The GLAST experiment at SLAC

DataFlow Public Interface (DFI)

GLAST Electronics group

Users Manual

Document Version:	0.5
Document Issue:	1
Document Edition:	English
Document Status:	not for public release
Document ID:	LAT-TD-07689
Document Date:	November 24, 2005

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

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

Abstract

To do so, the application codes against a set of C++ classes called the *DataFlow Interface*. (DFI) A complete description and explanation of these classes is one of the principal goals of this document.

Intended audience

While, a design proposal, this document is intended principally as a guide for the *users* of the DFI. These include:

- Testers of Flight Software
- Developers of Flight Software
- Developers of I&T (Integration and Test) based systems

Conventions used in this document

Certain special typographical conventions are used in this document. They are documented here for the convenience of the reader:

- Field names are shown in bold and italics (*e.g.*, ***respond*** or ***parity***).
- Acronyms are shown in small caps (*e.g.*, SLAC or TEM).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

References

1 *1553 Product Handbook*, United Technologies Microelectronics Center Inc., 1992.

2 *CCSDS Package User Manual*, LAT Flight Software User Manual.

3 *CTDB 1553 Drivers*, LAT Flight Software User Manual.

4 *EGSE Used to Test GASU*, LAT-TD05787.

5 *Enhanced Summit Family Product Handbook*, UTMC Microelectronics Systems Inc., October 1999.

6 *GLAST 1553 Bus Protocol Interface Control Document*, Spectrum Astro, Inc.

7 *GLAST LAT Instrument – Spacecraft Interface Requirements Document, 433-IRD-0001 Revision B*, NASA Goddard Space Flight Center, April 2002.

8 *GLAST LAT to Spacecraft Interface Control Document*, Spectrum Astro, Inc., February 2003.

9 *GLAST Spacecraft Interface Board Hardware Specification*, LAT Hardware Specification Document.

10 *LTX User Manual*, V1-1-0, LAT FSW User Manual, August 2003.

11 *Military Standard 1553B Notice 2*, United States Department of Defence, September 1986.

12 *Military Standard 1553B, Aircraft Internal Time Division Command/Response Multiplex Data Bus*, United States Department of Defence, September 1978.

13 *MSG User Manual*, LAT Flight Software User Manual.

14 *PBS Package Documentation,* LAT Flight Software Code Documentation.

15 *PMC-1553 Reference Manual, Rev 1.2*, Alphi Technology Corporation, June 1998.

16 *Recommendation for Space Data System Standards, Advanced Orbiting Systems, Networks and Data Links: Architectural Specification, Blue Book 701.0-B-3*, Consultative Committee for Space Data Systems, June 2001.

17 *Recommendation for Space Data System Standards, Telecommand Part 3, Data Management Service Architectural Specification, Blue Book 203.0-B-1*, Consultative Committee for Space Data Systems, January 1987.

18 *Telecommand and Telemetry Formats*, LAT-TD-02659.

19 “GASU Based Teststands - A hardware and software Primer”, Michael Huffer, LAT-TD-03664.

20 *Symmetricom* TTM635/637VME Time and Frequency Processor, revision B Users Guide, 8500-0138 February, 2004

21 LAT Flight Software, CMX Manual, version V2-2-3, *A. P. Waite*, updated 30, November 2004

22 “The GLT Electronics Module- Programming ICD specification”, Michael Huffer, LAT-TD-01545.

23 See the LAT Flight Software, web-site for the traveller documentation for LPA (currently undefined)

24 See the LAT Flight Software, web-site for the traveller documentation for datagrams (currently undefined)

25 See the LAT Flight Software, web-site for the traveller documentation for LCI (currently under project APP, package LCI)

26 "The EBM Electronics Module- Programming ICD specification", Michael Huffer, LAT-TD-01546.

27 "The Virtual Spacecraft (VSC) Users Manual", Michael Huffer, LAT-TD-05601.

28 "Online System Science Data Format (LDF)", Ric Claus, LAT-TD-07066 (see the I&T online web site documentation for the latest version of this document)

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

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	DataFlow Public Interface (DFI) Users Manual		
	Version:	0.5		
	Issue:	1		
	Edition:	English		
	ID:	LAT-TD-07689		
	Status:	not for public release		
	Created:	February 9, 2002		
	Date:	November 24, 2005		
	Access:	V:\GLAST\Electronics\Design Documents\DFI\0.5\frontmatter.fm		
	Keywords:	GASU Based Teststands		
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	Michael Huffer		
	Written by:	Michael Huffer		

Table 2 Approval sheet

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

Document Status Sheet

Table 3 Document Status Sheet

Title:	DataFlow Public Interface (DFI) Users Manual		
ID:	LAT-TD-07689		
Version	**Issue**	**Date**	**Reason for change**
0.2	1	11/10/2005	First internal release
0.4	1	11/10/2005	First internal release of calibration definitions. Includes updates as reflected from Ric's and JJ's comments.
0.5	1	11/24/2005	Includes updates as reflected from Eric's, James's, Ric's and JJ's comments. Includes first stab at making the configuration and results of LPA's event handlers public.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1
Introduction

Examples will go here!

Chapter 2
The DataFlow Interface

2.1 Overview

By LAT convention, information passed on the science stream is always organized in units of *datagrams* (see [23]). A Datagram is the common structure used to encapsulate all science data created and transmitted by Flight Software (FSW). The invariant contents of a datagram are encapsulated in the `Datagram` class described in Section 2.3. The function of this package is to support the *parsing* and *processing* of the information contained within a datagram. Parsing is the act of iterating over the logical structure of a datagram. At each logical boundary, the information contained at that boundary must be communicated to the user for application specific processing. The classes used to support parsing are all based on the Datagram`Parser` class described in Section 2.4.

The relationship between the classes of this package is illustrated in Figure 1:

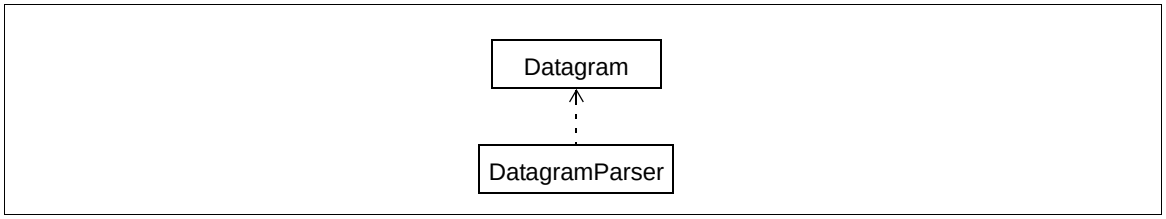

Figure 1 Class dependencies for the datagram support package

2.2 Name space - `Dfi`

2.3 Datagram

This class specifies the data structure assembled from arriving CCSDS science packets (see [23]). In addition to the member functions described below, the contents of datagram may be accessed through the datagram parser described in Section 2.4.

The definition for this class is contained in Listing 1:

Listing 1 Class definition for `Datagram`

```
 1:  class Datagram {
 2:    public:
 3:      Datagram(const Root&);
 4:   public:
 5:     ~Datagram();
 6:   public:
 7:      unsigned length()  const;
 8:      unsigned typeId()  const;
 9:      unsigned version() const;
10:    };
```

2.3.1 Constructor synopsis

`Datagram` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

2.3.2 Member synopsis

`length` This function returns the total length of the datagram in bytes. The member function has no arguments and throws no exceptions.

`typeId` Returns the datagram identifier. This function has no arguments and throws no exceptions.

`version` Returns the datagram version number. This function has no arguments and throws no exceptions.

2.4 Datagram Parser

This class iterates over the contents of a specified datagram (see Section 2.3). Logically, the contents of datagram are structured as a three level hierarchy of *root*, *contribution*, and *record* as illustrated in Figure 2:

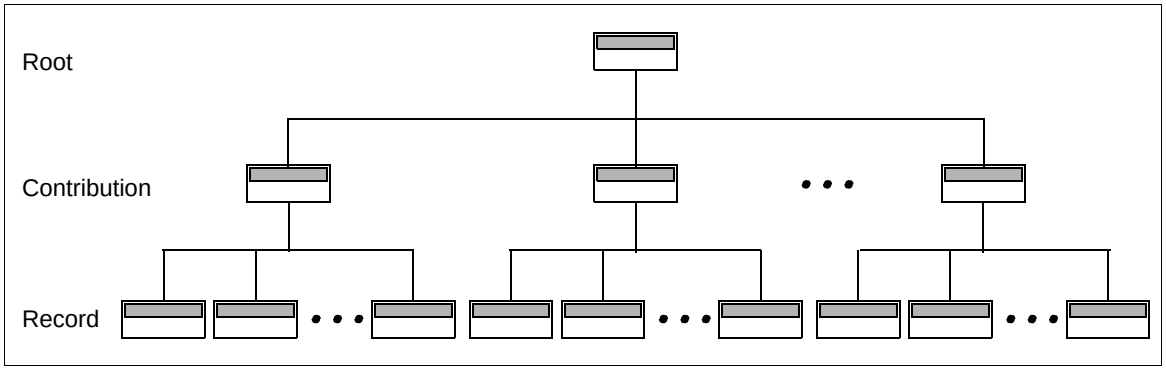

Figure 2 Abstract structure of a datagram

Each node, at each level, contains both a fixed (the "header") and variable amount of information. A datagram:

— contains always *one* and only one root

— *may* contain one or more contributions.

— contribution *may* contain one or more records.

Both the shape of each node (its size, contents and structure) and the number of nodes are determined by the type identifier of the datagram (see Section 2.3). Datagrams are always a multiple of 32-bit words and are always 32-bit word aligned.

