a command from the ground. See [30] for details.

0.4.0.0.2.1 5.3.12.0.2.1 TOT Number of Points Minimum

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall permit gathering data for 4 points or less.

0.4.0.0.2.2 5.3.12.0.2.2 TOT Number of Points Maximum

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit gathering data for at least 12 points.

0.4.0.0.2.3 5.3.12.0.2.3 TOT Number of Samples

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit at least 10 samples to be gathered at each data point.

0.4.0.1 5.3.12.1 TKR Threshold and Charge Scans

The analysis of the following data is performed on the ground to determine the best threshold for each GTFE chip. This analysis also provides a noise value for each channel.

0.4.0.1.1 5.3.12.1.1 Threshold Scan Measurements

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall collect such data as is necessary to determine the data capture efficiency for each channel for several threshold points at a fixed calibration DAC value.

Note that one channel per GTFE can be pulsed at the same time (24 channels per layer).

0.4.0.1.2 5.3.12.1.2 Threshold Scan Parameters Stored in File

[1] (3.1.3.2.2), [4] (5.4.14)

The number of threshold points and their threshold DAC values, the number of samples for each data point, and the calibration DAC value for each GTFE shall be given by a file or files that can be updated via a command from the ground.

0.4.0.1.3 5.3.12.1.3 Deleted

0.4.0.1.4 5.3.12.1.4 Charge Scan Measurements

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall collect such data as is necessary to determine the data capture efficiency for each channel for several calibration charge points at a nominal DAC value.

Note that one channel per GTFE can be pulsed at the same time (24 channels per layer).

0.4.0.1.5 5.3.12.1.5 Charge Scan Parameters Stored in File

[1] (3.1.3.2.2), [4] (5.4.14)

The number of threshold points and their threshold DAC values, the number of samples for each data point, and the threshold DAC value for each GTFE shall be given by a file or files that can be updated via a command from the ground.

0.4.0.1.6 5.3.12.1.6 TKR Scan Number of Points Minimum

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall permit gathering data for 5 points or less.

0.4.0.1.7 5.3.12.1.7 TKR Scan Number of Points Maximum

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit gathering data for at least 25 points.

0.4.0.1.8 5.3.12.1.8 TKR Scan Number of Samples

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit at least 50 samples to be gathered at each data point.

0.4.0.2 5.3.12.2 TKR Trigger Check

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall collect such data as is necessary to determine the per channel trigger efficiency for large injection charge (DAC 63).

0.4.0.3 5.3.12.3 ACD Charge Injection Calibration Mode

0.4.0.3.1 5.3.12.3.1 Deleted

0.4.0.3.2 5.3.12.3.2 ACD Charge Injection

[1] (3.1.3.2.2), [4] (5.4.14)

In Charge Injection Calibration Mode, the FSW shall permit initiation of at least 100 triggers for each of 64 steps in each of the two ranges (low and high) for each of the 194 front end ASICs (GAFEs).

Note: This mode will normally be coupled with a set of pedestal measurements. For testing purposes, initial configuration values may be found in the ACD Charge Injection Calibration test script delivered to Integration & Test.

0.4.0.3.3 5.3.12.3.3 ACD Charge Injection Calibration Mode Data

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall collect and deliver the raw PHA values.

0.4.0.4 5.3.12.4 CAL Charge Injection Calibration Mode

0.4.0.4.1 5.3.12.4.1 Deleted

0.4.0.4.2 5.3.12.4.2 CAL Charge Injection

0.4.0.4.2.1 5.3.12.4.2.1 CAL Charge Injection Triggers

[1] (3.1.3.2.2), [4] (5.4.14)

In Charge Injection Calibration Mode, the FSW shall initiate a programmable number of triggers for each of a programmable number of amplitude steps, over a programmable time period.

0.4.0.4.2.2 5.3.12.4.2.2 CAL Charge Injection Charge

[1] (3.1.3.2.2), [4] (5.4.14)

Charge shall be injected into a programmable set of GCFE chips simultaneously.

0.4.0.4.2.3 5.3.12.4.2.3 CAL Charge Injection Programmable Triggers

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit the collection of at least 1000 triggers per injection DAC value.

0.4.0.4.2.4 5.3.12.4.2.4 Deleted

0.4.0.4.2.5 5.3.12.4.2.5 CAL Charge Injection Programmable Amplitude Steps

[1] (3.1.3.2.2), [4] (5.4.14)

FSW shall permit the collection of data with at least 128 different injection DAC values.

0.4.0.4.2.6 5.3.12.4.2.6 CAL Charge Injection Settling Time

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall provide the capability to ensure a delay of at least 0.5 seconds between step changes greater than or equal to 0.5 V.

0.4.0.4.2.7 5.3.12.4.2.7 CAL Charge Injection Charge Range

[1] (3.1.3.2.2), [4] (5.4.14)

The FSW shall allow programmable DAC values from 0 to 4095, with the minimum programmable separation between successive DAC values no greater than 2.

