LCIO - Users manual

v2-14

F. Gaede, DESY IT
H. Vogt, DESY Zeuthen

July 2, 2020

Contents

1 INTRODUCTION

Note: Some of the latest developments in LCIO are not yet described in this manual. However it is still a good starting point if you are new to LCIO as it explains the basic concepts to quite some detail. For some of the newer developments and the current versions please refer to https://github.com/iLCSoft/LCIO.

LCIO is a persistency framework that defines a data model for the linear collider physics community. It is intended to be used in both simulation studies and analysis frameworks. Its light weight and portability makes it also suitable for use in detector R&D applications.
It provides a C++ and a Java implementation with a common interface (API). A Fortran interface to the C++ implementation is provided as well.
This manual is intended for application developers that want to incorporate LCIO in their programs. This includes e.g. simulation developers as well as physicists that want to read LCIO files for their analysis. It focuses on the practical aspects of using LCIO. A more general discussion can be found in [1] and other documents listed on the LCIO homepage [2].

2 Installation

2.1 Getting LCIO

If LCIO is not yet installed at your site you can get a recent copy from the GitHub repository as follows *bash*:

```
git clone https://github.com/iLCSoft/LCIO.git
git checkout v02-14    ##  use a specific version
```

2.2 Requirements

In order to build LCIO you need to have a Java VM (version 1.4 or higher) installed on your platform. This is true even if you only want to install the C++ version as the API-files are generated from an abstract description for Java and C++ using the AID [4] tool. The build scripts used in LCIO also require that `gcc` and `gmake/make` are installed.
The C++ version of LCIO is developed and tested under Linux (SL4 and SL5 32 and 64bit) and also builds under Windows/Cygwin or MacOSX.
As the -ansi compiler flag is used it should be fairly easy to port it to other platforms with an ANSI compliant C++ compiler.

2.3 Building LCIO

There are two build tools that can be used to build LCIO: maven and CMake [12].
CMake [12] has been introduced in version v01-11 to facilitate cross plattform support for the C++ version.

2.3.1 Building LCIO with CMake

If you want to use CMake for building LCIO you need at least CMake version 2.6.
If you want to build LCIO using CMake, make sure that the cmake command is in your path, e.g.

```
export PATH=Path_to_CMake/bin:$PATH
```

Then issue the following commands:

```
mkdir build           # create a directory for the build
cd build
cmake ..              # check build options ( change options with: cmake -DOPTION=ON|OFF .. )
                      #  e.g. to also build the Java version, use: cmake -DINSTALL_JAR=ON ..
                      #  optionally use: ccmake .. and configure interactively
make install
```

CMake options may also be specified in a separate file and read with the -C option. (please check the CMake [12] documentation for more details)

```
# ILCSoft.cmake is a configuration file generated for each ilcsoft reference installation.
# It contains the list of paths from all tools within the installation.
# in this example the BUILD_ROOTDICT option from LCIO is activated which requires the
# path to the ROOT package. This path is read from the ILCSoft.cmake configuration file.

cmake -C /afs/desy.de/project/ilcsoft/sw/i386_gcc41_sl5/v01-11/ILCSoft.cmake \
      -DBUILD_ROOTDICT=ON ..
```

2.3.2 Building LCIO with Maven

Download and install Maven (http://maven.apache.org/download.html) and then type

```
mvn clean install
```

Both ways of building LCIO will create the following libraries and executables:

```
./lib
     lcio.jar       <-- Java library (and executables)
     liblcio.so     <-- C++ lcio library
     libsio.so      <-- C++ sio library

./bin               <-- C++ examples and tools
     anajob
     copyfix
     dumpevent
     readcalibration
     recjob
     simjob
     lcio_event_counter
     stdhepjob

./bin               <-- f77 examples
     anajob_F
     simjob_F
     recjob_F
```

2.4 Checking the Installation

To check whether the (C++) installation was successful run the *simjob* program:

```
simjob simjob.slcio
```

This creates a simple LCIO file that you can read with

```
anajob simjob.slcio
```

The same for the Java - Linux version:

```
runSimJob.sh simjob.sclio
runAnalysisJob.sh simjob.sclio
```

or for the Java Windows (Cygwin) version:

```
runSimJob[.bat] simjob.sclio
runAnalysisJob[.bat] simjob.slcio
```

In case CMake has been used to build LCIO it is possible to run a small chain of tests by calling:

```
make tests && make test
```

2.5 Building the documentation

A few targets are defined to build the documentation (see below), that is also available from the LCIO homepage [2], see (3.1).

If you are using CMake (see 2.3) all documentation can be built and installed by setting the cmake option `INSTALL_DOC` to ON as follows:

```
cmake -DINSTALL_DOC=ON ..
make install
```

In order to build individual documentation components the following targets may be used:

```
make doc             <-- build all documentation (doc_cpp + doc_java + doc_manual)
make doc_cpp         <-- C++ API documentation (needs {\em doxygen}~\cite{ref_doxygen})
make doc_java        <-- Java API documentation
make doc_refman      <-- Reference manual
make doc_usrman      <-- User manual
```

If you are not using CMake keep on reading the rest of this section on how to build the documentation with ant:

You can create the documentation of the Java API with *javadoc* [5]:

```
ant doc
```

and then open the file `/doc/api/index.html` with your Browser.

And the current version of this manual is created with

```
ant doc.manual
```

as `/doc/manual.pdf` and `/doc/manual.ps` provided you have latex in your `$PATH`

3 Using LCIO

This section gives an introduction on how to use LCIO. We describe the user interface, the data model and provide some code examples. For most of the section we focus on Java and C++ as these are the main languages supported by LCIO. Thanks to the AID [4] tool Java and C++ hava an API which is generated from a common source and thus very similar. The Fortran interface, implemented as a wrapper to the C++ implementation, is described in 3.9.

3.1 Java and C++ API

Detailed documentation of the API is provided both for Java and C++ on the LCIO homepage [2] generated directly from documentation in the source code using *javadoc* and *doxygen* respectively. If you are experienced in either Java or C++ you will probably find most of what you need to use LCIO in the corresponding version of the API documentation. However, reading the detailed description of the data entities in [3.2] is probably usefull.
A few words on the design of LCIO might be helpful to browse the documentation. LCIO is organized in a hierarchical package structure where the packages combine classes with a well defined purpose as shown in table 1. So a good starting point to browse the API documentation is the package/namespace

C++ Namespace	Java package	Purpose
EVENT	hep.lcio.event	The base interfaces of all data classes in LCIO. To be used for reading existing data.
IO	hep.lcio.io	The base interfaces for io of data.
IMPL	hep.lcio.implementation.event	The (default) implementations of the base interfaces that are defined in EVENT. Needed for writing new data or extending existing data, e.g. during reconstruction.
UTIL	hep.lcio.util	Holds convenient and support classes and methods for the LCIO objects.
lcio	n.a.	The namespace lcio combines EVENT, IO, UTIL and IMPL for user convenience (#include "lcio.h").
Namespaces/packages below are shown for completeness only, don't use in your code!		
IOIMPL	hep.lcio.implementation.io	Extensions to the default implementations needed for IO. With the exception of LCFactory all other classes are for internal use only.
SIO	hep.lcio.implementation.sio	The persistency implementation using SIO. Users should not use any of the classes defined here explicitly but through their base interfaces defined in IO.
(DATA)	*(hep.lcio.data)*	Removed in v1.3

Table 1: *Overview of the packages (namespaces) used in LCIO.*

overview. Depending on the use case you will need mostly classes from one or two namespaces, e.g. if you need to read data from an existing LCIO file, you only need the classes in EVENT. If you want to write data with LCIO you have to instantiate the implementation classes provided in IMPL (or implement the interface from EVENT).

3.1.1 Exceptions

Both the Java and C++ versions of LCIO make use of the exception mechanism. Exceptions are implemented differently in Java and C++[1]. For LCIO we adopted a scheme of using exceptions that hides these differences as much as possible. All Exceptions except `IOException` in Java inherit from `RunTimeException`, i.e. are *unchecked* exceptions. Thus the compiler will in general not complain about missing catch blocks or throw declarations. The same holds true for the C++ compiler anyhow.

[1]The Java Exception mechanism is more powerful in that it offers compile time check of correct *try-catch* blocks or *throw* clauses and the notion of *unchecked* and *checked* exceptions.

In the API you will find *throw* clauses/declarations at certain methods. These are meant as a hint to the user of what could go wrong in the method at hand - do not try to catch all exceptions with a dedicated *try-catch* block around every function call. Most exceptions will be due to *programming errors* and will cause an abort of the program – when not caught by the user – which is usually what you want.
There are exceptions to this rule however. In particular the `DataNotAvailableException` will be thrown by methods accessing event data that are not available, e.g. if you are trying to get a named collection from the event that is not available. In this case it depends on your application whether you want to continue or not - if the computation is vital for the rest of the program you'll probably want to abort - if the code just produces some check plots for a particular subdetector you might as well carry on with other 7modules.
In case an exception is thrown at run time that is not caught the program will abort and you will get some printout as to what caused the exception - in Java this is done by the VM in C++ we print the type of the exception with some information on the problem (usually the class and function name).