For each level of the hierarchy the datagram parser provides a corresponding interface method. Derived classes are expected to provide an implementation for each one of these methods. The parser implements a depth first traversal of the hierarchy, triggering the appropriate method as it reaches each node. At any point in the traversal process parsing may be aborted by an appropriate implementation of any one of the these methods. Note that these methods are *protected*, the derived class is expected to hide these methods and re-export datagram contents in a fashion appropriate to datagram type. For example, see the `EventParser` described in Section 4.3. The parser initiates its traversal by calling its `parse` method. The argument to this method is the datagram to be parsed.

The definition for this class is contained in Listing 2:

Listing 2 Class definition for `DatagramParser`

```
 1:  class DatagramParser {
 2:    public:
 3:      DatagramParser(unsigned typeId);
 4:    public:
 5:      virtual ~DatagramParser();
 6:    public:
 7:      void parse(const Datagram&) const;
 8:    public:
 9:      unsigned typeId() const;
10:    private:
11:      virtual bool _process(const Root&)         = 0;
12:      virtual bool _process(const Contribution&) = 0;
13:      virtual bool _process(const Record&)       = 0;
14:      virtual bool _process()                    = 0;
15:    };
```

2.4.1 Constructor synopsis

`DatagramParser` The constructor's argument is a value corresponding to a datagram's type identifier (see Section 2.3). This value specifies which type of datagrams can be parsed (see the `parse` method described below). The constructor throws no exceptions.

2.4.2 Member synopsis

`parse` This method is called to initiate datagram parsing. The function has a single argument which is a reference to an object specifying the datagram to be parsed. This function compares the type identifier of that datagram to the value expected by the parser (see the constructor). If they do not agree, the exception described in Section 2.5.1 is thrown. If the structure of the datagram is not self-consistent, the exception described in Section 2.5.2 is thrown. This function returns no value.

`typeId` This method returns a value corresponding to a type identifier. This value specifies which type of datagrams may be parsed (see the `parse` method described above). This value was passed as an argument to the constructor. The function has no arguments and throws no exceptions.

`_process(const Root&)` This method is called *once* for each datagram. The method was triggered by the `parse` method (see above). This function is pure virtual and, therefore, its implementation will be provided by the sub-class. The argument is a reference to an object which describes the root. Note that the argument referenced by this method (and any objects *it* may reference) is *ephemeral*. When the method returns these objects are no longer accessible. Consequently, any information in these objects which the user finds necessary to persist across calls to this method

must be copied. This function returns a boolean specifying whether or not to abort parsing. If parsing is aborted control is immediately returned to the caller of the `parse` method. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. The function throws no exceptions.

`_process(const Contribution&)` This method is called once for each contribution in the parsed datagram. Contributions will be parsed in the order in which they occur in the datagram. Parsing is depth first: if the contribution contains one or more records, they will be processed before progressing to the next contribution. The method was triggered by the `parse` method (see above). This function is pure virtual and, therefore, its implementation will be provided by the sub-class. The argument is a reference to an object which describes the contribution. Note that the argument referenced by this method (and any objects *it* may reference) is *ephemeral*. When the method returns these objects are no longer accessible. Consequently, any information in these objects which the user finds necessary to persist across calls to this method must be copied. This function returns a boolean specifying whether or not to abort parsing. If parsing is aborted control is immediately returned to the caller of the `parse` method. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. The function throws no exceptions.

`_process(const Record&)` This method is called once for each record of each contribution in the parsed datagram. Records will be parsed in the order in which they occur in the datagram. This function is pure virtual and, therefore, its implementation will be provided by the sub-class. The argument is a reference to an object which describes the record. Note that the argument referenced by this method (and any objects *it* may reference) is *ephemeral*. When the method returns these objects are no longer accessible. Consequently, any information in these objects which the user finds necessary to persist across calls to this method must be copied. This function returns a boolean specifying whether or not to abort parsing. If parsing is aborted control is immediately returned to the caller of the `parse` method. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. The function throws no exceptions.

`_process()` This method is called *once* for each datagram. The method was triggered by the `parse` method (see above). This function is pure virtual and, therefore, its implementation will be provided by the sub-class. It will be called immediately before control is returned to the caller of the `parse` method and after the last contribution of the datagram has been parsed. This function returns a boolean specifying whether or not to abort parsing. If parsing is aborted control is immediately returned to the caller of the `parse` method. If the function returns `TRUE`, parsing *continues*. Note, that because this is the last method triggered, the returned value actually has no practical effect. If the function returns `FALSE`, parsing *aborts*. The function has no arguments and throws no exceptions.

2.5 Exceptions

2.5.1 Invalid Datagram Type

to be written.

2.5.2 Invalid Datagram Structure

to be written.

Chapter 3
The Event support package

3.1 Overview

While FSW transmits an arbitrary number of different types of datagrams on the science stream, many of these types share the common characteristic of being *event* oriented. This particular package captures those characteristics common to all event based datagrams. These characteristics are:

- They are produced in the context of a *Run*. While an accurate definition and description of a run is beyond the scope of this document, for the purposes of this document, a run may thought of as a time interval over which the state of FSW remains relatively constant. The specification of a run is encapsulated in the `RunInfo` class described in Section 3.5.
- They contain one or more events. An event has both meta and detector based information. For example, an event's *time* is considered one type of meta information. Meta information is encapsulated in the class `MetaEvent` (described in Section 3.4).
- They update run based statistics. For example, livetime and elapsed time are two statistics tallied over the lifetime of a run. Statistics are accumulated into *scalers*, one for each individual statistic. The set of scalers are encapsulated in the `Scalers` class described in Section 3.8.
- They have a well defined structure at both their beginning and end. The prefix information is called a datagram's *Open* context and the suffix information, the datagram's *Close* context. This information is encapsulated in respectively, the `OpenInfo` (see Section 3.6) and `CloseInfo` classes (see Section 3.7).

The classes of this package stand in isolation and are only useful in the context of a facility which generates event oriented datagrams. At this point there are two:

- the LAT Physics Analysis facility (LPA) discussed in chapter 4.
- the LAT Charge Injection facility (LCI) discussed in chapter 5.

The relationship between the classes of this package is illustrated in Figure 3:

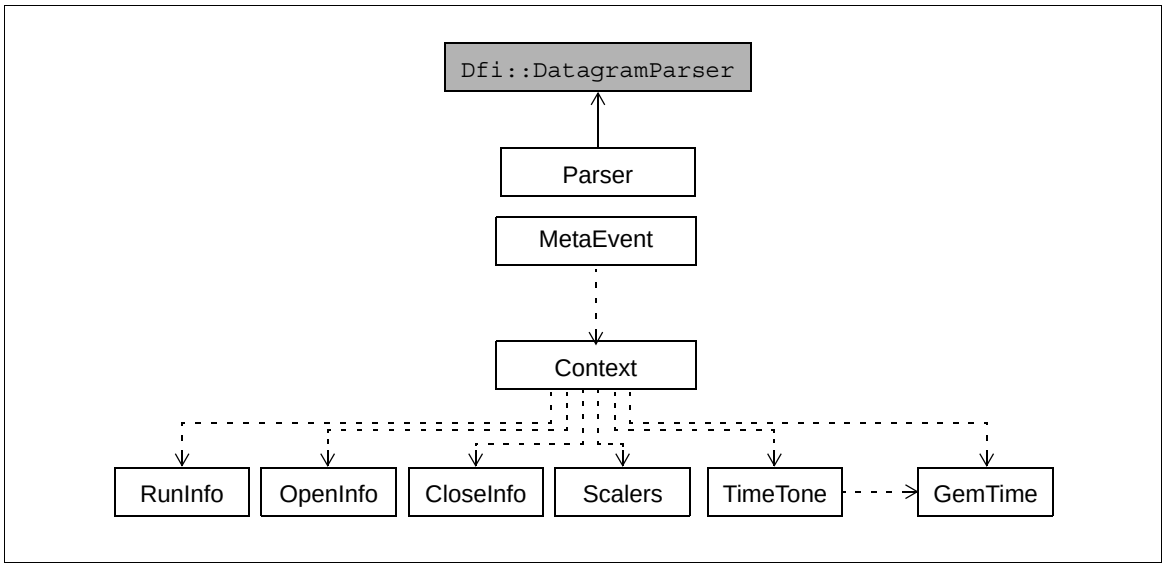

Figure 3 Class dependencies for the Event support package

3.2 Name space - `DfiEvent`

3.3 Event Parser

The principal function of this class is to iterate over the *events* contained within an event oriented datagram. This class is never used in isolation, but is instead sub-classed in order to parse a specific kind of event datagram. See, for example, the LPA parser described in Section 4.3.

Any event parsed in such a datagram has two components: meta information about the event and the event data itself. Typically, this data is transmitted, by FSW, to the ground *compressed*. Depending on how the data were compressed, decompression of an event requires a varying amount of external resources be available to the user. For example, LPA events at their maximum compression level require pedestal knowledge. This information is only present in a FSW database (see xxx) and therefore, to decompress those events requires a connection to that database. In many other cases, the event's meta information is alone sufficient to decompress the event. In short, higher levels of compression require more resources to decompress and these resources may or may not be available to the user. An argument to the parser's constructor specifies to the parser how many resources the user wishes to bring to bear when decompressing event data. This argument will then be used by the parser to garner

those resources. For example, to form a connection to a FSW database. If the argument specifies more resources then the parser is able to garner, an exception is declared.

In order to maximize the portability of the user's parser, it behoves the user to request only what resources they may need.