0.4.0.4.3 5.3.12.4.3 CAL Charge Injection Calibration Mode Data

0.4.0.4.3.1 5.3.12.4.3.1 Deleted

0.4.0.4.3.2 5.3.12.4.3.2 Deleted

0.4.0.4.3.3 5.3.12.4.3.3 Deleted

0.4.0.4.3.4 5.3.12.4.3.4 Deleted

0.4.0.4.3.5 5.3.12.4.3.5 CAL Charge Injection Data – Log-Accept Occupancy Rates

[1] (3.1.3.2.2), [4] (5.4.14)

The telemetered data shall include enough information so that ground software is able to determine rates of CAL log-accept occupancy, including by log-end for a specified layer over the programmed time period.

0.4.0.4.3.6 5.3.12.4.3.6 CAL Charge Injection Data – Trigger Request Rates

[1] (3.1.3.2.2), [4] (5.4.14)

The telemetered data shall include enough information so that ground software is able to determine rates of CAL FLE and FHE trigger primitives, including by log-end for a specified layer over the programmed time period.

1 Conspectus

The LAT electronics include circuitry to inject a known amount of charge into the front-ends for the purposes of diagnostics and calibration. The LAT Charge Injection Calibration software uses this circuitry to produce a dataset that contains information required to calibrate the front-end electronics. FSW does *not* provide tools to analyze such data.

The complete calibration consists of an initial LAT configuration, performed using LATC, followed by repeated cycles of configuration, collection, construction, compaction and consignment. The LCI configuration affects only a small subset of the LAT registers in the EBM, GEM and threshold registers scattered around the DAQ system and can be inhibited, to leave the initial LAT configuration in place.

The collection phase of the cycle is concerned with the recovery of events from the circular buffer of the LCB, ensuring that the calibration will not stall as a result of back pressure. Once a complete event has been accumulated some preprocessing is performed to handle fragmentation, locate the diagnostics block and unpack the tracker or calorimeter data.

The compaction and consignment phases operate on a complete block of collected events. The data volume is reduced to manageable levels by a combination of selection and loss less compression before begin sent to the SSR. The five phases of the calibration cycle are largely independent, a fact that will be reflected in the final design.

1.0 Single CPU, Single task

The collection and construction routines should be written to be as lightweight as possible, permitting them to be run within the context of the LCB event callback. All substantial processing must be done within the context of the single LCI task. Furthermore, it is intended to perform all charge injection calibration work on a single CPU, the LAT SIU. The absence of any inter-CPU synchronization will considerably simplify the resulting implementation.

As implementation and testing proceed it may become apparent that this approach is not feasible and the work will have to be split across multiple tasks or multiple CPUs. In an effort to mitigate this risk all reasonable efforts will be made to ensure that the initial implementation can be modified to run in multiple tasks by extension, rather than complete refactoring.

2 Control

A single master task residing on the SIU will perform the LAT Charge Injection Calibration. Therefore, the majority of the interaction between LCI and other portions of flight software will be mediated by ITC message queues. Resource allocation and deallocation will be performed by the *initialise*[1] and *teardown* functions, respectively. Two more functions, *start* and *stop* will be provided to start and stop the task. The *stop* and *teardown* function may be required for testing purposes but should not be necessary for on-orbit operations. Once started the LCI task will respond to the commands *calibrate,* and *abort*. To allow LCI to respond to abort commands in a timely manner, the calibration is divided up into *cycle*s, each of which is initiated by an ITC message generated at the end of the previous cycle and queued back to the LCI task.

2.0 Initialise

This function performs the initial resource allocation for the LCI master task, creates and initialises the ITC message queues, and puts the task into the state INITIALIZED. It does not start the LCI task.

2.0.0 Arguments

(a) Capacity.

The capacity of the calibration task represents the maximum number of events that can be collected and compacted in one cycle.

The design capacity was 100.

This argument will be eliminated in a future version of LCI, once it has been integrated with the CFG startup facility.

[1] Note that the function titles in this document may or may not match function names in the final implementation.

2.0.1 Operations

(a) Allocate sufficient storage for

  (i) The ITC message queues.

  (ii) The LATC configuration.

  (iii) The LCI configuration.

  (iv) The Command-Response lists.

  (v) The data collection.

  (vi) The data compaction.

(b) Initialize the ITC message queues.

2.1 Start

This function starts the ITC task. The function blocks until the underlying task is in the STARTED state, then sets the LCI state to WAITING.

2.1.0 Arguments

None.

2.1.1 Operations

(a) Check that the current LCI state is INITIALISED.

(b) Start the ITC task.

  (i) Use the blocking call to ensure that the ITC task is started when this function returns.

(c) Set the LCI state to WAITING.

2.2 Stop

Although not required for on-orbit operations, it may be useful during testing to stop the LCI task.

2.2.0 Arguments

None

2.2.1 Operations

(a) Check that the current LCI state is WAITING.

(b) Stop the ITC task.

(c) Set the LCI state to INITIALISED.

2.3 Teardown

Although not required for on-orbit operations, it may be useful during testing to free the resources acquired by LCI during the initialisation.

2.3.0 Arguments

None

2.3.1 Operations

(a) Check that the current LCI state is INITIALISED.