Note to C++ programmers: *We included* `std::exception` *in all* throw *clauses in C++. This way we ensure a behavior similar to Java, i.e. despite a* throw *clause basically all exceptions can be thrown by the function, as all STL-Exceptions as well as all LCIO-Exceptions inherit from the base* `std::exception`*. So the user can decide to catch* run time *exceptions for parts of the code in order to prevent an abort of the program.*

3.2 Data model

Figure 1 shows the event data model of LCIO. The most important class is LCEvent, it serves as a container for all data that are related to one event. It holds an arbitrary number of named collections (class LCCollection) of data objects (class LCObject). Run related information is stored in LCRun-Header. Run-headers, events and collections have an instance of LCParameters that allows to attach arbitrary named parameters of types *int, float* or *string* to either of these objects. This mechanism is foreseen to allow storage of meta data, e.g. the name of the algorithm that has been used to create a certain collection. The LCCollection has a type word that is used as a flag word, where certain bits are used by LCIO to denote some well defined feature of the objects in the collection. Typically these flag bit positions are defined as global constants in the class LCIO.
The following gives a detailed description of the data entities that are defined in LCIO. An overview of these classes is shown in figure 2.

3.2.1 MCParticle

There will be exactly one collection with name "MCParticle" in every event that holds the Monte Carlo truth particles as generated by the generator or as added by the simulation program (decay in flight). Particles that are created during simulation will be added to the existing list of MCParticles. Adding particles with their correct lineage ceases when a particle decays or interacts in a non-tracking region. (Otherwise the number of MCParticles explodes in calorimeter shower development). All energy depositions are assigned to the initial particle that decayed or interacted in a non-tracking region. The exception is a particle producing a hit in a tracker. This particle is recorded, with the tracker hit assigned to it, and the particle is flagged as a "backscatter". Thus direct tracker hits for a particular particle can be distinguished from those not actually produced by that particle.
The *generatorStatus* and *simulatorStatus* attributes define the creation and destruction status of the particle, where the *generatorStatus* is defined by the generator program and the *simulatorStatus* is combined from the following boolean flags:

- **isCreatedInSimulation**
  True if Simulator created particle. False if Generator created particle.

- **isBackscatter**
  True if particle created by Simulator as a result of interaction or decay in non-tracking region. By convention, such particles are not saved. However, if this particle creates a tracker hit, the

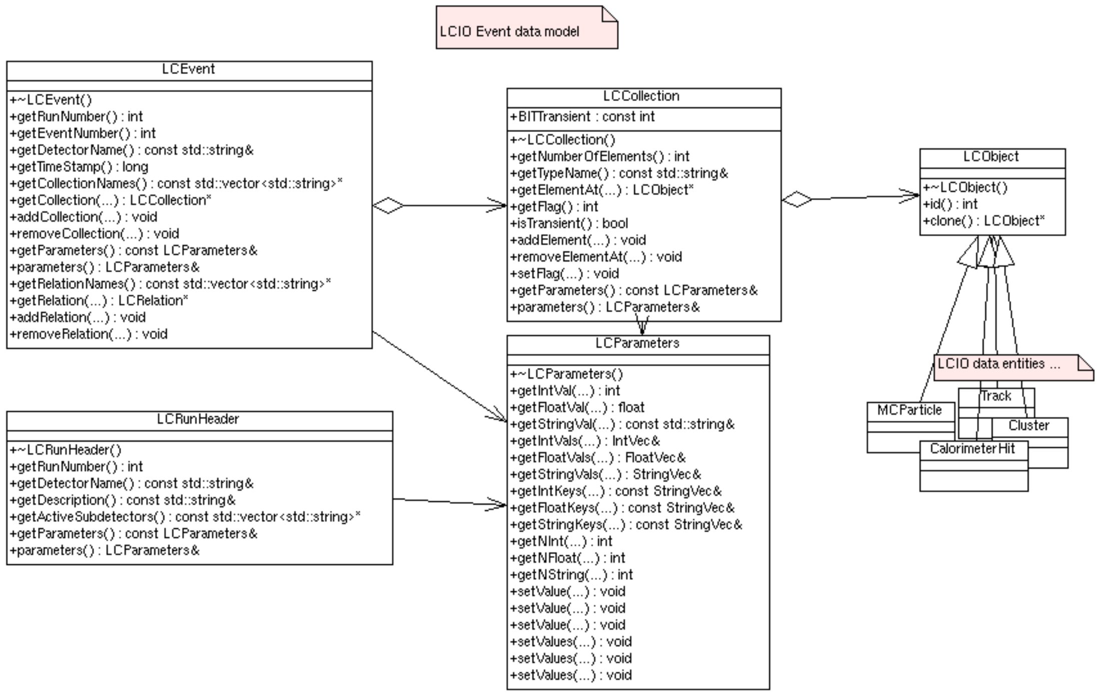
LCIO Event data model
LCEvent
+~LCEvent()
+getRunNumber() : int
+getEventNumber() : int
+getDetectorName() : const std::string&
+getTimeStamp() : long
+getCollectionNames() : const std::vector<std::string>*
+getCollection(...) : LCCollection*
+addCollection(...) : void
+removeCollection(...) : void
+getParameters() : const LCParameters&
+parameters() : LCParameters&
+getRelationNames() : const std::vector<std::string>*
+getRelation(...) : LCRelation*
+addRelation(...) : void
+removeRelation(...) : void
LCCollection
+BITTransient : const int
+~LCCollection()
+getNumberOfElements() : int
+getTypeName() : const std::string&
+getElementAt(...) : LCObject*
+getFlag() : int
+isTransient() : bool
+addElement(...) : void
+removeElementAt(...) : void
+setFlag(...) : void
+getParameters() : const LCParameters&
+parameters() : LCParameters&
LCObject
+~LCObject()
+id() : int
+clone() : LCObject*
LCIO data entities ...
MCParticle
Track
Cluster
CalorimeterHit
LCRunHeader
+~LCRunHeader()
+getRunNumber() : int
+getDetectorName() : const std::string&
+getDescription() : const std::string&
+getActiveSubdetectors() : const std::vector<std::string>*
+getParameters() : const LCParameters&
+parameters() : LCParameters&
LCParameters
+~LCParameters()
+getIntVal(...) : int
+getFloatVal(...) : float
+getStringVal(...) : const std::string&
+getIntVals(...) : IntVec&
+getFloatVals(...) : FloatVec&
+getStringVals(...) : StringVec&
+getIntKeys(...) : const StringVec&
+getFloatKeys(...) : const StringVec&
+getStringKeys(...) : const StringVec&
+getNInt(...) : int
+getNFloat(...) : int
+getNString(...) : int
+setValue(...) : void
+setValue(...) : void
+setValue(...) : void
+setValues(...) : void
+setValues(...) : void
+setValues(...) : void

Figure 1: *Overview of the event data model defined by LCIO. The boxes correspond to classes and the arrows denote relationships between these. The main class in LCIO is LCEvent that serves as a container of named collections (LCCollection) of data objects (LCObject). All data that are related to one event is stored in such collections (A few example classes are shown, see 2 for details on the data entities.) The LCRunHeader is used to store run related information. The class LCParameters can be used to attach keyed parameters of type int, float, string to either LCRunHeader, LCEvent or LCCollection in order to store so called meta data, e.g. the encoding of certain bits in some type word, or the order of some parameters in a given array.*