The definition for this class is contained in Listing 3:

Listing 3 Class definition for an event `Parser`

```
 1:  class Parser : public Dfi::DatagramParser {
 2:    public:
 3:      enum  Decompression {None};
 4:    public:
 5:      Parser(Decompression);
 6:    public:
 7:      virtual ~Parser();
 8:    public:
 9:      Decompression level() const;
10:    };
```

3.3.1 Constructor synopsis

`Parser` The constructor's argument is a enumeration specifying the maximum level of event data decompression supported by the parser. See Section 3.3 for information on how this argument is used. Note to myself, Amedeo, and others: I'm waiting for JJ to specify the various values of this enumeration. Until then, take the value(s) with a grain of salt. If the parser cannot support the specified decompression level the constructor will throw the exception specified in Section 3.12.1.

3.3.2 Member synopsis

`level` This function returns an enumeration which specifies the amount of resources supported for event decompression by the parser. This value was specified as an argument to the constructor (see above). This function has no arguments and throws no exceptions.

3.4 Context

This class encapsulates the information available when processing a *datagram*. Processing an datagram is always accomplished in the context of an event parser (see for example Section 3.3). Therefore, the contents of this object will always reflect their value corresponding

to where within a datagram the parser is, at any time, pointing. In turn this implies that some, (but not necessarily all) the values contained by this object will vary as a function of event (see Section 3.10).

The definition for this class is contained in Listing 4:

Listing 4 Class definition for `Context`

```
 1:  class Context{
 2:    public:
 3:      Context(const _QSE_ctx*);
 4:   public:
 5:     ~Context();
 6:    public:
 7:      const RunInfo&   run()      const;
 8:      const OpenInfo&  open()     const;
 9:      const CloseInfo& close()    const;
10:      const Scalers&   scalers()  const;
11:      const TimeTone&  current()  const;
12:      const TimeTone&  previous() const;
13:    };
```

3.4.1 Constructor synopsis

`Context` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within a datagram, the meta data necessary to construct an object of this class. The constructor throws no exceptions.

3.4.2 Member synopsis

`run` Returns a reference to an object containing the datagram's *run* information (see Section 3.5). This function has no arguments and throws no exceptions.

`open` Returns a reference to an object containing the datagram's *open* information (see Section 3.6). This function has no arguments and throws no exceptions.

`close` Returns a reference to an object containing the datagram's *close* information (see Section 3.7). This function has no arguments and throws no exceptions.

`scalers` Returns a reference to an object containing a run's scalers (see Section 3.8). This function has no arguments and throws no exceptions.

`current` This method returns a reference to an object which specifies the most recently arrived time-tone (see Section 3.9). This object (and the object returned by the method below) are used to both convert and correct LAT *relative* time to *absolute* time. This function has no arguments and throws no exceptions.

`previous` This method returns a reference to an object which specifies the previous time-tone (see Section 3.9). In other words, if the time-tone returned by the `current` method (see above) was delivered at time *n*, this method returns the time-tone at *n - 1*. This object (and the object returned by the method above) are used to both convert and correct LAT *relative* time to *absolute* time. This function has no arguments and throws no exceptions.

3.5 Run Information

This class encapsulates the description of a FSW run (see Section 3.1). This description will stay constant over the lifetime of a run (see section 3.6 and 3.7 for information on how to determine when the run changes). The definition for this class is contained in Listing 5:

Listing 5 Class definition for `RunInfo`

```
 1:  class RunInfo {
 2:    public:
 3:      RunInfo(const _QSE_ctx*);
 4:   public:
 5:     ~RunInfo();
 6:    public:
 7:      enum Platform {Lat, Testbed, Host};
 8:      Platform platform() const;
 9:    public:
10:      enum DataOrigin {Orbit, MonteCarlo, Ground};
11:      DataOrigin origin() const;
12:    public:
13:      unsigned id()          const;
14:      unsigned startedAt()   const;
15:    };
```

3.5.1 Constructor synopsis

`RunInfo` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within a datagram, the meta data necessary to construct an object of this class. The constructor throws no exceptions.

3.5.2 Member synopsis

`platform` This function returns an enumeration which specifies the system under which the datagram was generated. Valid systems are enumerated in Table 4. This function has no arguments and throws no exceptions.

`origin` This function returns an enumeration which specifies the origin of the platform's data. Valid origins are enumerated in Table 5. This function has no arguments and throws no exceptions.

`id` This function returns the run identifier. This identifier was established by the ground, transmitted to the LAT and simply reflected, with no interpretation by FSW, in the data. This function has no arguments and throws no exceptions.

`startedAt` Returns the absolute time at which the run began. This time is expressed as the number of seconds since the standard epoch (00:00:00.0 hours at January 1st, 2001). This function has no arguments and throws no exceptions.

Table 4 Enumeration of platform identifiers for the `RunInfo` class

Enumeration	The system which captured the data was...
`Lat`	The flight article
`Testbed`	A hardware simulated LAT
`Host`	A software simulated LAT

Table 5 Enumeration of origin identifiers for the `Run` class

Enumeration	The origin of the platform's data was...
`Orbit`	On orbit
`MonteCarlo`	From a simulation
`Ground`	From a ground-based test

3.6 Open Information

This information encapsulated by this class is present *once* per datagram parsed by an event parser (see, for example Section 3.3). On the whole, the class is self-describing, with the `requester` method returning what *entity* caused the datagram to be emitted and the `reason` method expressing *why*. For example, if the ground requested a run to be initiated, the `requester` method would return the enumeration `Operator` and the `reason` method would return the enumeration `Start`.

The definition for this class is contained in Listing 6:

Listing 6 Class definition for `OpenInfo`

```
 1:  class OpenInfo {
 2:    public:
 3:      OpenInfo(const _QSE_ctx*);
 4:    public:
 5:     ~OpenInfo();
 6:    public:
 7:      unsigned modeChanges() const;
 8:      unsigned datagrams()   const;
 9:    public:
10:      enum Reason {Start=1, Resume=2};
11:      Reason reason() const;
12:    public:
13:      enum Requester {Operator, Automatic, Unknown};
14:      Requester requester() const;
15:    public:
16:      enum Crate {Epu0, Epu1, Epu2, Siu0, Siu1, Aux};
17:      Crate crate() const;
18:    public:
19:      enum Mode {Normal, TOO, ARR=3, Calibration, Diagnostic};
20:      Mode mode() const;
21:    };
```

3.6.1 Constructor synopsis

`OpenInfo` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to the datagram. The constructor throws no exceptions.

3.6.2 Member synopsis

`modeChanges` This function returns the number of mode changes since the start of the run. See the `mode` method described below for a definition of potential mode changes. This function has no arguments and throws no exceptions.

`datagrams` This function returns the current number of datagrams sent by the platform since the start of the run. This function has no arguments and throws no exceptions.

`reason` This function returns an enumeration which specifies the reason the datagram was created. Valid reasons are enumerated in Table 8. The member function has no arguments and throws no exceptions.

`requester` This function returns an enumeration which specifies *who* requested the datagram to be created. Valid requesters are enumerated in Table 9. This function has no arguments and throws no exceptions.

`crate` This function returns an enumeration which specifies on which flight crate the datagram was generated. Valid crates are enumerated in Table 6. This function has no arguments and throws no exceptions.

`mode` This function returns an enumeration which specifies the mode under which the datagram was generated. Valid modes are enumerated in Table 7. This function has no arguments and throws no exceptions.

Table 6 Enumeration of crate identifiers for the `OpenInfo` class

Enumeration	meaning is...
`Epu0`	Event Processing Unit (EPU) *zero* (0)
`Epu1`	Event Processing Unit (EPU) *one* (1)
`Epu2`	Event Processing Unit (EPU) *two* (2)
`Siu0`	System Interface Unit (SIU) *zero* (0)
`Siu1`	System Interface Unit (SIU) *one* (1)
`Aux`	External Crate

Table 7 Enumeration of mode identifiers for the `OpenInfo` class

Enumeration	datagram was emitted...
`Normal`	while in normal mode
`TOO`	during a Target Of Opportunity (TOO)
`ARR`	during an Autonomous Re-point Request (ARR)
`Calibration`	during a detector calibration
`Diagnostic`	while operating in a diagnostic mode

Table 8 Enumeration of reason identifiers for the `OpenInfo` class

Enumeration	meaning is...
`Start`	First datagram after the start of a run.
`Resume`	First datagram after a run was paused and then restarted.

Table 9 Enumeration of requesters for the `OpenInfo` class

Enumeration	meaning is...
`Operator`	Datagram was emitted through operator request
`Automatic`	Datagram was one in a series of previously begun sequence.
`Unknown`	?

3.7 Close Information

This information encapsulated by this class is present *once* per datagram parsed by an event parser (see, for example Section 3.3). On the whole, the class is self-describing, with the `requester` method returning what *entity* caused the datagram to be closed and the `reason` method expressing *why*. For example, if the ground requested a run to stop, the `requester` method would return the enumeration `Operator` and the `reason` method would return the enumeration `Stop`.

The definition for this class is contained in Listing 7:

Listing 7 Class definition for `CloseInfo`

```
 1:  class CloseInfo {
 2:    public:
 3:      CloseInfo(const _QSE_ctx*);
 4:   public:
 5:     ~CloseInfo();
 6:    public:
 7:      enum Reason {Stop, Pause, Abort, Unknown};
 8:      Reason reason() const;
 9:    public:
10:      enum Requester {Operator,
11:                      Automatic,
12:                      TimedOut,
13:                      CountedOut,
14:                      Full,
15:                      Unknown};
16:      Requester requester() const;
17:    };
```

3.7.1 Constructor synopsis

`CloseInfo` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness.The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

3.7.2 Member synopsis

`reason` This function returns an enumeration which specifies the reason the datagram was closed. Valid reasons are enumerated in Table 10. The member function has no arguments and throws no exceptions.