(b) Teardown the ITC message queues.

(c) Free all the memory allocated for LCI.

2.4 Calibrate

A calibration is initiated in response to the calibrate command.

2.4.0 Arguments

(a) LATC configuration master file ID used to configure the LAT.

(b) LCI configuration ID.

(c) Destination, either File ID or LATp address, of the calibration data

2.4.1 Operations

(a) Check that the current LCI state is WAITING.

(b) Set the LCI state to CALIBRATING.

(c) Store the node, task and queue IDs for sending the “finish” message.

(d) Register the LCI event callback with the LCB on protocol 1.

(e) Load the LATC configuration from the file system.

(f) Open the LCI configuration.

   (i) Make the configuration as requiring a reload of the LATC configuration to force the initial configuration.

   (ii) Set the number of outstanding events to be the number of requested events in the first configuration.

   (iii) Configure the LAT.

(g) Initialise the output consignment.

(h) Queue an item to the LCI bulk queue – this will invoke the Cycle function

2.5 Cycle

This function is initiated when something appears on the LCI bulk queue. If the ABORT flag has been set then the function will return LCI_ABORTED. If there are no more configurations to read, then the function will perform post calibration clean up and return LCI_SUCCESS. Otherwise the function will perform one configure, collect, compact and consign cycle of the calibration and then queue an item to the LCI bulk queue to initiate the next cycle.

Since the function is registered as a callback with ITC, its signature is already defined.

```
    unsigned int LCI_cycle(void*                         prm,
                           const struct _ITC_QueueItem* qitem,
                           void*                         pay,
                           unsigned int                  len)
```

2.5.0 Operations

(a) Check the abort flag

(b) If the number of outstanding events is greater than the capacity of LCI

   (i) Set the number of events to collect on this cycle to the capacity of LCI

   (ii) Decrease the number of outstanding events by the capacity of LCI

(c) Otherwise

   (i) Set the number of events to collect on this cycle to the number of outstanding events

   (ii) Set the number of outstanding events to zero.

(d) Collect, compact and consign the requested events.

(e) If there are no more outstanding events

  (i) Read a configuration from the LCI configuration system

  (ii) Configure the LAT.

  (iii) Set the number of outstanding events to be the number of events requested for this configuration.

(f) Collect the requested number of event.

(g) Compact the block of events and consign to the SSR.

(h) If there are any errors (including abort or configuration end-of-file)

  (i) Restore the LCB event callback.

  (ii) Close the configuration.

  (iii) Complete the consignment.

  (iv) Send a message to the initiating task indicating calibration ended and status.

2.6 Abort

The amount of time that the LAT is occupied with calibration is variable, depending on the number of cycles requested in the LCI configuration file and the number of events per cycle. In any case, it will be significant, which means than changing circumstances could make it be desirable to terminate the calibration early. The abort command contains no additional information and triggers no action when sent to the LCI task if is not currently acquiring data. At the beginning of each calibration cycle the command queue is checked for the presence of an abort command. If the abort command is found then the calibration terminates immediately.

2.6.0 Arguments

None.

2.6.1 Operations

(a) Check that the current LCI state is CALIBRATING.

(b) Set the current LCI state to ABORTING.

3 Configuration

One of the arguments passed to the start function will identify the file used to configure LCI. The LCI package will provide an XML parser to compile raw XML configuration files into a binary format for upload to the LAT SIU. The XML vocabulary is described in detail in the Compilation section of this document. The binary file will used to determine the contents of a small subset of LAT configuration registers.

One configuration block of the configuration file may result in many calibration cycles. The configuration block will specify

- The type of calibration being performed (ACD, CAL or TKR).
- A pointer to the common multi-item command-response list used by LCI to interact with the LAT.
- Number of pulses.
- Pulse period.
- Input DAC settings.
- Settling delay.
- Thresholds.
- Calorimeter ADC ranges to be read.
- Calorimeter column mask.
- Tracker channel mask.
- Trigger sequence – TACK delay.

Many of these quantities can be iterated over, so there will actually be five numbers stored for each; the initial value, the number of steps, the size of the steps, the current step and the current value. The heart of the LCI configuration will be a single structure containing all of this information.

In addition to the configuration data, the structure will also contain a pointer to the configuration source. The in-flight configuration operations will be *open*, *read*, *configure*, and *close*. The LCI XML parser will create a binary configuration file so will require the operations *create*, *write* and *finish*[2]. Some pieces of configuration information will be required by other sections of LCI such as the number of pulses, passed into the call to *collect* by the control code, the calibration type, required by the *compact* function, and a mask of the calorimeter ranges to be compacted and consigned. Accessor functions such as *triggers, type* and *calRange* will be provided to expose this information whilst hiding the configuration structure.

3.0 Open

The flight version of this function will open the LCI configuration file and read in the first configuration block, but alternative test and development implementations may perform other operations, such as resource allocation and initialization. In general, the purpose of this function is to prepare the configuration source for reading.

3.0.0 Arguments

(a) Pointer to the configuration structure

(b) File ID

3.0.1 Operations

(a) Open the configuration file.