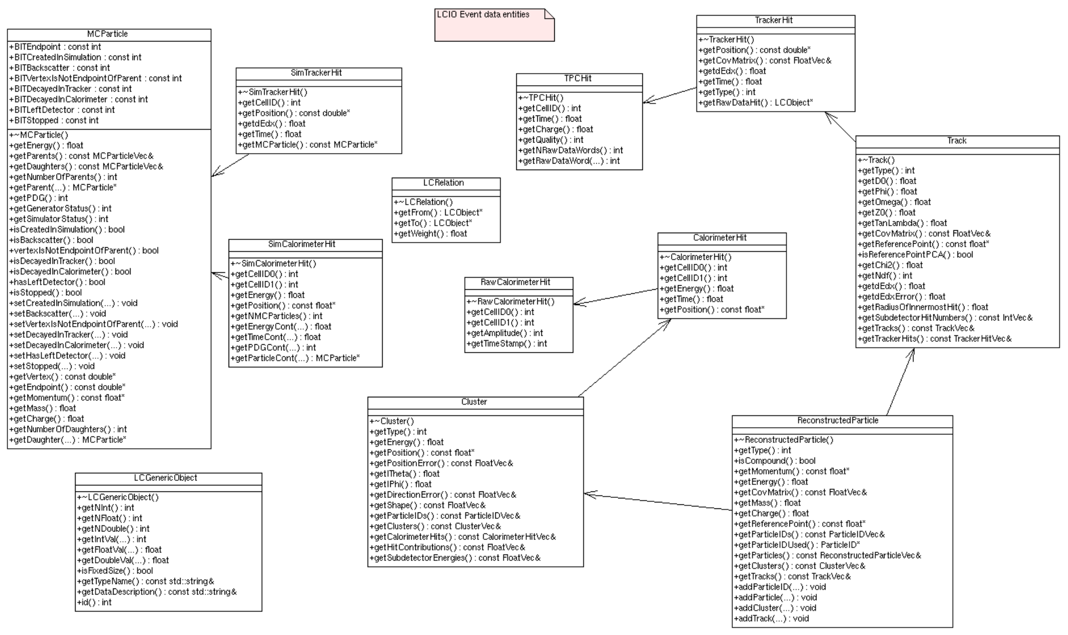
LCIO Event data entities
MCParticle
+BITEndpoint : const int
+BITCreatedInSimulation : const int
+BITBackscatter : const int
+BITVertexIsNotEndpointOfParent : const int
+BITDecayedInTracker : const int
+BITDecayedInCalorimeter : const int
+BITLeftDetector : const int
+BITStopped : const int
+~MCParticle()
+getEnergy() : float
+getParents() : const MCParticleVec&
+getDaughters() : const MCParticleVec&
+getNumberOfParents() : int
+getParent(...) : MCParticle*
+getPDG() : int
+getGeneratorStatus() : int
+getSimulatorStatus() : int
+isCreatedInSimulation() : bool
+isBackscatter() : bool
+vertexIsNotEndpointOfParent() : bool
+isDecayedInTracker() : bool
+isDecayedInCalorimeter() : bool
+hasLeftDetector() : bool
+isStopped() : bool
+setCreatedInSimulation(...) : void
+setBackscatter(...) : void
+setVertexIsNotEndpointOfParent(...) : void
+setDecayedInTracker(...) : void
+setDecayedInCalorimeter(...) : void
+setHasLeftDetector(...) : void
+setStopped(...) : void
+getVertex() : const double*
+getEndpoint() : const double*
+getMomentum() : const float*
+getMass() : float
+getCharge() : float
+getNumberOfDaughters() : int
+getDaughter(...) : MCParticle*
SimTrackerHit
+~SimTrackerHit()
+getCellID() : int
+getPosition() : const double*
+getdEdx() : float
+getTime() : float
+getMCParticle() : const MCParticle*
TPCHit
+~TPCHit()
+getCellID() : int
+getTime() : float
+getCharge() : float
+getQuality() : int
+getNRawDataWords() : int
+getRawDataWord(...) : int
TrackerHit
+~TrackerHit()
+getPosition() : const double*
+getCovMatrix() : const FloatVec&
+getdEdx() : float
+getTime() : float
+getType() : int
+getRawDataHit() : LCObject*
LCRelation
+~LCRelation()
+getFrom() : LCObject*
+getTo() : LCObject*
+getWeight() : float
Track
+~Track()
+getType() : int
+getD0() : float
+getPhi() : float
+getOmega() : float
+getZ0() : float
+getTanLambda() : float
+getCovMatrix() : const FloatVec&
+getReferencePoint() : const float*
+isReferencePointPCA() : bool
+getChi2() : float
+getNdf() : int
+getdEdx() : float
+getdEdxError() : float
+getRadiusOfInnermostHit() : float
+getSubdetectorHitNumbers() : const IntVec&
+getTracks() : const TrackVec&
+getTrackerHits() : const TrackerHitVec&
SimCalorimeterHit
+~SimCalorimeterHit()
+getCellID0() : int
+getCellID1() : int
+getEnergy() : float
+getPosition() : const float*
+getNMCParticles() : int
+getEnergyCont(...) : float
+getTimeCont(...) : float
+getPDGCont(...) : int
+getParticleCont(...) : MCParticle*
CalorimeterHit
+~CalorimeterHit()
+getCellID0() : int
+getCellID1() : int
+getEnergy() : float
+getTime() : float
+getPosition() : const float*
RawCalorimeterHit
+~RawCalorimeterHit()
+getCellID0() : int
+getCellID1() : int
+getAmplitude() : int
+getTimeStamp() : int
Cluster
+~Cluster()
+getType() : int
+getEnergy() : float
+getPosition() : const float*
+getPositionError() : const FloatVec&
+getITheta() : float
+getIPhi() : float
+getDirectionError() : const FloatVec&
+getShape() : const FloatVec&
+getParticleIDs() : const ParticleIDVec&
+getClusters() : const ClusterVec&
+getCalorimeterHits() : const CalorimeterHitVec&
+getHitContributions() : const FloatVec&
+getSubdetectorEnergies() : const FloatVec&
ReconstructedParticle
+~ReconstructedParticle()
+getType() : int
+isCompound() : bool
+getMomentum() : const float*
+getEnergy() : float
+getCovMatrix() : const FloatVec&
+getMass() : float
+getCharge() : float
+getReferencePoint() : const float*
+getParticleIDs() : const ParticleIDVec&
+getParticleIDUsed() : ParticleID*
+getParticles() : const ReconstructedParticleVec&
+getClusters() : const ClusterVec&
+getTracks() : const TrackVec&
+addParticleID(...) : void
+addParticle(...) : void
+addCluster(...) : void
+addTrack(...) : void
LCGenericObject
+~LCGenericObject()
+getNInt() : int
+getNFloat() : int
+getNDouble() : int
+getIntVal(...) : int
+getFloatVal(...) : float
+getDoubleVal(...) : float
+isFixedSize() : bool
+getTypeName() : const std::string&
+getDataDescription() : const std::string&
+id() : int

Figure 2: *Overview of the data entities defined by LCIO (namespace EVENT). All classes are subclasses of LCObject. The shown dependencies correspond to links between objects, e.g. for a given reconstructed particle you can get the tracks and clusters that have been used to create the particle. There is no direct link from 'real data' objects to Monte-Carlo objects, instead LCRelation is to be used for that.*

particle is added to the MCParticle list with this flag set, and the parent set to the particle that initially decayed or interacted in a non-tracking region

- **vertexIsNotEndpointOfParent**
  True if particle created as a result of a continuous process where the parent particle continues, i.e. hard ionization, Bremsstrahlung, elastic interactions, etc.
- **isDecayedInTracker**
  True if the particle decayed or interacted (non-continuous interaction, particle terminated) in a tracking region.
- **isDecayedInCalorimeter**
  True if tracking of particle stopped because particle left the world volume.
- **hasLeftDetector**
  True if tracking of particle left the world volume.
- **isStopped**
  True if particle lost all kinetic energy inside world volume and did not decay.

For the *MCParticle* only parent relationships are stored. When reading the data back from the file the daughter relationships are reconstructed from the parents. This is to ensure consistency. Care has to be taken when analyzing the particle 'tree'. Because a particle can have more than one parent the particle list in fact does not consist of a set of trees (one for each mother particle) but forms a 'directed acyclic graph'. Thus the user has to avoid double counting in his code. Of course this only matters at the parton level, as real particles have exactly one or no parent.

3.2.2 SimCalorimeterHit

Generic class for calorimeter hits from a simulation program. Typically every calorimeter subdetector will create a collection of this type. SimCalorimeterHits have a float energy amplitude and an int cellid, and optionally a 3d hit/cell position. In order to study clustering and reconstruction algorithms there are links to the MCParticles that contributed to the hit. There are two different levels of detail foreseen(check/set the flag LCIO.CHBIT_STEP):

- LCIO.CHBIT_STEP=0, normal mode: one MCParticle contribution for every MCParticle that contributed to the hit, i.e. there will be only one MCParticle contribution except for cells where showers from two particles overlap.
- LCIO.CHBIT_STEP=1, detailed mode: one contribution for every simulator step that contributed to the hit. Every contribution has the energy, time and PDG of the shower particle that caused the energy deposit. This mode will result in a lot of entries per calorimeter hit and is intended for detailed studies only.

In general the position of calorimeter hits will be decoded from the cellid to save disk space. For user convenience it is however possible to store the three dimensional position of the cell with every hit. Check/set the collection flag bit LCIO.CHBIT_LONG=1. Also for cases where it is impossible or inconvenient to decode the cellid in one word, SimCalorimeterHit can have an optional second cellid (LCIO.CHBIT_ID1=1).

3.2.3 SimTrackerHit

This is the generic class for storing simulated hits from tracking subdetectors. It records the space-point, energy and time of the hit as well as a link to the MCParticle that caused it.