`requester` This function returns an enumeration which specifies *who* requested the datagram to be closed. Valid requesters are enumerated in Table 11. This function has no arguments and throws no exceptions.

Table 10 Enumeration of reason identifiers for the `CloseInfo` class

Enumeration	meaning is...
`Stop`	Last datagram in a run.
`Pause`	Last datagram before a run was paused.
`Abort`	Last datagram before a run was aborted.
`Unknown`	?

Table 11 Enumeration of requesters for the `CloseInfo` class

Enumeration	meaning is...
`Operator`	Datagram was closed through operator request
`Automatic`	Datagram was one in a series of previously begun sequence.
`TimedOut`	Maximum Time limit per datagram was reached.
`CountedOut`	Maximum events/datagram limit reached
`Full`	Maximum size limit was reached
`Unknown`	?

3.8 Scalers

This class represents the statistics (scalers) accumulated by FSW over the lifetime of a run. Each scaler has a corresponding method which returns the current value of that scaler. For example, the `livetime` method returns the amount of accumulated livetime. As these scaler are accumulated over the lifetime of a run, they are initialized each time a run is initiated. The scalers are updated for each *event* parsed by an event parser (see, for example Section 3.3). Note, that the values returned by the methods of this class are 64-bit unsigned integers.

The definition for this class is contained in Listing 8:

Listing 8 Class definition for `Scalers`

```
 1: class Scalers {
 2:   public:
 3:     Scalers(const _QSE_ctx*);
 4:  public:
 5:    ~Scalers();
 6:   public:
 7:     unsigned long long int elapsed()   const;
 8:     unsigned long long int livetime()  const;
 9:     unsigned long long int prescaled() const;
10:     unsigned long long int discarded() const;
11:     unsigned long long int sequence()  const;
12:     unsigned long long int deadzone()  const;
13:   };
```

3.8.1 Constructor synopsis

`Scalers` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

3.8.2 Member synopsis

`elapsed` This function returns the current elapsed time of the run. This time is expressed in LAT clock-tics. Nominally, one clock-tic is equal to 50 nanoseconds. This function has no arguments and throws no exceptions.

`livetime` This function returns the current amount time livetime (see [22]) since the run began. This time is expressed in LAT clock-tics. Nominally, one clock-tic is equal to 50 nanoseconds. This function has no arguments and throws no exceptions.

`prescaled` This function returns the current number of events prescaled away since the run began (see [22]). This function has no arguments and throws no exceptions.

`discarded` This function returns the current number of events discarded due to deadtime since the run began (see [22]). This function has no arguments and throws no exceptions.

`sequence` This function returns the current number of events discarded due to deadtime since the run began (see [22]). This function has no arguments and throws no exceptions.

`deadzone` This function returns the current number of times the GEM was in the "deadzone" (see [22]). The GEM takes two clock cycles to recover from one window opening before it may open another. During this time the GEM is "dead". A window wishing to open during in this two clock period is caught in the "deadzone". This function has no arguments and throws no exceptions.

3.9 Time Tone

Time-Tone messages are received by the LAT from the spacecraft once per second. This class represents the contents of such a message. As these messages arrive, they are embedded in the next available datagram and emitted onto the science stream. These messages reflect the absolute time and are used to correct any local (LAT relative) measured time, for example, the time at which an event was created (see Section 3.4). As in any transmitted message, the possibility exists that either the acquisition of the message's data, or its transmission could fail. In such a case, the `invalid` method will return `TRUE`. It is important to establish the validity of a Time-Tone message *before* using any of its timing information, as this information is undefined unless the `invalid` method returns `FALSE`. If the method does return `TRUE`, the exact reason *why* the message is invalid can be determined by probing each of the error methods. These methods (for example, `gpsMissing`) return a boolean indicating whether or not the corresponding error is was present when constructing the message. Note, that the message could be invalid for *more* then one of these reasons.

The definition for this class is contained in Listing 9:

Listing 9 Class definition for `TimeTone`

```
 1:  class TimeTone {
 2:    public:
 3:      TimeTone(const _QSE_ctx*);
 4:   public:
 5:     ~TimeTone();
 6:   public:
 7:      unsigned long  incomplete()  const;
 8:      unsigned long  timeSecs()    const;
 9:      const GemTime& timeHack()    const;
10:      unsigned  long flywheeling() const;
11:   public:
12:      bool missingGps()      const;
13:      bool missingCpuPps()   const;
14:      bool missingLatPps()   const;
15:      bool missingTimeTone() const;
16:    };
```

3.9.1 Constructor synopsis

`TimeTone` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

3.9.2 Member synopsis

`incomplete` Returns a positive (non-*zero* value) if the acquisition of the time-tone information could not complete successfully. If the acquisition of this information did complete, a value of *zero* (0) is returned. This function has no arguments and throws no exceptions.

`timeSecs` Returns the *absolute* time at the time hack. This time is defined as the number of seconds since the standard epoch (00:00:00.0 hours at January 1st, 2001) in seconds. This function has no arguments and throws no exceptions.

`timeHack` Returns a reference to an object corresponding to the *local* time at the time hack. (see Section 3.11) This function has no arguments and throws no exceptions.

`flywheeling` Returns a value specifying how many times a CPU failed to construct its Time-Tone message. In such a case, a "fake" message is synthesized and an internal counter increments. While incrementing this counter the CPU is said to be "fly wheeling". The counter is reset at the first opportunity the CPU is able to successfully construct a message. As a message is constructed once per second,

this return value could also be interpreted as the number of seconds the CPU has fly wheeled. If the returned value of this method is non-zero, one or more of the error flags (see the methods above) will be asserted. This function has *no* arguments and throws *no* exceptions.

missingGps Returns a boolean specifying whether the time indicated within the time specified in the message was synchronized by the spacecraft's GPS receiver and can, therefore, be assumed to be stable with respect to the LAT's clock. If the value returned is TRUE, the message was synthesized with respect to the spacecraft's local clock (the spacecraft is "fly wheeling"). If the value returned is FALSE, the time was synthesized with respect to the spacecraft's GPS receiver. This function has *no* arguments and throws *no* exceptions.

missingCpuPps Returns a boolean specifying whether the arrival of the 1-PPS signal at the CPU timed out. If the value returned is TRUE, the 1-PPS signal timed out, if FALSE it did not. This function has *no* arguments and throws *no* exceptions.

missingLatPps Returns a boolean specifying whether the arrival of the 1-PPS signal at the LAT timed out. If the value returned is TRUE, the 1-PPS signal timed out, if FALSE it did not. This function has *no* arguments and throws *no* exceptions.

missingTimeTone Returns a boolean specifying whether the arrival of the time tone message from spacecraft to LAT timed out. If the value returned is TRUE, the 1-PPS signal timed out, if FALSE it did not.This function has *no* arguments and throws *no* exceptions.

3.10 Meta Event

This class encapsulates the information available when processing an *event*. Processing an event is always accomplished in the context of an event parser (see for example Section 3.3). Therefore, the contents of this object will always reflect their value at the time the event was created. In turn this implies that some, (but not necessarily all) the values contained by this object will vary as a function of event. For example, the event time changes for each event, while the information represented by the `run` method would only change when a new run is declared.

The definition for this class is contained in Listing 10:

Listing 10 Class definition for `MetaEvent`

```
 1:  class MetaEvent {
 2:    public:
 3:      MetaEvent(const _QSE_ctx*);
 4:   public:
 5:     ~MetaEvent();
 6:    public:
 7:      unsigned       timeTics() const;
 8:      const GemTime& timeHack() const;
 9:      const Context& context()  const;
10:    };
```

3.10.1 Constructor synopsis

`MetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within a datagram, the meta data necessary to construct an object of this class. The constructor throws no exceptions.

3.10.2 Member synopsis

`timeTics` Returns the time (in tics) at which the event was triggered. The time is measured *relative* to the LAT's local timebase (see [22]). This clock increments at the LAT system clock rate (nominally 20 MHZ) and therefore, nominally one tic equals 50 nano-seconds. This function has no arguments and throws no exceptions.

`timeHack` Returns a reference to an object corresponding to the *local* time at the time hack. (see Section 3.11) This function has no arguments and throws no exceptions.

`context` Returns a reference to an object containing the datagram's *context* information (see Section 3.4). This function has no arguments and throws no exceptions.

3.11 GEM Time

Each time the 1-PPS signal arrives at the GEM, the GEM takes two actions:

— It samples and stores the current value of the LAT clock. This clock increments at the LAT system clock rate (nominally 20 MHZ) and therefore, nominally one count equals 50 nanoseconds.

— It increments a 7-bit counter and then samples and stores its current value. This value is initialized to *zero* whenever the GEM resets or the counter overflows.

These two quantities are stored in a GEM register (see [22]) where they may sampled at any time. Note that this nominally this register changes its value only once per second. An instantiation of this class represents one *sample* of this register. This register is sampled at two significant times:

— at the arrival of the 1-PPS signal at any one of its crates (by FSW). This sample is contained in `TimeTone` (see Section 3.9).

— when an event comes into existence (by the DAQ system). This sample is contained in the `MetaEvent` (see Section 3.4).