(b) Read in the file version and type information.

(c) Read in the GEM portion of the configuration blocking.

(d) Read in the ACD, CAL or TKR portion of the configuration block.

3.1 Read

Each call to read will refresh the configuration structure by incrementing one of the configuration quantities. When all quantities have run over the complete range, the next configuration block will be read from file. When there are no more configurations available the function will return (but not report) end of file. Any other errors that occur will be reported normally.

3.1.0 Arguments

(a) Pointer to the configuration structure

[2] These functions will not be required on-orbit so will be provided by a separate constituent.

3.1.1 Operations

(a) Consider the first variable.

(b) If there are no more steps in the loop over this variable

  (i) Reset the variable to its initial condition.

  (ii) Move onto the next variable.

(c) Otherwise

  (i) Increment this variable's counter.

(d) If all of the variables have reached their terminal condition.

  (i) Refresh the configuration block from the source file.

  (ii) If any of the variables in the configuration block are marked USE_LATC

    1. Set the reload flag.

(e) If there are no more configuration blocks in the source file.

  (i) Return END_OF_FILE.

3.2 Configure

The information from the configuration structure will be translated into a series of commands to manipulate a subset of the LAT registers.

3.2.0 Arguments

(a) Pointer to the configuration structure.

(b) Pointer to the common multi-item command-response list.

(c) Pointer to the LATC structure.

3.2.1 Operations

(a) If the reload flag is set

  (i) Load the LATC configuration.

  (ii) Load the GEM registers.

(b) Configure the ACD, CAL or TKR.

(c) Delay to allow the injection DACs to settle.

3.3 Type

This function just returns the current configuration type.

3.3.0 Arguments

(a) Pointer to the configuration structure.

3.3.1 Operations

(a) Return the configuration type.

3.4 Triggers

This function just returns the number of periodic triggers to be requested by the collection code.

3.4.0 Arguments

(b) Pointer to the configuration structure.

3.4.1 Operations

(b) Return the number of triggers indicated by the configuration

3.5 CalRange

This function just returns the mask of calorimeter ranges to be retained..

3.5.0 Arguments

(c) Pointer to the configuration structure.

3.5.1 Operations

(c) Return the range mask.

3.6 Reload

This function just set the reload flag, indicating that the LATC configuration should be reloaded the next time *configure* is called.

3.6.0 Arguments

(d) Pointer to the configuration structure.

3.6.1 Operations

(d) Set the reload flag to TRUE.

3.7 Close

The flight version of LCI will simply closes the open file and clears the configuration structure.

3.7.0 Arguments

(a) Pointer to the configuration structure.

3.7.1 Operations

(a) Close open file

(b) Clear configuration structure.

3.8 Create

The formal version of this function will simply open the LCI configuration file, but alternative test and development implementations may perform other operations, such as resource allocation and initialization. In general, the purpose of this function is to prepare the configuration source for writing.

3.8.0 Arguments

(a) Pointer to the configuration structure.

(b) File ID

(c) Type, one of “ACD”, “CAL” or “TKR”

3.8.1 Operations

(a) Open the configuration file.

(b) Write out the file type and version information.

3.9 Write

Each call to write will transfer the contents of the configuration structure to the output file.

3.9.0 Arguments

(a) Pointer to the configuration structure.

3.9.1 Operations

(a) Append the configuration information to the source.

3.10 Finish

The flight version of LCI will simply close the open file and clear the configuration structure. Other implementations may require other cleanup to be performed.

3.10.0 Arguments

(a) Pointer to the configuration structure.

3.10.1 Operations

(a) Close open file

(b) Clear configuration structure.

4 Collection

Data collection within LCI will be asynchronous with the periodic triggers initiated by LCI software running within the context of the LCI master task and the resulting events being handled by LCI software running within the context of the LCB event task. The event handling is driven by the event fabric design. Events arriving on the LCB are deposited into a finite size ring buffer and may be fragmented into multiple packets, so the event handling consists of copying packets into a buffer and, once a complete event has arrived on the LCB, passing the event onto the next phase of the cycle, construction. Once the packet has been copied, it can be freed. The heart of the data collection system will be a structure containing

- A pointer to a synchronization structure to coordinate collection.
- A pointer to the common multi-item command-response lists.
- A counter of the expected number of events.
- A pointer to the destination of the complete events.
- An event directory, provided by EDS.
- A state information vector, provided by EDS.
- Various pointers for the management of the collection buffer.

The data collection will be coordinated by a single routine, *collect,* that will initiate the periodic triggers and wait for the events to be collected. The function called from the LCB event handling callback. *put*, will copy the packet into the buffer and update the event directory with each new packet until a complete event is collected, when the event will be passed onto the destination.

4.0 Callbacks

4.0.0 Event callback

Since the function is registered as a callback its signature is already defined by the LCB package.

```
unsigned int LCI_evt_cb(void* prm, unsigned int dsc, LCBD_evt* pkt)
```

The opaque pointer passed as the first argument to this function is a user-defined parameter. It will point to the collector structure.

4.0.0.0 Operations

(a) Call the put routine.