3.2.4 RawCalorimeterHit

This class is intended to be used for real calorimeter data, e.g. from test beam calorimeters. It stores a cellid, amplitude and optionally (LCIO.RCHBIT_TIME=1) a time word, all as int values as this is what is read out from the DAQ system. Also a second cellid can be utilized if needed (LCIO.RCHBIT_ID1=1). To save disk space when storing raw data only, set the colelction flag bit LCIO.RCHBIT_NO_PTR=1, suppressing an additional 32-bit pointer tag per hit.

3.2.5 CalorimeterHit

The CalorimeterHit should be used for final reference to the calorimeter hits, i.e. after calibration, clustering etc. It has a float energy and optional time word. Clusters will have links to objects of type CalorimeterHit.

3.2.6 TPCHit

Deprecated. Please use the new classes for raw tracker data: TrackerRawData, TrackerData and TrackerPulse.

3.3 TrackerRawData

Generic class for raw tracker data. One or optionally two (LCIO.TRAWBIT_ID1=1) cell ids can be used to decode the readout channel. Specific subdetectors will use different encodings, e.g. a silicon vertex detector might use cellID0 and cellID1 to encode the module number, row and column of the hit. An arbitrary number of ADC values can be stored in the hit. This can be either a complete spectrum (waveform) as for a TPC like detector or just one or two consecutive ADC readout values.

3.4 TrackerData

Same as the TrackerRawData, except that the data has been corrected (calibrated). In particular the calibrated ADC values are stored as floating point charge values with arbitrary, i.e. subdetector specific units.

3.5 TrackerPulse

Tracker pulse that is typically created from TrackerData by some sort of feature extraction, e.g. for a TPC like detector the pulse will contain the integrated charge of the spectrum and the corresponding time.

3.5.1 TrackerHit

The TrackerHit serves as a generic hit type that can be used for pattern recognition. It allready has a space point position and error matrix, computed from the raw data. TrackerHits point back to the raw data hits that they have been created from.

3.5.2 Cluster

Clusters are made from sets of CalorimeterHits that they point back to. Clusters can also be combined from other Clusters allowing a tree-like structure, e.g. one could build clusters with a geometrical algorithm and then combine some of these clusters to 'particles' applying some track-match criterieon. Due to the imaging capabilities of the LC calorimeters, clusters have an intrinsic direction, denoted ITheta, IPhi. For fast Monte Carlo or in case where the hits are droped from the files (DST) Clusters can have vector of subdetector energies. Check/set the collection variable ClusterSubdetectorNames for the decoding of the indices in the vector. And similarly check/set the collection variable ClusterShapeParameters for the indices in the shape parameter vector. Clusters have a type word where the lower 15 bits can be used to decode the type of the cluster by the user, e.g. define a bit per subdetector. Check/Set the collection variables ClusterTypeBitNames and ClusterTypeBitIndices.

3.5.3 Track

For LCIO we use the following parameters for Track:

- **d0**
  Impact paramter of the track in the r-phi plane.
- **phi**
  Phi of the track at the reference point.
- **omega**
  The signed curvature of the track in [1/mm] - the sign is that of the particle's charge.
- **z0**
  Impact paramter of the track the in the r-z plane.
- **tanLambda**
  Lambda is the dip angle of the track in the r-z plane at the reference point.

By default LCIO tracks have the point of closest approach (PCA) as the reference point. But any other point can be chosen at will, check `isReferencePointPCA()`. A full covariance matrix is stored with every track as the lower triangel matrix of the parameters in the given order.
Track points back to the hits that have been used in the fit. If the hits are not available there is a vector of subdetector hit numbers, check/set the collection parameter TrackSubdetectorNames to decode the indices. Also Tracks can point to other Tracks (typically track segments from other subdetectors) that have been combined to the track at hand.
Tracks have a type word where the lower 15 bits can be used to decode the type of the track by the user, e.g. define a bit per subdetector. Check/Set the collection variables TrackTypeBitNames and TrackTypeBitIndices.

3.5.4 ReconstructedParticle

ReconstructedParticle is the class to be used for every object that is reconstructed. ReconstructedParticle can be a single particle, e.g. a track with a pion PID, or a compound object like a jet made from many particles. ReconstructedParticle has lists of Tracks, Clusters and ReconstructedParticles that have been combined to form this particle. The particle type is encoded in a type word - check/set the collection parameters ReconstructedParticleTypeNames and ReconstructedParticleTypeValues. ReconstructedParticles have a 4-momentum and for convenience a redundant mass field. In general users will want to fill these consistently. Typically kinematics is calculated from the most likely PID assigned to this particle. ReconstructedParticle also points nack to CLusters and Tracks that have been used to create the particle.
Users have to make sure that they don't double count when looping over lists of ReconstructedParticles. LCIO might provide a convenient method to help looping over lists in a future release.
Reconstruction programs are expected to provide one reasonable list of ReconstructedParticles in the event that serves as a starting point for physics analysis. The corresponding LCCollection should be flagged as *default* and has to fulfill the following requirements:

- **unambigous:** elements in the list can be combined to compound particles like K_s but no particle can be used in more than one compound. Additional particle hypotheses are supposed to be stored in another list.

- **complete:** the list should be reasonably complete, i.e. everything seen in the event should have been taken into account to create that list.

3.5.5 LCGenericObject

Users are free to store additional data in LCIO files that cannot be expressed in terms of the classes that are provided with LCIO. To do so it is easiest to implement the interface LCGenericObject in a user defined concrete class. In order to write data, simply create instances of this class and add them to an LCCollection that in turn is added to the LCEvent. When reading back this information users are free to instantiate objects of the user class' type via copy constructor or simply access the information through the LCGenericObject interface.
For C++ a template for LCGenericObjects with fixed size: `LCFixedObject<int nint,int nfloat, int ndouble>` is provided in the UTIL namespace. Users have to provide minimal code by inheriting from this class.

3.5.6 LCIntVec, LCFloatVec and LCStringVec

Simple user extension data can be stored in LCCollections of vectors of type of *float, int* and *string.*
The LCIntVec can also be used to store indices of objects in another collection thus building subcollections. For example one can have a lepton finder that returns lepton candidates as a list of indices that point into the LCCollection of type Track that was given to finder.

3.5.7 LCRelation

As described above, some LCIO classes have built-in links/relations to other classes, e.g. tracks point back to hits. LCIO intentionally does not define such links to point back from reconstructed objects to Monte Carlo truth information, in order to allow a clean development of tools that could run on 'real data'.
Using the LCRelation one is still able to make such relations for simulation data. In general LCRelation can be used to store a weighted many to many relationship between objects of two arbitrary (but fixed) type. LCRelations are stored as any other data entity in an LCCollection. For user convenience a LCRelationNavigator is provided in *UTIL/hep.lcio.util* that allows to quickly navigate the relationships.
The type of the from- and to-object are stored in the collection parameters RelationFromType and RelationToType.

3.6 Data format

As a first concrete data format for LCIO we chose to use SIO (Serial Input Output) [3]. SIO has been developed at SLAC and has been used successfully in the *hep.lcd* framework. It is a serial data format that is based on XDR and thus machine independent. While being a sequential format it still offers some OO-features, in particular it allows to store and retrieve references and pointers within one record. In addition SIO includes on the fly data compression using zlib. LCIO files that use SIO, i.e. all current ones, have the extension *.slcio*.
A detailed description of the data layout in the SIO files is given in `$LCIO\doc\lcio.xml`

3.7 How to read LCIO files

There are a number of examples in the *src* directory – some examples for reading and accessing data can be found in:

```
src/cpp/src/EXAMPLE
  anajob.cc
  dumpevent.cc
  recjob.cc
  readcalibration.cc
src/cpp/src/IMPL
  LCTOOLS.cc

src/java/hep/lcio/example
  AnalysisJob.java
  RecJob.java
  LCTools.java
```

3.7.1 File handling

Before you can read from an LCIO file you have to create an instance of LCReader using LCFactory:

```
LCReader* lcReader = LCFactory::getInstance()->createLCReader() ;
```

or in Java:

```
LCReader lcReader = LCFactory.getInstance().createLCReader();
```

The factory pattern is used to hide the concrete implementation of the data format (see 3.6) from user code. Now opening and closing a file is as simple as:

```
lcReader->open( "my_data.slcio" ) ;

// ... here we can read sth. from the file

lcReader->close() ;
```

As described in the section on exceptions (3.1.1) you could enclose the above code or parts of it in a *try-catch* block, e.g.

```
try{

  lcReader->open( "my_data.slcio" ) ;

  // ...

  lcReader->close() ;
}
```

```
  catch(IOException& e){

    cout << " Unable to read and analyze the LCIO file - " << e.what() << endl ;
  }
```

if you want to do anything else in your application after what would have been done in " `//...` " failed.
Here is the corresponding Java code:

```
  lcReader.open( "my_data.slcio" ) ;

  // ... do sth. ....

  lcReader.close() ;
```

It is obviously straightforward to get the Java version of the code from the C++ one so we only give example code in C++ from now on.