The definition for this class is contained in Listing 11:

Listing 11 Class definition for `GemTime`

```
1:  class GemTime {
2:    public:
3:      GemTime(const _QSE_ctx*);
4:   public:
5:     ~GemTime();
6:   public:
7:      unsigned tics()  const;
8:      unsigned hacks() const;
9:    };
```

3.11.1 Constructor synopsis

`GemTime` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

3.11.2 Member synopsis

`tics` Returns the time *relative* to the LAT's local timebase (see [22]). The counter corresponding to this clock increments at the LAT system clock rate (nominally 20 MHZ) and therefore, nominally one tic equals 50 nanoseconds. This function has no arguments and throws no exceptions.

`hacks` Returns the current number of 1-PPS time hacks (see [22]). The counter corresponding to this clock increments at each received time hack (1 HZ) and therefore, nominally one tic equals 1 second. This counter has a range of seven (7) bits, which on overflow simply wraps. This function has no arguments and throws no exceptions.

3.12 Exceptions

3.12.1 Decompression not supported

to be written.

Chapter 4
The LPA support package

4.1 Overview

This package supports the parsing of datagrams whose origin were the LAT Physics Analysis facility (LPA). Parsing implies the search for, and the processing of, the events contained within a LPA datagram. As one advances through the datagram processing events, the meta information for that event is made available (see Section 4.4). The act of parsing is encapsulated in the `EventParser` class described in Section 4.3. For each event the user has the option of decompressing and then accessing the event's data (see Section 4.5). Note, that this classes in this package depend heavily on the classes found in the `Event` package (see Chapter 3).

The relationship between the classes of this package is illustrated in Figure 4:

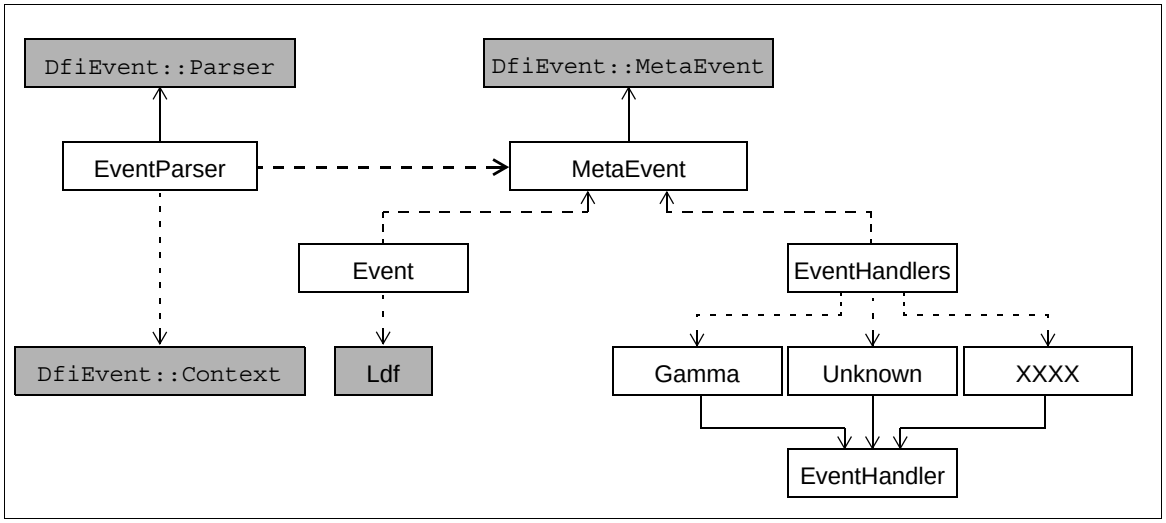

Figure 4 Class dependencies for the LPA support package

4.2 Name space - `DfiLpa`

4.3 Event Parser

The principal function of this class is to iterate over the *events* contained within a LPA datagram. In order to do so, the class defines three virtual methods which must be satisfied by a derived class:

- `Open`, which is called once, *before* any events are parsed.
- `Process`, which is called for *each* event of the datagram and represents the work to be performed by the derived class with respect to that event.
- `Close`, which is called once , *after* all events of the datagram have been parsed.

In short, three user methods are triggered: once at the beginning of the datagram, once at its end and once for every event in between. Each triggered method, is passing as an argument, a reference to the appropriate information for that method (see Section 3.4 and Section 4.5). Note that the parser processes events in the order in which they are stored in the datagram.

To initiate parsing the base class's `parse` method is invoked (see Section 2.4). This method takes as an argument the datagram to be parsed. Note, that the type identifier of the datagram must correspond to an LPA datagram, if not, an exception is declared.

The definition for this class is contained in Listing 12:

Listing 12 Class definition for LPA `EventParser`

```
 1: class EventParser : public DfiEvent::Parser {
 2:   public:
 3:     Parser(DfiEvent::Decompression);
 4:   public:
 5:     virtual ~EventParser();
 6:   public:
 7:     virtual bool open(const DfiEvent::Context&)  = 0;
 8:     virtual bool process(const MetaEvent&)       = 0;
 9:     virtual bool close(const DfiEvent::Context&) = 0;
10:   };
```

4.3.1 Constructor synopsis

`EventParser` This constructor creates a object to process the events of a LPA datagram. The constructor's argument is a enumeration specifying the maximum level of event data decompression supported by the parser. See Section 4.5 for information on how this argument is used. If the parser cannot support the specified decompression level the constructor will throw the exception specified in Section 3.12.1.

4.3.2 Member synopsis

`open` This method will be called *once* per datagram. It is called immediately *before* any events of the datagram are parsed (see below). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *opened*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`process` This method will be called for each event present in the parsed datagram. The argument is a reference to an object which provides a description of the event to be processed (see Section 4.4). Note, that the contents of this object reflect their value at the time of the corresponding event. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`close` This method will be called *once* per datagram. It is called immediately *after* all (if any) events of the datagram are parsed (see above). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *closed*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

4.4 Meta Event

A reference to an instance of this class is passed into the `process` method of the event parser (described in Section 4.3). This method is called by the parser for each event contained within a datagram. As the parser advances through a datagram's events, the contents of the `MetaEvent` are constantly changing to reflect their value for any, one, specified event. Note that this class it is derived from `DfiEvent::MetaEvent` (see Section 3.4). In order to go from a *meta* event to the event's *data*, requires instantiating an `Event` (see Section 4.5), passing as an argument an object of this class.

The definition for this class is contained in Listing 13:

Listing 13 Class definition for LPA `MetaEvent`

```
1:  class MetaEvent : public DfiEvent::MetaEvent {
2:    public:
3:      MetaEvent(const _QSE_ctx*);
4:    public:
5:     ~MetaEvent();
6:    public:
7:      unsigned softwareKey() const;
8:      unsigned hardwareKey() const;
9:    };
```

4.4.1 Constructor synopsis

`MetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

4.4.2 Member synopsis

`softwareKey` This function returns the key which uniquely defines the current software configuration. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

`hardwareKey` This function returns the key which uniquely defines the current hardware configuration. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

4.5 Event

This class is used to extract from a `MetaEvent` (see Section 4.4) the event's data. Typically, this information was transmitted from the LAT to the ground *compressed*. Instantiating an object of this class will (as appropriate) automatically decompress the event's data. Once expansion has been performed by the constructor, the `data` method can be called to access the event's data. The type of object returned by this method is `Ldf`. A discussion of this structure is beyond the scope of this document (see [28]). Note, that under some circumstances the event data cannot be decompressed. For example, the compression level of the event is greater then the compression level supported for the parser which located this event (see Section 3.3). In such a case, the constructor will throw an exception . Finally, the event data is *copied* to this

object, consequently, any object instantiated from this class may have a lifetime completely independent of both the parser and the datagram from which it was derived.

The definition for this class is contained in Listing 14:

Listing 14 Class definition for `LPA Event`

```
1:  class Event {
2:    public:
3:      Event(const MetaEvent&);
4:    public:
5:     ~Event();
6:    public:
7:      const Ldf& data() const;
8:    };
```

4.5.1 Constructor synopsis

`Event` The argument is a reference to the meta-event which contains the event data to be decompressed. If the event cannot be expanded the exception described in Section 4.9.1 is thrown.

4.5.2 Member synopsis

`data` This method returns a reference to the decompressed event data. The function throws no exceptions.

4.6 Event Handlers

To be written.

The definition for this class is contained in Listing 15:

Listing 15 Class definition for `EventHandlers`

```
 1:  class EventHandlers {
 2:    public:
 3:      EventHandlers();
 4:   public:
 5:      virtual ~EventHandlers();
 6:   public:
 7:      virtual boolean has(const Gamma&)   const = 0;
 8:      virtual boolean has(const Unknown&) const = 0;
 9:      virtual boolean has(const XXXX&)    const = 0;
10:   public:
11:      void find(const MetaEvent&) const;
12:    };
```

4.6.1 Constructor synopsis

the default.

4.6.2 Member synopsis

`find` This method is called to initiate finding the set of Event Handlers in the specified meta event. The function has a single argument which is a reference to an object specifying the meta event to be searched for Event Handlers. This function throws no exceptions and returns no value.

`has(const Gamma&)` This method will be called for each gamma based event handler present in the searched meta event. The argument is a reference to an object which provides a description of the event handler (see Section 4.7). Note, that the contents of this object reflect their value at the time of the corresponding meta event. This function returns a boolean specifying whether or not to abort searching the meta event. If the function returns `TRUE`, searching *continues*. If the function returns `FALSE`, searching *aborts*. When the parser aborts, control is returned to the caller of the `find` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

4.7 Gamma Handler

To be written.

The definition for this class is contained in Listing 16:

Listing 16 Class definition for `Gamma`

```
 1:  class Gamma : public EventHandler {
 2:    public:
 3:      Gamma(const _QSE_ctx*);
 4:   public:
 5:     ~Gamma();
 6:   public:
 7:      unsigned xxx() const;
 8:    };
```

4.7.1 Constructor synopsis

`Gamma` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

4.7.2 Member synopsis

to be written.