4.0.1 Event verify

It is possible that one or more events may be lost during the collection phase of a calibration cycle. Although undesirable, such an error is not necessarily a fatal error. In such cases LCI will attempt to send a single solicited trigger verify that the event path is not physically broken. A special event callback will be installed that uses a synchronisation object to signal the reception of an event. Again the function signature is already defined by the LCB package.

```
unsigned int LCI_evt_verify(void* prm, unsigned int dsc, LCBD_evt* pkt)
```

The opaque pointer passed as the first argument to this function is a user-defined parameter. It will point to the synchronization object.

4.0.1.0 Operations

(a) Signal the reception of an event.

4.1 Collect

The process of requesting a set of triggers and then waiting for them to be collected is bound together in a single function.

4.1.0 Arguments

(a) Pointer to the collector structure.

(b) The number of expected triggers.

4.1.1 Operations

(a) Initiate periodic triggers.

(b) Wait for the events to be collected.

(c) If the some of the events fail to appear in a timely manner

  (i) Replace the event callback with the verify callback.

  (ii) Initiate a solicited trigger.

  (iii) Wait for the event to arrive.

  (iv) If this event fails to arrive in a timely manner

    1. Report a fatal error.

4.2 Put

This function accepts packets from the LCB and pushes them into the destination, updating any necessary state information to assist with the reformation of truncated events.

4.2.0 Arguments

(a) Pointer to the collector structure.

(b) Event descriptor word.

(c) Pointer to the start of the packet.

4.2.1 Operations

(a) Check that events are expected.

(b) Copy the packet into the buffer.

(c) Update the state information vector

(d) If this packet is the last of an event

  (i) Pass the event onto the destination.

  (ii) Decrement the counter of expected events.

  (iii) If this is the last event to collect.

1. Signal that to the LCI task.

5 Cue

During calibration the control loop must be synchronized with the collection callback function. The control loop must wait until all the expected events have been collected. Rather than imposing a synchronization scheme at design time it will suffice to say that LCI will include a *cue* structure that is passed into two functions, *wait* and *signal*. For error handling purposes there will be a *cancel* function.

An initial implementation of this system will probably use a simple semaphore but careful construction of the interface should allow this to be changed to a message queue with few modifications, should the need arise.

5.0 Wait

This function blocks until another task calls *signal* or *cancel* or until a timeout expires.

5.0.0 Arguments

(a) Pointer to the cue structure.

(a) Timeout value.

5.0.1 Operations

(a) Block until the other side of the synchronization mechanism is triggered (*signal* or *cancel*) or until a timeout expires.

(b) Return the condition that terminated the wait.

5.1 Signal

This function releases a task blocked in a call to *wait* and sets the status to SIGNALED.

5.1.0 Arguments

(a) Pointer to the cue structure

5.1.1 Operations

(a) Set the status to SIGNALLED.

(b) Release a task blocked in a call to wait.

5.2 Cancel

This function releases a task blocked in a call to wait and sets the status to CANCELLED.

5.2.0 Arguments

(a) Pointer to the cue structure.

5.2.1 Operations

(a) Set the status to CANCELLED.

(b) Release a task blocked in a call to wait.

6 Construction

The raw data will not be particularly useful to the down stream phases of LCI, so the Event Data Service (EDS) routines will be used to unpack the calorimeter, tracker and ACD data. This, along with the raw diagnostic data, will then be stored until all events in a cycle have been collected and the compaction phase can start. Unlike the raw data, the unpacked data has a fixed size so can be stored as a simple array, with a pair of indices tracking progress writing data into and reading data from array. The structure will contain

- The size of the array.
- An array of events.
- A read index.
- A write index.

Events will be unpacked and added to the storage area by the *construct* function. They will be subsequently recovered by the *get* function and a *restart* function will be provided to allow the events to be parsed multiple times.

6.0 Construct

This function takes a pointer to the directory of a reassembled event and unpacks the data into the storage area.

6.0.0 Arguments

(a) Pointer to the construction structure.

(b) Pointer to the event directory.

6.0.1 Operations

(a) Check that the storage area is not full.

(b) Unpack each of the tracker, calorimeter and ACD data blocks into the storage area.

(c) Copy the diagnostic data into the storage area.

6.1 Get

Hands back a pointer to the next event in the set.

6.1.0 Arguments

(a) Pointer to the construction structure.

6.1.1 Operations

(a) Check that there are more events in the storage.

(b) Return the next event and increment the next event index.

6.2 Restart

Sets the current event pointer back to the first event in the collection.

6.2.0 Arguments

(a) Pointer to the collector structure

6.2.1 Operations

(a) Check that the mode is READ.

(b) Set the current pointer back to the first event in the collection.

7 Compaction

The data reduction stage falls into two parts, selection and compression. During the selection stage unnecessary information is rejected, and duplicate information summarized at the start of the compacted output. For example, each event contains a seventeen-bit sequence number that should be one more than the previous event. Given that the output data contains the first sequence number, the sequence numbers of all the other events are known.