3.7.2 Reading from the file

There are two different ways of reading data from an LCIO file. One is via 'readNext*Something*()' methods, where *Something* is either the next RunHeader or the next Event. This way the user has control over what is being read from the file. On the other hand you can only read one type of *records* at a time as you don't know the exact number of events for each run. A simple event loop looks like this:

```
    LCEvent*  evt;
    while( (evt = lcReader->readNextEvent()) != 0 ) {

      LCTOOLS::dumpEvent( evt ) ;

      nEvents ++ ;
    }
```

For small applications and data files this is a reasonable way of analyzing data.
The other way of reading LCIO data files is via a *Listener* (or *Observer*) pattern. In this approach you have to write your own analysis module(s) which implements the *LCRunListener* and *LCEventListener* interface and register those with the *LCReader*. When reading the data stream the corresponding registered modules are called depending on the type of the current record.
By writing modules for distinct tasks, e.g. vertex track reconstruction, track finding, clustering algorithms etc. this already defines an application framework for reconstruction with the LCEvent as underlying data structure.
For example you can define one analysis module as run and event listener:

```
// class for processing run and event records
class MyAnalysis : public LCRunListener, public LCEventListener{

public:
   MyAnalysis() ;
  ~MyAnalysis() ;

  void processEvent( LCEvent * evt )  ;
  void processRunHeader( LCRunHeader* run) ;

  void modifyEvent( LCEvent * evt ) { /* not needed */ ;}
  void modifyRunHeader(LCRunHeader* run){ /* not needed */ ;}
  //...
};
```

Here the *processEvent/RunHeader* methods are used, as they provide *read only* access to the data. The only modification of the data allowed in *read only* mode is the addition of new collections to the event (as this doesn't alter existing information). This will probably suffice for most analysis and reconstruction applications. So unless you need to do any of the following:

- remove collections from the event
- change elements in the collections, i.e. fix bugs
- add data to existing collections, e.g. background hits

you should use the *read only* mode in the *processRunHeader/Event* methods which is also default for the *readNextEvent/RunHeader* methods.
In an analysis job one could for example create histograms for every run in `processRunHeader( LCRunHeader* run)` and then fill the histograms in `processEvent( LCEvent * evt ).`
The corresponding – simplified but complete – main program will then look something like this (in C++):

```
#include "lcio.h"
#include "IO/LCReader.h"
#include "MyAnalysis.h"

using namespace lcio ;

int main(int argc, char** argv ){

  LCReader* lcReader = LCFactory::getInstance()->createLCReader() ;

  lcReader->open( argv[1] ) ;

  MyAnalysis myAnalysis ;
  lcReader->registerLCRunListener( &myAnalysis ) ;
  lcReader->registerLCEventListener( &myAnalysis ) ;

  lcReader->readStream() ;  // read the whole stream !

  lcReader->close() ;
  return 0;
}
```

A more elaborated example (C++) that defines a mini framework where you can specify analysis modules at runtime can be found in 4.1.

3.7.3 Accessing the data

Check the API documentation and the examples on how to access the data in the LCIO data structures. Mostly this is straightforward, e.g. printing run data is as easy as:

```
  LCRunHeader *runHdr ;
  ...
  cout << "  Run : " << runHdr->getRunNumber()
       << " - "      << runHdr->getDetectorName()
       << ":  "      << runHdr->getDescription()
       << endl ;
```

In order to access the information stored in the event, you need to know the collection names that hold the relevant data as well as the type of the objects, e.g. to access the TPC hits one could write code like the following:

```
LCCollection* col = evt->getCollection("TPCHits") ;

int nHits =  col->getNumberOfElements() ;

for( int i=0 ; i< nHits ; i++ ){

  TPCHit* hit =
    dynamic_cast<TPCHit*>( col->getElementAt( i ) ) ;

    cout << "    hit " << i
         << " - charge: "  << hit->getCharge()
         << " - time:   "  << hit->getTime()
         << endl ;
  ...
}
```

A few comments are in order:

- **collection name:** LCIO stores collections under user defined names. In order to retrieve the collection from the data, you need to know this name. Usually the person that created a data file will also have provided some documentation on the content (and on collection names).
  NB: The underlying implementation of the LCIO data format is *self documenting*, i.e. information on collection names and types is included in the files. This is used in the `anajob` and `dumpevent` example programs - run either of those to print collection names and types of an unknown LCIO file.
- **casts:** as LCIO uses 'untyped' collections, we have to down-cast from *LCObject* to the corresponding type, i.e. we need to know the type of the data. This is analogous to the *Collection* interface in Java. As C++ does not provide a common base class we had to introduce *LCObject* as a common base for event data in LCIO.

3.8 How to write LCIO files

Some examples for writing data with LCIO can be found in

```
src/cpp/src/EXAMPLE
  simjob.cc
  recjob.cc
src/java/hep/lcio/example
  SimJob.java
  RecJob.java
```

3.8.1 File handling

Before you can write to an LCIO file you have to create an instance of LCWriter using LCFactory:

```
LCWriter* lcWriter = LCFactory::getInstance()->createLCWriter() ;
```

or in Java:

```
LCWriter lcWriter = LCFactory.getInstance().createLCWriter();
```

Opening and closing the file is the same as described for the LCReader, e.g. in Java:

```
lcWriter.open( "my_data.slcio" ) ;

  // ... write sth. to the file....

lcWriter.close() ;
```

3.8.2 Writing to the file

The LCWriter interface only uses the interfaces in EVENT/hep.lcio.event. So all classes implementing these interfaces can be written with LCIO. This could in principle be used for existing classes in already existing applications eventhough copying the data into instances of the LCIO implementation classes is more convenient. If you design a new application in either Java or C++ you should use the implementation classes (IMPL/hep.lcio.implementation.event) that LCIO provides. These are also used in the following examples. The principle steps involved are: create an instance of the implementation class, set the attributes and then call the corresponding write method of the LCWriter interface. For example writing the run header is done with:

```
  LCRunHeaderImpl* runHdr = new LCRunHeaderImpl ;

  runHdr->setRunNumber( rn ) ;

  runHdr->setDetectorName( "HCALPPT" ) ;

  lcWrt->writeRunHeader( runHdr ) ;

  delete runHdr ;    // see below in  'LCIO Memory management in C++'
```

The event works in the same way, except that we now also need collections - the necessary steps are: create the event, set the event attributes, create a collection (with type name), add objects to the collection and finally add the collection to the event with a unique name:

```
 LCEventImpl*  evt = new LCEventImpl() ;     // create the event

 evt->setRunNumber(  rn   ) ;
 evt->setEventNumber( i ) ;                  // set the event attributes

 LCCollectionVec* calVec = new LCCollectionVec( LCIO::SIMCALORIMETERHIT );
                                             // create a collection for the
                                             // type SimCalorimeterHit

 for(int j=0;j<NHITS;j++){

   SimCalorimeterHitImpl* hit = new SimCalorimeterHitImpl ;
   hit->setEnergy( 3.1415 )  ;
   //...
   calVec->push_back( hit ) ;                // add hit objects to the collection
 }

 // ....
 evt->addCollection(calVec,"HCalPPTHits");   // add the collection with a name

 lcWrt->writeEvent( evt ) ;                  // write the event to the file


 // ------------ IMPORTANT ------------- !
 // we created the event so we need to delete it ...
 delete evt ;
 // -------------------------------------
```

The last line is of course only relevant for C++:

3.8.3 LCIO Memory management in C++

The philosophy of memory management in LCIO is: " If you created the object (using new), you are responsible for deleting it when it is no longer needed !".

In order to make this easier the LCEvent will delete everything that has been attached to the event. If you are only reading, LCIO will create the objects and thus delete them (when reading the next object of the same type) – thus no need for deleting on your side.
If you are reading and adding sth. to the event it is LCIO that deletes the event (as it created it) and thus also everything attached to it – again no need for deleting on your side.
As a rule of thumb: Use `delete evt/runhdr` *at the end of every event/run loop where you created the event/runHeader and don't use delete in all other cases.*
Of course if you use your own implementation of the EVENT interface you are also responsible for the memory management.

3.9 The Fortran Interface

The Fortran API of LCIO is based on a set of wrapper functions to the C++ implementation using "cfrotran.h" to create the correct Fortran name.
The main idea is that we use integers in Fortran to represent pointers to objects on the C++ side. There will be one wrapper function for every class method of the implementation classes (namespaces IMPL and IO) plus two additional methods to create and delete the object respectively. All functions that operate on objects, i.e. all functions except for the creation functions (constructors) need as a first argument the integer pointer to the particluar object.
By using a unique naming convention the documentation of the C++ version of the API can be utilized for the Fortran API as well.
Moreover example code written in C++ can be translated into Fortran basically line by line as far as LCIO is concerned – of course language specific control structures will have to be different. So even if you are only intereseted in using LCIO from Fortran it is probably a good idea to read the above sections on Java and C++ as well to get some insight into the general structure of LCIO.