4.8 Event Handler

To be written.

The definition for this class is contained in Listing 17:

Listing 17 Class definition for `EventHandler`

```
 1:  class EventHandler {
 2:    public:
 3:      EventHandler(const _QSE_ctx*);
 4:   public:
 5:     ~EventHandler();
 6:   public:
 7:      unsigned file()     const;
 8:      unsigned priority() const;
 9:      unsigned index()    const;
10:    };
```

4.8.1 Constructor synopsis

`EventHandler` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

4.8.2 Member synopsis

`file` This function returns the key which uniquely defines the Event Handler configuration. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

`priority` This function returns the key which uniquely defines the Event Handler Priority. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

`index` This function returns the key which uniquely defines the Event Handler index. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

4.9 Exceptions

4.9.1 Decompression Failed

to be written.

Chapter 5
LCI support

5.1 Overview

This package supports the parsing of datagrams whose origin were the LAT Charge Injection facility (LCI). Parsing implies the search for, and the processing of, the events contained within a LCI datagram. As one advances through the datagram processing events, the meta information for that event is made available. This information varies as a function of the subsystem being calibrated, however, each subsystem generates a common set of information and this information is encapsulated in the `MetaEvent` class (see Section 5.4). Parsing is also subsystem dependent, however, just as in the case for event information, all parsing derives from a common base class (`EventParser` described in Section 3.3). For each event the user has the option of decompressing and then accessing the event's data (see Section 5.6). Note that the classes of this package depend heavily on the classes found in the `Event` package, described in Chapter 3.

In short, this chapter describes the set of classes whose behavior is independent of subsystem calibrated and a description of those classes whose behavior is subsystem dependent are relegated to their own individual chapters. In particular:

- The ACD (Chapter 6)
- The Tracker (Chapter 7)
- The Calorimeter (Chapter 8)

The relationship between *all* the classes of this package is illustrated in Figure 5:

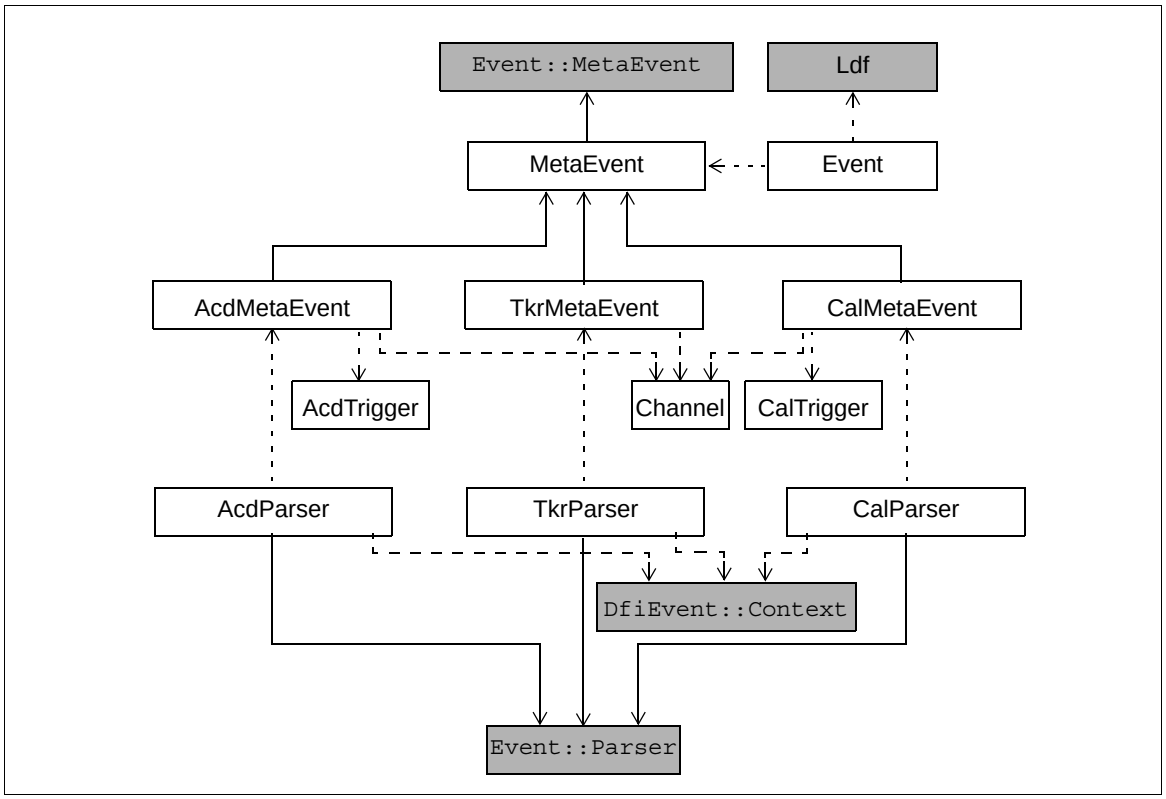

Figure 5 Class dependencies for the LCI support package

5.2 Name space - `DfiLci`

5.3 Constants

This constant is used as a sentinel value in the returns from many of the helper functions in the classes described below. For example, the method `single` of `Channel` (see Section 5.6).

```
static const unsigned short UNDEFINED = 0xFFFF;
```

5.4 Meta Event

A reference to an instance of this class is passed into the `process` method of the event parser (described in Section 3.3). This method is called by the parser for each event contained within a datagram. As the parser advances through a datagram's events, the contents of the `MetaEvent` are constantly changing to reflect their value for any, one, specified event. Note

that this class it is derived from `DfiEvent::MetaEvent` (see Section 3.4). In order to go from a *meta* event to the event's *data*, requires instantiating an `Event` (see Section 5.6), passing as an argument an object of this class.

The definition for this class is contained in Listing 18:

Listing 18 Class definition for LCI `MetaEvent`

```
 1:  class MetaEvent : public DfiEvent::MetaEvent {
 2:    public:
 3:      MetaEvent(const _QSE_ctx*);
 4:    public:
 5:     ~MetaEvent();
 6:    public:
 7:      bool     autoRange()        const;
 8:      bool     zeroSuppresion()   const;
 9:      unsigned periodicPrescale() const;
10:    public:
11:      unsigned softwareKey() const;
12:      unsigned writeCfg()    const;
13:      unsigned readCfg()     const;
14:    };
```

5.4.1 Constructor synopsis

`MetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

5.4.2 Member synopsis

`autoRange` This function returns a boolean which specifies whether or not for the specified calibration auto ranging was enabled. If the returned value is `TRUE`, auto ranging was enabled. If the returned value is `FALSE`, it was not. This function has no arguments and throws no exceptions.

`zeroSuppression` This function returns a boolean which specifies whether or not for the specified calibration data the data were zero-suppressed. If the returned value is `TRUE`, data were zero-suppressed. If the returned value is `FALSE`, they were not. This function has no arguments and throws no exceptions.

`periodicPrescale` This function returns the value which the GEM's periodic trigger was prescaled. Consequently, this value (in conjunction with the value of the system clock[1]), is used to determine the actual rate of the periodic trigger. A value of one (1) divides the system clock rate by two, a value of two (2) divides the system clock rate by three, and so forth. The periodic trigger is used by LCI to drive the rate at which it both injects charge and reads out the detector. This function has no arguments and throws no exceptions.

`softwareKey` This function returns the key which uniquely defines the LCI software configuration. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned science data. This function has no arguments and throws no exceptions.

`writeCfg` This function returns the LATC file key which uniquely defines the hardware configuration initially applied for the calibration. This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in returned the returned calibration data. This function has no arguments and throws no exceptions.

`readCfg` This function returns the LATC file key which uniquely defines the map of LAT hardware nodes to *ignore* when reading or verifying the configuration initially applied for the calibration (see the method above). This key was established by the ground, transmitted to the LAT, where it was applied and reflected, by FSW in the returned calibration data. This function has no arguments and throws no exceptions.

5.5 Event

This class is used to extract from a `MetaEvent` (see Section 5.4) the event's data. Typically, this information was transmitted from the LAT to the ground *compressed*. Instantiating an object of this class will (as appropriate) automatically decompress the event's data. Once expansion has been performed by the constructor, the `data` method can be called to access the event's data. The type of object returned by this method is `Ldf`. A discussion of this structure is beyond the scope of this document (see [28]). Note, that under some circumstances the event data cannot be decompressed. For example, the compression level of the event is greater then the compression level supported for the parser which located this event (see Section 3.3). In such a case, the constructor will throw an exception . Finally, the event data is *copied* to this object, consequently, any object instantiated from this class may have a lifetime completely independent of both the parser and the datagram from which it was derived.

The definition for this class is contained in Listing 19:

1. Nominally 20 MHZ.

Listing 19 Class definition for `LCI Event`

```
 1:  class Event {
 2:    public:
 3:      Event(const MetaEvent&);
 4:    public:
 5:     ~Event();
 6:    public:
 7:      const Ldf& data() const;
 8:    };
```

5.5.1 Constructor synopsis

`Event` The argument is a reference to the meta-event which contains the event data to be decompressed. If the event cannot be expanded the exception described in Section 8.5.1 is thrown.

5.5.2 Member synopsis

`data` This method returns a reference to the decompressed event data. The function throws no exceptions.