Once all the unnecessary data has been removed whatever is left must be stored in a loss less manner. The Charge Injection Calibration utility will make use of the arithmetic probability encoder provided by the ZLIB package to compress the data. Detailed descriptions of this compression technique may be found elsewhere.

Given that the data copy and event reformation performed inside the collection callback can be accomplished in significantly less time than the inter-pulse period it might be possible to transfer the work done in during selection and/or the first compression pass to that callback[3]. To preserve logical modularity during the development of LCI the initial implementation will not do this, but the compaction function will be segmented in such a way as to allow such a modification to be made by moving a couple of function calls from one function to another.

The compaction process is sufficiently complex that some of the work is delegated to two wholly owned subsidiaries, *catechise*, and *compress*.

The compaction structure will hold context used during the compaction process. In particular it will contain

- Pointers to the compression structure for each of the data sets.
- The arithmetic probability encoder context.

A single function, compact, will then compact the complete set of events.

[3] Alternatively, the compaction could be split off into a separate task. The initial implementation will avoid the additional complexities of this approach, but all reasonable efforts should be made to allow such a change to be made by extension rather than a complete refactoring of the code.

7.0 Catechise

The compaction method chosen requires the interesting data in each event be examined twice, with a different operation performed each time. It seems sensible to separate the operation of examining the event from the operation that acts upon the data.

The *catechise* routine will skip through the unpacked calorimeter, tracker and ACD data and the raw diagnostic data. At each level of the hierarchy a user defined function may be invoked, if such a function has been defined. The catechist structure will hold pointers to the user defined functions, each accompanied by an opaque pointer that will be passed as the first argument to the functions.

7.1 Compress

The arithmetic probability encoder required a table of probabilities against which to encode the data. Each possible value that a unit of the data might have has a corresponding probability. For maximum possible compression the frequency table can be built from the actual data being compressed, leading to the two-pass approach pursued here. The block of events is subdivided into data set for compression and each data set is histogramed. This histogram is then used to generate the various lookup and frequency tables used by the encoder. Since this process is general for all data set, the histogram and derived tables will be bound together in a structure that holds

- Number of different symbols that the elements of the data set can have, a function of the number of bits in the data and equal to the number of bins of the histogram.
- Histogram of the number of times each possible symbol occurs in the data set.
- Count of the non-zero bins in the histogram, equivalent to the number of distinct symbols in the data set.
- Lookup table mapping the symbols to the integers from zero to the count.
- Frequency table, essentially a zero suppressed version of the histogram giving the frequency of the mapped symbols rather than the original symbols.
- Normalised probability distribution of the frequency table.

Several operations will be defined for this structure. Some of this operations will be accessors, used to maintain code modularity. Others will be more substantial.

7.1.0 Histogram

To avoid exposing the internal details of the compression structure to other sections of LCI, this operation take a symbol and increment the appropriate bin of the histogram.

7.1.0.0 Arguments

(a) A pointer to the compression structure.

(b) Value to add to the histogram.

7.1.0.1 Operations

(a) Check that the value does not exceed the range of the histogram.

(b) Increment the appropriate histogram bin.

7.1.1 Process

After the histogram has been filled, but before the encoding can being, the derived tables and quantities must be calculated.   This function performs that step.

7.1.1.0 Arguments

(a) A pointer to the compression structure.

7.1.1.1 Operations

(a) Count the number of non-zero histogram bins and verify that there is at least one.

(b) Populate the lookup and frequency tables.

(c) Calculate the normalised probabilities.

7.1.2 Pack

The tables used to encode the data will be required to decode the data later. This function efficiently packs the lookup and frequency tables and adds them to the consignment.

7.1.2.0 Arguments

(a) A pointer to the compression structure.

(b) A pointer to the consignment.

7.1.2.1 Operations

(a) Pack the lookup table and frequency table data.

(b) Add to the consignment.

7.1.3 Encode

Like the histogram function, the encode function exists to maintain code modularity.

7.1.3.0 Arguments

(a) A pointer to the compression structure.

(b) A pointer to the encoder context.

(c) The symbol to encode.

7.1.3.1 Operations

(a) Lookup the symbol.

(b) Call the arithmetic probability encoder passing the result of the lookup, a pointer to the table of normalised probabilities and the encoder context.

7.1.4 Unpack

This function recovers the lookup and frequency tables from the consignment and unpacks them.

7.1.4.0 Arguments

(a) A pointer to the compression structure.

(b) A pointer to the consignment.

7.1.4.1 Operations

(a) Get the lookup table and frequency table data from the consignment.

(b) Unpack the data.

7.1.5 Decode

Decode is the compliment of encode.

7.1.5.0 Arguments

(a) A pointer to the compression structure.

(b) A pointer to the encoder context.

7.1.5.1 Operations

(a) Call the arithmetic probability decoder to get the next symbol.

(b) Perform a reverse lookup.

7.2 Compact

A single function call will compact an entire block of events.

7.2.0 Arguments

(a) A pointer to the compaction structure.

(b) A pointer to the source construction.

(c) A pointer to the destination consignment.

(d) The data type to select and compress, one of ACD, CAL or TKR.