3.9.1 Note

*The handling of pointers in Fortran **must** be done carefully. One has to ensure that pointers are always used in the right context. Pointers of objects given back to C++ can not be checked by the language to be applicable. In most cases of wrong pointer usage the execution of the executable will result in a **segmentation violation**.*

3.9.2 Naming convention

The following naming convention is used for the fortran wrapper functions to the C++ implementation of LCIO:

- all function names start with **lc**
- **lc** is followed by a three letter acronym that uniquley identifies the corresponding C++ class, e.g. **evt** for LCEvent. See table 2 for a complete listing (for the classes 'LCVec, STL vector' see in the **first part of Appendix B under utility**)
- the function name ends on the full lowercased name of the class method, e.g. `LCEventImpl::getRunNumber()` becomes **lcevtgetrunnumber()**
- the constructor and destructor of the class end on **create** and **delete** respectively, eg. **lcevt-delete()**
- All constants defined in `Event::LCIO` are defined as constants in Fortran with the same name prepended by **'LCIO_'**, e.g. the type name for MCParticles defined in C++ in `LCIO::MCPARTICLE` is defined in a character constant in Fortran named **LCIO_MCPARTICLE** (see the Fortran include file **lciof77apiext.inc**).

Additional methods to handle string, int and float vectors from Fortran are provided
for the user extension classes (LCIntVec, LCFloatVec, LCStringVec):
lcivcgetlength(), **lcivcgetintat()**

C++ class	f77 acronym
CalorimeterHitImpl	cah
ClusterImpl	clu
LCCollectionVec	col
LCEventImpl	evt
LCGenericObjectImpl	gob
MCParticleImpl	mcp
LCObject (Vector)	obv
ParticleIDImpl	pid
ReconstructedParticleImpl	rcp
LCReader	rdr
LCRelationImpl	rel
LCRunHeaderImpl	rhd
LCRelationNavigator	rnv
SimCalorimeterHitImpl	sch
SimTrackerHitImpl	sth
TPCHitImpl	tph
TrackerHitImpl	trh
TrackImpl	trk
LCWriter	wrt
LCVec, STL vector	???

Table 2: Three letter acronyms for f77 wrapper functions.

lcfvcgetlength(), **lcfvcgetfloatat()**
lcsvcgetlength(), **lcsvcgetstringat()**

and for the stl vector class:
intvectorgetlength(), **intvectorgetelement()**
floatvectorgetlength(), **floatvectorgetelement()**
stringvectorgetlength(), **stringvectorgetelement()** .

For strings the Fortran CHARACTER* declaration has to be large enough, otherwise the original string is truncated.
A Summary of all functions in the basic Fortran API is given in the first part of **Appendix B** .

3.9.3 Extension of the Base Fortran API

An additional set of Fortran functions is provided for user convenience. These are higher level functions that usually allow to access several attributes of data objects with one function call. These functions are declared in `$LCIO/src/f77/lciof77apiext.inc`, and are summarized in the second part of **Appendix B** .
Most Fortran programs for simulation use the `hepevt` common block. Conversion functions from the LCIO MCParticle collection to the `hepevt` common block and vice versa are also provided in the extended interface. See the example in 4.4.
The wrapper functions for the **LCParametersImpl** class are provided in the extension of the base Fortran API. This class is related to the classes
LCRunHeaderImpl, LCEventImpl, and LCCollectionVec
and has the methods
setIntValues, setFloatValues, setStringValues, getIntKeys, getFloatKeys, getStringtKeys, getIntValues, getFloatValues, and getStringValues .

The set/get methods are performed by the wrapper fuctions:

```
lcsetparameters (class_name, class_pointer, method_name, key_name, vector_pointer)
lcgetparameters (class_name, class_pointer, method_name, key_name, vector_pointer)
```

.
The vector_pointer is given to lcsetparameters and obtained from lcgetparameters and the vector content may be set or obtained by the stl vector wrapper fuctions.
Methods of the extended Fortran API can be used

to create the vectors for the user extension classes (LCIntVec, LCFloatVec, LCStringVec)
lcintvectorcreate(...)
lcfloatvectorcreate(...)
lcstringvectorcreate(...)

and to fetch the content of a vector
lcgetintvector(...)
lcgetfloatvector(...)
lcgetstringvector(...)

to create/delete the stl vectors having int, float, string, and pointer datamembers (e.g needed for the setShape method in the ClusterImpl Class)
intvectorcreate(...)
intvectordelete(...)
floatvectorcreate(...)
floatvectordelete(...)
stringvectorcreate(...)
stringvectordelete(...)
pointervectorcreate(...)
pointervectordelete(...)

and to fetch the content of a vector
getintvector(...)
getfloatvector(...)
getstringvector(...)
getpointervector(...)

The parameters of the functions above are explained in the the second part of **Appendix B** (The extended Fortran API to LCIO).

3.9.4 Reading and writing LCIO files

Examples for reading and writing LCIO files can be found in:

```
src/f77
  simjob.F
  recjob.F
  anajob.F
```

To build these examples, do:

```
gmake -C src/f77
```

These examples correspond to that provided in `$LCIO/src/cpp/src/EXAMPLE` for C++.

The complete interface is declared in the include file `$LCIO/src/f77/lciof77api.inc`. A simple example for reading an LCIO file with Fortran is:

```
 ...
#include "lciof77api.inc"
#include "lciof77apiext.inc"

      PTRTYPE reader, event, runhdr
      integer status, irun, ievent

      reader = lcrdrcreate()
      status = lcrdropen( reader, 'simjob.slcio' )

      if( status.eq.LCIO_ERROR) then
         goto 99
      endif

      do
         event = lcrdrreadnextevent( reader )
         if( event.eq.0 ) goto 11

         status = lcdumpevent( event )
      enddo
 11   continue
 ...
```

The function `lcdumpevent(event)` is part of the extended interface described in 3.9.3. Note that all functions that operate on existing objects have as a first argument the integer pointer to this particluar object. All functions that do not return a pointer to an object do return a status word instead that can be compared to `LCIO_ERROR` (`LCIO_SUCCESS`).
An example how to write a LCIO file from Fortran is:

```
 ...
      writer = lcwrtcreate()
      status = lcwrtopen( writer, filename , LCIO_WRITE_NEW )

 ...
      do iev = 1,nev

          event = lcevtcreate()

          status = lcevtsetrunnumber( event, irun )
          status = lcevtseteventnumber( event,  iev )

          schcol = lccolcreate( LCIO_SIMCALORIMETERHIT )

          do k=1,nhit

             hit = lcschcreate()
             status = lcschsetcellid0( hit, 1234 )
             status = lcschsetenergy( hit, energy )

             status = lccoladdelement( schcol, hit )
          enddo

          status = lcwrtwriteevent( writer , event )

c------- need to delete the event as we created it
          status = lcevtdelete( event )
      enddo

      status = lcevtaddcollection(event,schcol ,'MyCalHits')

      status = lcwrtclose( writer )
 ...
```

Note that as in the C++ case we have to delete the event if we created it as described in 3.8.3.

3.9.5 Using a C++ RunEventProcessor class for LCIO files processing in Fortran

A RunEventProcessor class is added to the Fortran API. To use it a wrapper function **lcrdrevent-processor** is provided which has to be called as:

```
...
#include "lciof77api.inc"
      integer          lcrdreventprocessor, status
      character*30     filenamesin(3)
      character*30     filenameout
      PTRTYPE          writer, pv
      common /outinfo/ writer
...
      filenamesin(1)  = 'simjob_f1.slcio'
      filenamesin(2)  = 'simjob_f2.slcio'
      filenamesin(3)  = 'simjob_f3.slcio'
      filenameout     = 'recjob_f.slcio'
***     create writer, open output file
      writer = lcwrtcreate()
      status = lcwrtopen( writer, filenameout , LCIO_WRITE_NEW )
***     create a stringvector of input file names
      pv = lcstringvectorcreate( filenamesin, 3, len (filenamesin(1))
***     create reader, open input file chain, start event loop
      status = lcrdreventprocessor( pv )
...
      subroutine processrunheader ( pheader )
      PTRTYPE pheader
***      some code if wanted, e.g.
      status = lcdumprunheader ( pheader )
      end

      subroutine modifyrunheader ( pheader )
      PTRTYPE pheader
***      some code if wanted
      end

      subroutine processevent ( pevent )
      PTRTYPE pevent
***      some code if wanted
      end

      subroutine modifyevent ( pevent )
#include "lciof77api.inc"
      PTRTYPE          pevent
      PTRTYPE          writer
      common /outinfo/ writer
```

```
...
***      some code if wanted
*
***      write event
      status = lcwrtwriteevent ( writer, pevent )
...

      end
```

The 4 subroutines **processrunheader, modifyrunheader, processevent, and modifyevent** must be provided if the wrapper function **lcrdreventprocessor** is used. They are called via the LCRunListener, LCEventListener classes. As an expample the code in `$LCIO/src/f77/recjob.F` may be looked at. The function `lcdumprunheader(pheader)` is part of the extended interface described in 3.9.3.