5.6 Channel

To be written. The definition for this class is contained in Listing 20:

Listing 20 Class definition for `Channel`

```
 1:  class Channel {
 2:    public:
 3:      Channel(const _QSE_ctx*);
 4:   public:
 5:     ~Channel();
 6:   public:
 7:      unsigned short single() const;
 8:      bool           all()    const;
 9:      bool           latc()   const;
10:    };
```

5.6.1 Constructor synopsis

`Channel` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

5.6.2 Member synopsis

`single` This function returns the single channel number enabled in the corresponding calibration data. If this value is equal to the constant `UNDEFINED` (see Section 5.3), a single channel was *not* enabled and one of the methods `all`, or `latc` will return a legitimate value. The interpretation of the returned channel number is dependent on which subsystem was calibrated. See Table 12, Table 13, and Table 14. This function has no arguments and throws no exceptions.

`all` This function returns a boolean which specifies whether or not all channels were enabled in the corresponding calibration data. If the returned value is `TRUE`, all channels were enabled. If the returned value is `FALSE`, all channels were not enabled. The interpretation of *all* is dependent on which subsystem was calibrated. See Table 12, Table 13, and Table 14. This function has no arguments and throws no exceptions.

`latc` This function returns a boolean which specifies whether the channels enabled in the corresponding calibration data were determined by an external (`LATC`) database. If the returned value is `TRUE`, the enabled channels were determined by the database. If the returned value is `FALSE`, they were not. This function has no arguments and throws no exceptions.

Chapter 6
ACD calibration support

6.1 Overview

The two classes described below are used to process the information returned from LCI when calibrating the ACD. These classes are based on the classes described in Chapter 5. Each time the calibration system (LCI) injects charge a corresponding event is generated. An event's information is encapsulated in the `AcdMetaEvent` class (see Section 6.3). These calibration events are contained in a series of datagrams, sent down to the ground on the science stream (see [27]) by LCI. In order to process the events within any one datagram the user parses the datagram using the `AcdParser` described in Section 6.2.

6.2 ACD Parser

The principal function of this class is to iterate over the a LCI datagram generated during an ACD electronics calibration. In order to do so, the class defines three virtual methods which must be satisfied by a derived class:

- `Open`, which is called once, *before* any events are parsed.
- `Process`, which is called for *each* event of the datagram and represents the work to be performed by the derived class with respect to that event.
- `Close`, which is called once , *after* all events of the datagram have been parsed.

In short, three user methods are triggered: once at the beginning of the datagram, once at its end, and once for every event in between. Each triggered method, is passing as an argument, a reference to the appropriate information for that method (see Section 3.4 and Section 6.3). Note that the parser processes events in the order in which they are stored in the datagram.

To initiate parsing the base class's `parse` method is invoked (see Section 2.4). This method takes as an argument a reference to the datagram to be parsed. Note, that the type identifier of

the datagram must correspond to an LCI datagram resulting from an ACD calibration. If not, an exception is declared.

The definition for this class is contained in Listing 21:

Listing 21 Class definition for `AcdParser`

```
 1:  class AcdParser : public DfiEvent::Parser {
 2:    public:
 3:      AcdParser(DfiEvent::Decompression);
 4:    public:
 5:      virtual ~AcdParser();
 6:    public:
 7:      virtual bool open(const DfiEvent::Context&)   = 0;
 8:      virtual bool process(const AcdMetaEvent&)     = 0;
 9:      virtual bool close(const DfiEvent::Context&) = 0;
10:    };
```

6.2.1 Constructor synopsis

`AcdParser` The constructor's argument is a enumeration specifying the maximum level of event data decompression supported by the parser. See Section 3.3 for information on how this argument is used. If the parser cannot support the specified decompression level the constructor will throw the exception specified in Section 3.12.1.

6.2.2 Member synopsis

`open` This method will be called *once* per datagram. It is called immediately *before* any events of the datagram are parsed (see below). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *opened*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`process` This method will be called for each event present in the parsed datagram. The argument is a reference to an object which provides a description of the event to be processed (see Section 6.3). Note, that the contents of this object reflect their value at the time of the corresponding event. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`close` This method will be called *once* per datagram. It is called immediately *after* all (if any) events of the datagram are parsed (see above). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *closed*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

6.3 ACD Meta Event

A reference to an instance of this class is passed into the `process` method of the event parser (described in Section 6.2). This method is called by the parser for each event contained within a datagram. As the parser advances through a datagram's events the contents of the meta event are constantly changing to reflect their value for any, one, specified event. Note that this class it is derived from the common event calibration class `MetaEvent` (see Section 5.4). In order to go from a *meta* event to the event's *data*, requires instantiating an `Event` (see Section 5.4), passing as an argument an object of this class.

The definition for this class is contained in Listing 22:

Listing 22 Class definition for `AcdMetaEvent`

```
 1:  class AcdMetaEvent : public MetaEvent {
 2:    public:
 3:      AcdMetaEvent(const _QSE_ctx*);
 4:    public:
 5:     ~AcdMetaEvent();
 6:    public:
 7:      unsigned short injected()  const;
 8:      unsigned short delay()     const;
 9:      unsigned short threshold() const;
10:      AcdTrigger     trigger     const;
11:      Channel        from        const;
12:    };
```

6.3.1 Constructor synopsis

`AcdMetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

6.3.2 Member synopsis

`injected` This function returns a value which defines the amount of charge which was injected for the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the amount of injected charge was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`delay` This function returns a value which defines the time delay between the injection of the charge and the TACK used to read out the corresponding calibration data. This value is specified in units of LAT clock tics, where one tic is nominally 50 nanoseconds. If the delay was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`threshold` This function returns a value which defines the charge threshold necessary to cross in order to generate the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the threshold was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`trigger` This method returns an object (see Section 6.4) which specifies the thresholds for the ACD's signals which are sent to the trigger system. This function has no arguments and throws no exceptions.

`from` This method returns an object (see Section 5.6) which describes the channel(s) enabled in the corresponding calibration data. The interpretation of channel number is subsystem dependent. See Table 12 for the interpretation of an ACD channel. This function has no arguments and throws no exceptions.

Table 12 Type definition for the channel class when used in an ACD calibration.

Member	Interpretation	Range[1]	
		minimum	maximum
`single`	One channel is enabled.	0	215
`all`	All channels in all GAFEs are enabled.	N/A	N/A

1. In decimal.

6.4 ACD Trigger Discriminators

An instance of this class is returned from the ACD's meta event (see Section 6.3). This class specifies the values of the discriminators which specify the threshold for the production of the trigger signals generated by the ACD and used by the trigger system

The definition for this class is contained in Listing 23:

Listing 23 Class definition for `AcdTrigger`

```
 1: class AcdTrigger {
 2:   public:
 3:     AcdTrigger(const _QSE_ctx*);
 4:   public:
 5:    ~AcdTrigger();
 6:   public:
 7:     unsigned short veto()        const;
 8:     unsigned short vetoVernier() const;
 9:     unsigned short hld()         const;
10:   };
```

6.4.1 Constructor synopsis

`AcdTrigger` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

6.4.2 Member synopsis

`veto` This function returns a value which (in conjunction with the value below) specifies the discrimination threshold necessary to toggle the ACD's veto signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`vetoVernier` This function returns a value which (in conjunction with the value below) specifies the discrimination threshold necessary to toggle the ACD's veto signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`hld` This function returns a value which specified the (High Level) discrimination threshold necessary to toggle the ACD's CNO signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

Chapter 7
Tracker calibration support

7.1 Overview

The two classes described below are used to process the information returned from LCI when calibrating the *tracker*. These classes are based on the classes described in Chapter 5. Each time the calibration system (LCI) injects charge a corresponding event is generated. An event's information is encapsulated in the `TkrMetaEvent` class (see Section 7.3). These calibration events are contained in a series of datagrams, sent down to the ground on the science stream (see [27]) by LCI. In order to process the events within any one datagram the user parses the datagram using the `TkrParser` described in Section 7.2.

7.2 Tracker Parser

The principal function of this class is to iterate over the a LCI datagram generated during an tracker electronics calibration. In order to do so, the class defines three virtual methods which must be satisfied by a derived class:

- `Open`, which is called once, *before* any events are parsed.
- `Process`, which is called for *each* event of the datagram and represents the work to be performed by the derived class with respect to that event.
- `Close`, which is called once , *after* all events of the datagram have been parsed.

In short, three user methods are triggered: once at the beginning of the datagram, once at its end, and once for every event in between. Each triggered method, is passing as an argument, a reference to the appropriate information for that method (see Section 3.4 and Section 7.3). Note that the parser processes events in the order in which they are stored in the datagram.

To initiate parsing the base class's `parse` method is invoked (see Section 2.4). This method takes as an argument a reference to the datagram to be parsed. Note, that the type identifier of

the datagram must correspond to an LCI datagram resulting from a tracker calibration. If not, an exception is declared.

The definition for this class is contained in Listing 24:

Listing 24 Class definition for `TkrParser`

```
 1:  class TkrParser : public DfiEvent::Parser {
 2:    public:
 3:      TkrParser(DfiEvent::Decompression);
 4:    public:
 5:      virtual ~TkrParser();
 6:    public:
 7:      virtual bool open(const DfiEvent::Context&)   = 0;
 8:      virtual bool process(const TkrMetaEvent&)     = 0;
 9:      virtual bool close(const DfiEvent::Context&) = 0;
10:    };
```

7.2.1 Constructor synopsis

`TkrParser` The constructor's argument is a enumeration specifying the maximum level of event data decompression supported by the parser. See Section 3.3 for information on how this argument is used. If the parser cannot support the specified decompression level the constructor will throw the exception specified in Section 3.12.1.