7.2.1 Operations

(a) Analyse (histogram) the data sets to be compressed.

(b) Add the lookup and frequency tables to the consignment.

(c) Register the consignment and a suitable output function with the arithmetic probability encoder.

(d) Encode the data.

The encode and analyse steps are achieved by creating an appropriated catechist and then catechising each event.

7.3 Expand

A single function call will recover the compressed data and repopulate the construction.

7.3.0 Arguments

(a) A pointer to the compaction structure.

(b) A pointer to the source consignment.

(c) A pointer to the destination construction.

(d) The data type to expand, one of ACD, CAL or TKR.

7.3.1 Operations

(a) Recover the lookup and frequency tables from the consignment.

(b) Register the consignment and a suitable input function with the arithmetic probability decoder.

(c) Decode the data.

The encode and steps is achieved by creating an appropriated catechist and then catechising unpopulated events until the complete set has been recovered.

8 Consignment

The consignment functions are responsible for the transportation of the compressed events to their final destination. In the final implementation this will be the SSR. A consignment structure will maintain a buffer and such state information as necessary. Functions will be provided to initialize the structure, add words to the consignment and complete the consignment. As with other phases presented earlier, the interface is designed to allow development and test implementations do something different.

8.0 Initiate

Before any actual data is added to the consignment the structure must be initialized.

8.0.0 Arguments

(a) A pointer to the consignment structure.

(b) Destination.

8.0.1 Operations

(a) Reset any state information.

(b) Prepare destination – this might be a simple variable assignment or it might be a file open.

8.1 Add

There are actually several function to handle blocks and single words of different sizes, but they all do essentially the same thing.

8.1.0 Arguments

(a) A pointer to the consignment structure.

(b) Pointer to the start of the array of words to add and length of array.

OR

Single word.

8.1.1 Operations

(a) Copy the data into a buffer.

(b) Update any state variables.

8.2 Complete

After all the data has been added to a consignment it is completed. This action will flush any buffers and may, depending on the implementation, free resources.

8.2.0 Arguments

(a) A pointer to the consignment structure

8.2.1 Operations

(a) Flush any buffers.

9 Conversation

The consignment and compaction phases of the calibration cycle call upon code external to LCI to perform work. In each case it is possible for that external code to return an error for any (or indeed every) symbol being encoded. However, under normal running conditions no errors are expected and those errors that might occur are generally fatal. Rather than trying to propagate such errors from the bottom to the top of the LCI call tree, and consequently littering the code with "if _msg_failure return" blocks an alternative approach has been chosen. The consignment and compaction objects will both be derived from a base class, conversation.

The conversation structure will hold

- The last error generate by one of the functions using this structure, or SUCCESS if there have been no errors.
- A flag indicating whether error messages should be reported or not.

If an error is returned from a function call by one of the compaction or consignment functions then this will passed into the *converse* function that will update the last error and, depending on the current setting of the report flag, call message report.

9.0 Converse

Handle an error message on behalf of the derived object.

9.0.0 Arguments

(a) Pointer to derived object, cast as a pointer to the conversation structure.

(b) Error message code.

9.0.1 Operations

(a) If this message is a failure code

  (i) If messages are being reported

    1. Report the message and store the message code.

  (ii) else

    1. Store the message code.

10 Compilation

The LCI package will provide an executable to compile XML configuration files into binary configuration files for upload to the LAT. The parser, unlike the configuration constituent described earlier, will be stateful. Once a basic configuration has been established subsequent statements will modify this configuration. The parser will take two arguments, a string identifying the subsystem configuration being specified ("ACD", "CAL" or "TKR") and the XML file name.

Many of the quantities specified in the configuration can be iterated over. In this case the XML element should contain either the elements <**initial**>, <**delta**> and <**count**>, or the single element <**constant**>, or a keyword. The keyword LATC is always acceptable, and indicates that LCI should not load any value for this parameter. Note that the LCI XML parser does not perform bounds checking on the parameters entered.

10.0 XML Vocabulary

Hierarchy of XML tags used to specify the LCI configuration.

10.0.0 Common

The following tags are used in all configuration files.

<**configuration**> : Tag encompassing changes to the configuration.

This tag can be empty, which will cause the parser to output another copy of the previous configuration.

<**number**> : Number of events to collect.

<**period**> : Clock ticks between calibration pulses.

DEFAULT gives values of 20000 for a pulse rate of 1kHz.

This is the one value that is subject to bounds checking. If a value less that 20000 is entered then 20000 will be used.

<**delay**> : Delay inserted after the configuration phase of the calibration cycle.

Large changes in DAC value can require a settling period.

<**strobe**> : It can be interesting to set up the LAT for calibration, but not pulse the front-end electronics, i.e. send a TACK only. The OFF keyword specifies that the TAM **should not** have the calstrobe bit set. The ON keyword specifies that the TEM **should** have the calstrobe bit sent.

<**zero_suppress**> :

ON enables zero-suppression.

OFF disables zero-suppression.

10.0.1 Calorimeter

The follow tags are use in calorimeter configuration files.

<**inject**> : *Iterate* : Size of calibration charge. Twelve bit DAC value for the calorimeter.

<**trigger**> : Trigger threshold

<**low**> : *Iterate* : Low energy trigger threshold. Seven bit DAC

The range select bit is just treated as the MSB.

<**high**> : *Iterate* : High energy trigger threshold. Seven bit DAC

<**log_accept**> : *Iterate* : Log accept threshold. Seven bit DAC

<range_uld> : *Iterate* : Range upper level discriminator threshold. Seven bit DAC

<reference> : *Iterate* : Seven bit reference DAC

<**range_mask**> : Four bit mask indicating the calorimeter ranges to compact and consign.

AUTO will cause the calorimeter auto-ranging to be used and a single range returned.

<**first_range**> : Sets the state of the USE_FRST_RNG bit and FRST_RNG bits.

LATC indicates that LCI should not set these bits.

OFF Indicates that the USE_FRST_RNG bit should be cleared.

A number from 0 to 3 indicates the the USE_FRST_RNG bit should be set, and the FRST_RNG bits should be set to the value.

<**tack**> : *Iterate* : Eight bit trigger sequence TACK delay.

<**column**> : *Iterate* : Calorimeter column selection [0, 11].

FOREACH indicates that LCI should iterate over all the columns in the tower.

ALL indicates that LCI should enable all columns of the tower.

10.0.2 Tracker

The following tags are used in tracker configuration files.

<**threshold**> : *Iterate* : Tracker trigger threshold. Seven bit DAC value

<**inject**> : *Iterate* : Size of calibration charge. Seven bit DAC value for the tracker.

<**tack**> : *Iterate* : Eight bit trigger sequence TACK delay.

<**channel**> : *Iterate* : Tracker channel selection.

FOREACH indicates that LCI should iterate over all the channels in the tower with one channel enabled per layer.

ONEPERTFE indicates that LCI should iterate over all the channels on a TFE with one channel enabled per TFE.

10.0.3 ACD

The following tags are used in ACD configuration files.

<**inject**> : *Iterate* : Size of calibration charge. Six bit DAC value.

<**pha**> : *Iterate* : PHA threshold. Sixteen bit DAC value

<**veto**> : *Iterate* : Veto threshold. Six bit DAC value

<**veto_vernier**> : *Iterate* : Six bit DAC value

<**hld**> : *Iterate* : Six bit DAC value

10.1 Decompilation

For completeness, an executable performing the inverse operation will also be supplied. It will read a binary configuration file and produce an XML file that would generate the same binary configuration file.

11 Conclusion

After data collection, compaction and consignment are complete the calibration data will be down linked to the ground and will be processed off line. Before the analysis can be performed the calibration data must be expanded and put into a form acceptable to the off line process. A third LCI executable will be provided to perform this transformation.

The final format of the output data is still to be determined.

12 Consummation

Some details of the implementation.

12.0 Constituents

List of the LCI constituents with some description.

12.0.0 Support

Libraries that provide functionality that will migrate to another package.

12.0.0.0 lci_ape

Stand in for the Arithmetic Probability Encoder portion of the ZLIB package. The LCI version uses a function pointer to flush complete words from the encoder context, whereas the current ZLIB version uses an in-memory buffer.

12.0.1 Flight

Flight libraries for LCI.

12.0.1.0 lci_cnv

The conversation structure and associated functions.

12.0.1.1 lci_csg

The "standard" consignment structure and routines, currently sending output to file, but will eventually write output to the SSR.

12.0.1.2 lci_csg_toMem

An alternative version of the consignment library that uses in-memory storage in place of the file.

12.0.1.3 lci_cmn

The cue, construct, catechise and compress functions are collected together into a single common library. These routines are not expected to have alternative versions, unlike consignment, configuration, collection and compaction, which might.

12.0.1.4 lci_cnf

The functions necessary to read and apply the configuration from a binary file to the LAT.

12.0.1.5 lci_cll

The collection routines.

12.0.1.6 lci_cpc

The compaction routines.

12.0.1.7 lci_cnt

This constituent provides the static LCI handle and the associated ITC task along with the command and control functions for LCI.

12.0.2 Host/Test

Some of the functionality describing in the design is only of use on host systems or during testing. Operations such as the creation of configurations and retrieval of consignments, for example. This functionality is separated off from the flight libraries to avoid loading unnecessary code onto the LAT.

12.0.2.0 lci_rtr

The functionality to retrieve a consignment from file.

12.0.2.1 lci_rtr_fromMem

The functionality to retrieve a consignment from memory.

12.0.2.2 lci_ext

Extensions to the common library, specifically the decompression and reconstruction functions.

12.0.2.3 lci_rcl

The recollection structure provides memory for holding the raw data recovered from a consignment.

12.0.2.4 lci_exp

Functionality to expand a compacted data set.

12.0.2.5 lci_xml

The XML handling functions.

12.0.3 Test

Test specific code is collected together into another set of constituents.

12.0.3.0 lci_tfw_cnt

12.0.3.1 lci_test_cnt

12.0.3.2 lci_test_cnf

12.0.3.3 lci_run