7.2.2 Member synopsis

`open` This method will be called *once* per datagram. It is called immediately *before* any events of the datagram are parsed (see below). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *opened*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`process` This method will be called for each event present in the parsed datagram. The argument is a reference to an object which provides a description of the event to be processed (see Section 7.3). Note, that the contents of this object reflect their value at the time of the corresponding event. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`close` This method will be called *once* per datagram. It is called immediately *after* all (if any) events of the datagram are parsed (see above). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *closed*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

7.3 Tracker Meta Event

A reference to an instance of this class is passed into the `process` method of the event parser (described in Section 7.2). This method is called by the parser for each event contained within a datagram. As the parser advances through a datagram's events the contents of the meta event are constantly changing to reflect their value for any, one, specified event. Note that this class it is derived from the common event calibration class `MetaEvent` (see Section 5.4). In order to go from a *meta* event to the event's *data*, requires instantiating an `Event` (see Section 5.4), passing as an argument an object of this class.

The definition for this class is contained in Listing 25:

Listing 25 Class definition for `TkrMetaEvent`

```
 1:  class TkrMetaEvent : public MetaEvent {
 2:    public:
 3:      TkrMetaEvent(const _QSE_ctx*);
 4:    public:
 5:     ~TkrMetaEvent();
 6:    public:
 7:      unsigned short injected()  const;
 8:      unsigned short delay()     const;
 9:      unsigned short threshold() const;
10:      unsigned short trigger     const;
11:      Channel        from        const;
12:    };
```

7.3.1 Constructor synopsis

`TkrMetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

7.3.2 Member synopsis

`injected` This function returns a value which defines the amount of charge which was injected for the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the amount of injected charge was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`delay` This function returns a value which defines the time delay between the injection of the charge and the TACK used to read out the corresponding calibration data. This value is specified in units of LAT clock tics, where one tic is nominally 50 nanoseconds. If the delay was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`threshold` This function returns a value which defines the charge threshold necessary to cross in order to generate the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the threshold was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`trigger` This function returns a value which (in conjunction with the value below) specified the discrimination threshold necessary to toggle the tracker's three-in-a-row signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`from` This method returns an object (see Section 5.6) which describes the channel(s) enabled in the corresponding calibration data. The interpretation of channel number is subsystem dependent. See Table 13 for the interpretation of a tracker channel. This function has no arguments and throws no exceptions.

Table 13 Type definition for the channel class when used in an tracker calibration.

Member	Interpretation	Range[1]	
		minimum	maximum
`single`	The channel number is represented in *layer* space. This channel is enabled in all layers, over all towers.	0	1535
`all`	The channel number is represented in *Front-End* space. This channel is enabled in all FEs, in all layers, over all towers.	0	63

1. In decimal.

Chapter 8
Calorimeter calibration support

8.1 Overview

The two classes described below are used to process the information returned from LCI when calibrating the calorimeter. These classes are based on the classes described in Chapter 5. Each time the calibration system (LCI) injects charge a corresponding event is generated. An event's information is encapsulated in the `CalMetaEvent` class (see Section 8.3). These calibration events are contained in a series of datagrams, sent down to the ground on the science stream (see [27]) by LCI. In order to process the events within any one datagram the user parses the datagram using the `CalParser` described in Section 8.2.

8.2 Calorimeter Parser

The principal function of this class is to iterate over the a LCI datagram generated during an calorimeter electronics calibration. In order to do so, the class defines three virtual methods which must be satisfied by a derived class:

- `Open`, which is called once, *before* any events are parsed.
- `Process`, which is called for *each* event of the datagram and represents the work to be performed by the derived class with respect to that event.
- `Close`, which is called once , *after* all events of the datagram have been parsed.

In short, three user methods are triggered: once at the beginning of the datagram, once at its end, and once for every event in between. Each triggered method, is passing as an argument, a reference to the appropriate information for that method (see Section 3.4 and Section 8.3). Note that the parser processes events in the order in which they are stored in the datagram.

To initiate parsing the base class's `parse` method is invoked (see Section 2.4). This method takes as an argument a reference to the datagram to be parsed. Note, that the type identifier of

the datagram must correspond to an LCI datagram resulting from an calorimeter calibration. If not, an exception is declared.

The definition for this class is contained in Listing 26:

Listing 26 Class definition for `CalParser`

```
 1:  class CalParser : public DfiEvent::Parser {
 2:    public:
 3:      CalParser(DfiEvent::Decompression);
 4:    public:
 5:      virtual ~CalParser();
 6:    public:
 7:      virtual bool open(const DfiEvent::Context&)   = 0;
 8:      virtual bool process(const CalMetaEvent&)     = 0;
 9:      virtual bool close(const DfiEvent::Context&) = 0;
10:    };
```

8.2.1 Constructor synopsis

`CalParser` The constructor's argument is a enumeration specifying the maximum level of event data decompression supported by the parser. See Section 3.3 for information on how this argument is used. If the parser cannot support the specified decompression level the constructor will throw the exception specified in Section 3.12.1.

8.2.2 Member synopsis

`open` This method will be called *once* per datagram. It is called immediately *before* any events of the datagram are parsed (see below). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *opened*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`process` This method will be called for each event present in the parsed datagram. The argument is a reference to an object which provides a description of the event to be processed (see Section 8.3). Note, that the contents of this object reflect their value at the time of the corresponding event. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

`close` This method will be called *once* per datagram. It is called immediately *after* all (if any) events of the datagram are parsed (see above). The argument is a reference to an object which contains the datagram context (see Section 3.4). Note, that the contents of this object reflect their value at the time of the datagram was *closed*. This function returns a boolean specifying whether or not to abort parsing. If the function returns `TRUE`, parsing *continues*. If the function returns `FALSE`, parsing *aborts*. When the parser aborts, control is returned to the caller of the `parse` method. Note this function is pure virtual and, therefore, its implementation must be provided by a derived class. The function throws no exceptions.

8.3 Calorimeter Meta Event

A reference to an instance of this class is passed into the `process` method of the event parser (described in Section 8.2). This method is called by the parser for each event contained within a datagram. As the parser advances through a datagram's events the contents of the meta event are constantly changing to reflect their value for any, one, specified event. Note that this class it is derived from the common event calibration class `MetaEvent` (see Section 5.4). In order to go from a *meta* event to the event's *data*, requires instantiating an `Event` (see Section 5.4), passing as an argument an object of this class.

The definition for this class is contained in Listing 27:

Listing 27 Class definition for `CalMetaEvent`

```
 1:  class CalMetaEvent : public MetaEvent {
 2:    public:
 3:      CalMetaEvent(const _QSE_ctx*);
 4:    public:
 5:     ~CalMetaEvent();
 6:    public:
 7:      unsigned short uld()       const;
 8:      unsigned short injected()  const;
 9:      unsigned short delay()     const;
10:      unsigned short threshold() const;
11:      CalTrigger     trigger     const;
12:      Channel        from        const;
13:    };
```

8.3.1 Constructor synopsis

`CalMetaEvent` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

8.3.2 Member synopsis

`uld` This function returns a value which defines the threshold necessary to cross range boundaries in the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the threshold was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`injected` This function returns a value which defines the amount of charge which was injected for the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the amount of injected charge was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`delay` This function returns a value which defines the time delay between the injection of the charge and the TACK used to read out the corresponding calibration data. This value is specified in units of LAT clock tics, where one tic is nominally 50 nanoseconds. If the delay was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`threshold` This function returns a value which defines the charge threshold necessary to cross in order to generate the specified calibration data. This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the threshold was determined from the LATC database the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`trigger` This method returns an object (see Section 8.4) which specifies the thresholds for the calorimeter's signals which are sent to the trigger system. This function has no arguments and throws no exceptions.

`from` This method returns an object (see Section 5.6) which describes the channel(s) enabled in the corresponding calibration data. The interpretation of channel number is subsystem dependent. See Table 14 for the interpretation of an calorimeter channel. This function has no arguments and throws no exceptions.

Table 14 Type definition for the channel class when used in a calorimeter calibration.

Member	Interpretation	Range[1]	
		minimum	maximum
`single`	The channel number is represented in *channel* space. This channel is enabled in all layers, over all towers.	0	11
`all`	All channels in all FEs are enabled, in all layers, over all towers.	N/A	N/A

1. In decimal.

8.4 Calorimeter Trigger Discriminators

An instance of this class is returned from the calorimeter's meta event (see Section 8.3). This class specifies the values of the discriminators which specify the threshold for the production of the trigger signals generated by the calorimeter and used by the trigger system

The definition for this class is contained in Listing 28:

Listing 28 Class definition for `CalTrigger`

```
1:  class CalTrigger {
2:    public:
3:      CalTrigger(const _QSE_ctx*);
4:    public:
5:     ~CalTrigger();
6:    public:
7:      unsigned short le() const;
8:      unsigned short he() const;
9:    };
```

8.4.1 Constructor synopsis

`CalTrigger` From the user's viewpoint the argument to this constructor is irrelevant and should be ignored. However, its definition is given here for completeness. The argument is a reference to a structure which allows the class to locate within the datagram, the meta-data necessary to construct an object of this class. The constructor throws no exceptions.

8.4.2 Member synopsis

`le` This function returns a value which specifies the discrimination threshold necessary to toggle the calorimeter's *Low Energy* trigger signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

`he` This function returns a value which specifies the discrimination threshold necessary to toggle the calorimeter's *High Energy* trigger signal sent to the GEM (see [22]). This value is specified in units of DAC counts. The relationship between DAC counts and charge is subsystem dependent. If the discriminated value was determined from the LATC database, the constant `UNDEFINED` (see Section 5.3) is returned. This function has no arguments and throws no exceptions.

8.5 Exceptions

8.5.1 Decompression Failed

to be written.