4 Real world examples

All the examples described above show how to use LCIO with Java, C++ and Fortran. They are build with LCIO and provided as binaries in `$LCIO/bin`. We also provide some examples that show how to use LCIO with some common physiscs simulation and analysis packages, such as *Pythia, Root and AIDA (JAS)* in `$LCIO/examples/java[cpp,f77]`. As these examples depend on external tools and libraries they are not build by default with LCIO. Please check the corresponding `README` files for instructions on how to build these examples:

4.1 lcioframe (C++, Root)

Defines a mini framework where you can specify analysis modules at runtime. The example modules create a root [7] file with some histograms. You can use the provided main program with your own analysis modules (and tools).

4.2 aida (JAVA, AIDA)

This Java example creates an AIDA file from an LCIO file with some histograms and an Ntuple. You can use any AIDA [10] compliant analysis tools for viewing the histograms, e.g. JAS3 [11].

4.3 lciohbook (Fortran, Hbook)

A Fortran example that creates an Hbook [9] file from an LCIO file with some histograms to be analyzed with PAW [9].

4.4 pythia (Fortran, Pythia)

An ASCII file of generator output can be produced by the script `pythiahepevt.sh`. This script runs a Pythia [8] job using the Pythia routine **PYHEPC** to fill the `hepevt` common block and a modified version of **LCWRITE** to create the ASCII file (is included in the script). To run this script one has to provide `cernlib` [9] 2002 or later. The script has to be modified if necessary to provide a valid path to the `cernlib` libraries.
The program **PYSIMJOB** (`pysimjob.F`) reads the ASCII file, fills the `hepevt` common block, and writes a LCIO file which is read again in a second pass to fill the `hepevt` common block.

Appendix

A ChangeLog

A.1 Changes from v1.3 to v1.4

There have been a number of changes in LCIO wrt. to the last public release v01-03:

- **LCGenericObjects in Java**
  Support for user defined objects that have an arbitrary number of attributes of type *int, float* and *double* now also in Java.

- **Support for subset collections**
  Added flag isSubset to LCCollection that allows to have collections that are subsets of existing collections in the event. In case the collection is not transient only references/pointers are written to the output file.

- **Small API change in LCGenericObject (C++)**
  Changed the return value of `LCGenericObject::getTypeName()` and `LCGenericObject::getDataDescription(` from `const std::string&` to `const std::string` to make the implementation easier.

- **UTIL::LCFixedObject template**
  Added convenience template `UTIL::LCFixedObject` for user defined subclasses of LCGenericObjects with fixed size. See: `src/cpp/src/EXAMPLE/CalibrationConstant.h` for example on how to use it.

- **Definition of LCEvent::getTimeStamp()**
  The 64bit event timestamp is defined to be **ns since 1.1.1970 00:00:00 UTC.** Changed C++ API to use long long (64 bit).

- **UTIL::LCTime (C++)**
  Helper class to convert between time stamps and calendar time.

- **Multiple I/O streams in C+**
  Support for multiple instances of LCReader and LCWriter objects (files) added.

- **LCIODEBUG flag in C++**
  Added LCIODEBUG environment variable to control compilation.

A.2 Changes from v1.0 to v1.3

- **Tracks, Cluster and ReconstructedParticles added**
  These are the main classes for reconstruction output.

- **RawCalorimeterHit added**
  A calorimeter hit class for testbeam data that reflects 'real data' more closely than CalorimeterHit.

- **LCRelation added**
  Can be used to map arbitrary weighted n to m relationships, with $n, m \geq 1$.

- **LCGenericObject added**
  Support for user defined objects that have an arbitrary number of attributes of type *int, float* and *double.*

- **Added time information to CalorimterHit**
  An optional time word has been added to CalorimeterHit.

- **Transient flag added to LCCollection**
  LCCollections can now be flagged as transient for use in applications that have LCEvent as the underlying transient data model.

- **Default flag added to LCCollection**
  LCCollections can now be flagged as default lists. There should be only one default LCCollection for every type.
- **New package UTIL for C++**
  A new namespace UTIL has been added to the C++ version. It holds convenient classes and methods that facilitate the handling ov LCIO data.
- **Support of CLHEP::HepLorentzvector for C++**
  For C++ there is a template LCFourVector that can be used to create CLHEP::HepLorentzvectors from MCParticles and ReconstructedParticles. A Java equivalent will added in the next release.
- **Added 'chain reader' to LCReader**
  Added a new method LCReader::open(vector fileNames) that allows to specify a list of files that will be read sequentially.
- **Made API more consistent**
  A number of methods have been added and/or modified to make the API more consistent. Some methods that have a new name or calling signature have been marked as deprecated. Usdage of those methods will generate a warning - please switch to the new methods.
- **StdHep-Reader**
  Added support for reading binary StdHep files and create an LCCollection of MCParticles. See UTIL::LCStdHepRdr and EXAMPLES/stdhepjob.cc.

B Summary of Fortran API Functions

The basic Fortran API to LCIO

Remarks:

The return value of the functions and the meaning of arguments are either:
∗ pointers denoted by a name beginning with the **letter p**
∗ character strings denoted by **...name** or **string**
∗ logicals denoted by **bool**
∗ integers denoted by **status** or a variable name starting with **i** or **n**
∗ long integers (INTEGER*8) denoted by **ilong**
∗ double precision variables name starting with **d**
∗ reals **else** ∗ arrays are denoted by a name ending with **v**

```
class LCReader:

create                 -> preader = lcrdrcreate()
delete                 -> status  = lcrdrdelete( preader )
open                   -> status  = lcrdropen( preader, filename )
close                  -> status  = lcrdrclose( preader )
readStream             -> status  = lcrdrreadstream( preader, nmax )
readNextRunHeader      -> pheader = lcreadnextrunheader( preader, iaccessmode )
readNextEvent          -> pevent  = lcrdrreadnextevent( preader, iaccessmode )
readEvent              -> pevent  = lcrdrreadevent( preader, irun, ievt )

RunEventProcessor (includes registerLCEventListener, registerLCRunListener)
                       -> status  = lcrdreventprocessor( filename )

class LCWriter:

create                 -> pwriter = lcwrtcreate()
delete                 -> status  = lcwrtdelete( pwriter )
open                   -> status  = lcwrtopen( pwriter, filename, imode )
close                  -> status  = lcwrtclose( pwriter )
writeRunHeader         -> status  = lcwrtwriterunheader( pwriter, pheader )
writeEvent             -> status  = lcwrtwriteevent( pwriter, pevent )


class LCRunHeader:

create                 -> pheader = lcrhdcreate()
delete                 -> status  = lcrhddelete( pheader )
setRunNumber           -> status  = lcrhdsetrunnumber( pheader )
setDetectorName        -> status  = lcrhdsetdetectorname( pheader , detname)
setDescription         -> status  = lcrhdsetdescription( pheader , descrstring )
addActiveSubdetector   -> status  = lcrhdaddactivesubdetector( pheader , sdname )

getRunNumber           -> irun    = lcrhgetrunnumber( pheader )
getDetectorName        -> detname = lcrhdgetdetectorname( pheader )
getDescription         -> string  = lcrhdgetdescription( pheader )

getActiveSubdetectors by:
getActiveSubdetectors  -> psdvec  = lcrhdgetactivesubdetectors( pheader )
getNumberOfElements    -> nelem   = lcsvcgetlength( psdvec )                         (stl vector Interface)
getElementAt           -> sdname  = lcsvcgetstringat( psdvec , i ) (i=1,...,nelem)   (stl vector Interface)


class LCEvent:

create                 -> pevent  = lcevtcreate()
delete                 -> status  = lcevtdelete( pevent )
setRunNumber           -> status  = lcevtsetrunnumber( pevent , irun)
setEventNumber         -> status  = lcevtseteventnumber( pevent , ievt)
setDetectorName        -> status  = lcevtsetdetectorname( pevent , detname)
setTimeStamp           -> status  = lcevtsettimestamp( pevent , ilong)
addCollection          -> status  = lcevtaddcollection( pevent , pcol , colname)
removeCollection       -> status  = lcevtremovecollection( pevent , colname)

getRunNumber           -> irun    = lcevtgetrunnumber( pevent )
getEventNumber         -> ievt    = lcevtgeteventnumber( pevent )
getDetectorName        -> detname = lcevtgetdetectorname( pevent )
getTimeStamp           -> ilong   = lcevtgettimestamp( pevent )
getCollection          -> pcol    = lcevtgetcollection( pevent , colname)

getCollectionNames by:
getCollectionNames     -> pstv    = lcevtgetcollectionnames( pevent )
getNumberOfElements    -> nelem   = lcsvcgetlength( pstv )                            (stl vector Interface)
getElementAt           -> colname = lcsvcgetstringat( pstv , i ) (i=1,...,nelem)      (stl vector Interface)
```

```
class LCCollection:

create                -> pcol    = lccolcreate( colname )
delete                -> status  = lccoldelete( pcol )
addElement            -> status  = lccoladdelement( pcol , pobject)
removeElementAt       -> status  = lccolremoveelementat ( pcol , i )
setFlag               -> status  = lccolsetflag( pcol , iflag )
setTransient          -> status  = lccolsettransient( pcol , bool ) (bool=true,false)

getTypeName           -> name    = lccolgettypename( pcol )
getNumberOfElements   -> number  = lccolgetnumberofelements( pcol )
getElementAt          -> pobject = lccolgetelementat( pcol , i )  (i=1,...,number)
getFlag               -> iflag   = lccolgetflag( pcol )
isTransient           -> bool    = lccolistransient( pcol )


class LCRelation, LCObject:

create (defaults)     -> prel    = lcrelcreate0()
create                -> prel    = lcrelcreate( pobjectfrom , pobjectto, weight )
delete                -> status  = lcreldelete( prel )
setFrom               -> status  = lcrelsetfrom( prel, pobjectfrom )
setTo                 -> status  = lcrelsetto( prel, pobjectto )
setWeight             -> status  = lcrelsetweight( prel, weight )

id                    -> id      = lcrelid( prel )
getFrom               -> pfrom   = lcrelgetfrom( prel )
getTo                 -> pto     = lcrelgetto( prel )
getWeight             -> weight  = lcrelgetweight( prel )

getLength             -> nelem   = lcobvgetlength( pobjv )        (pobjv = pfrom, pto)
getObject             -> id      = lcobvgetobject( pobjv , i )    (i=1,...,nelem)
getObjectID           -> id      = lcobvgetobjectid( pobjv , i )  (i=1,...,nelem)
getWeight             -> weight  = lcobvgetweight( pobjv , i )    (i=1,...,nelem)


class LCRelationNavigator:

create                -> prnv    = lcrnvcreate( fromname, toname )
create (from col)     -> prnv    = lcrnvcreatefromcollection( pcol )
delete                -> status  = lcrnvdelete()
addRelation           -> status  = lcrnvgaddrelation( prel, pobjectfrom , pobjectto, weight )
removeRelation        -> status  = lcrnvgremoverelation( prel, pobjectfrom , pobjectto )
createLCCollection    -> pcol    = lcrnvcreatecollection( prel )

getFromType           -> namefr  = lcrnvgetfromtype( prnv )
getToType             -> nameto  = lcrnvgettotype( prnv )
getRelatedToObjects   -> pobjvto = lcrnvgetrelatedtoobjects( prnv, pobj )
getRelatedFromObjects -> pobjvfr = lcrnvgetrelatedfromobjects( prnv, pobj )
getRelatedToWeights   -> pweightv= lcrnvgetrelatedtoweights( prnv, pobj )
getRelatedFromWeights -> pweightv= lcrnvgetrelatedfromweights( prnv, pobj )



class LCGenericObject:

create                -> pgob    = lcgobcreate()
create (dimensions)   -> pgob    = lcgobcreatefixed( nint, nfloat, ndouble )
delete                -> status  = lcgobdelete( pgob )
setIntVal             -> status  = lcgobsetintval( pgob, i, ival )
setFloatVal           -> status  = lcgobsetfloatval( pgob, i, fval )
setDoubleVal          -> status  = lcgobsetdoubleval( pgob, i, dval )

id                    -> id      = lcgobid( pgob )
getNInt               -> nint    = lcgobgetnint( pgob )
getNFloat             -> nfloat  = lcgobgetnfloat( pgob )
getNDouble            -> ndouble = lcgobgetdoubleval( pgob )
getIntVal             -> ival    = lcgobgetintval( pgob , i )      (i=1,...,nint)
getFloatVal           -> fval    = lcgobgetfloatval( pgob , i )    (i=1,...,nfloat)
getDoubleVal          -> dval    = lcgobsetdoubleval( pgob , i )   (i=1,...,ndouble)
isFixedSize           -> bool    = lcgobisfixedsize( pgob )
getTypeName           -> name    = lcgobgettypename( pgob )
getDataDescription    -> string  = lcgobgetdatadescription( pgob )


class SimTrackerHit:

create                -> pthit   = lcsthcreate()
delete                -> status  = lcsthdelete( pthit )
setCellID             -> status  = lcsthsetcellid( pthit , icellid )
setPosition           -> status  = lcsthsetposition( pthit , dposv )
setdEdx               -> status  = lcsthsetdedx( pthit , fdedx )
setEDep               -> status  = lcsthsetedep( pthit , fedep )
```

```
setTime                -> status  = lcsthsettime( pthit , ftime )
setMCParticle          -> status  = lcsthsetmcparticle( pthit , pmcp )
setMomentum            -> status  = lcsthsetmomentum( pthit , fpv )
setMomentumXYZ         -> status  = lcsthsetmomentumxyz( pthit , px, py, pz )
setPathLength          -> status  = lcsthsetpathlength( pthit , pathl )

getCellID              -> icellid = lcsthgetcellid( pthit )
getPosition            -> dposv(i)= lcsthgetposition( pthit , i ) (i=1,2,3)
getMomentum            -> fpv(i)  = lcsthgetmomentum( pthit , i ) (i=1,2,3)
getPathLength          -> pathl   = lcsthgetpathlength ( pthit )
getdEdx                -> fdedx   = lcsthgetdedx( pthit )
getEDep                -> fedep   = lcsthgetedep( pthit )
getTime                -> ftime   = lcsthgettime( pthit )
getMCParticle          -> pmcp    = lcsthgetmcparticle( pthit )


class SimCalorimeterHit:

create                 -> pschit  = lcschcreate()
delete                 -> status  = lcschdelete( pschit )
setCellID0             -> status  = lcschsetcellid0( pschit , icellid0 )
setCellID1             -> status  = lcschsetcellid1( pschit , icellid1 )
setEnergy              -> status  = lcschsetenergy( pschit , energy )
setPosition            -> status  = lcschsetposition( pschit , posv )
addMCParticleContr..   -> status  = lcschaddmcparticlecontribution( pschit , pmcp , energy , time , ipdg )

id                     -> id      = lccahid( pschit )
getCellID0             -> icellid0= lcschgetcellid0( pschit )
getCellID1             -> icellid1= lcschgetcellid1( pschit )
getEnergy              -> energy  = lcschgetenergy( pschit )
getPosition            -> status  = lcschgetposition( pschit , posv )
getNMCParticles        -> number  = lcschgetnmcparticles( pschit )
getParticleCont        -> pmcp    = lcschgetparticlecont( pschit , i ) (i=1,...,number)
getEnergyCont          -> energy  = lcschgetenergycont( pschit , i )   (i=1,...,number)
getTimeCont            -> time    = lcschgettimecont( pschit , i )     (i=1,...,number)
getPDGCont             -> ipdg    = lcschgetpdgcont( pschit , i )      (i=1,...,number)


class CalorimeterHit:

create                 -> pchit   = lccahcreate()
delete                 -> status  = lccahdelete( pchit )
setCellID0             -> status  = lccahsetcellid0( pchit , icellid0 )
setCellID1             -> status  = lccahsetcellid1( pchit , icellid1 )
setEnergy              -> status  = lccahsetenergy( pchit , energy )
setPosition            -> status  = lccahsetposition( pchit , posv )
setTime                -> status  = lccahsettime( pchit , time )
setType                -> status  = lccahsettype( pchit , itype )
setRawHit              -> status  = lccahsetrawhit( pchit , praw )

id                     -> id      = lccahid( pchit )
getCellID0             -> icellid0= lccahgetcellid0( pchit )
getCellID1             -> icellid1= lccahgetcellid1( pchit )
getEnergy              -> energy  = lccahgetenergy( pchit )
getPosition            -> status  = lccahgetposition( pchit , posv )
getTime                -> time    = lccahgettime( pchit )
getType                -> itype   = lccahgettype( pchit )
getRawHit              -> prawhit = lccahsetrawhit( pchit )


class TPCHit:

create                 -> pthit   = lctphcreate()
delete                 -> status  = lctphdelete( pthit )
setCellID              -> status  = lctphsetcellid( pthit , icellid )
setTime                -> status  = lctphsettime( pthit , time )
setCharge              -> status  = lctphsetcharge( pthit , charge )
setQuality             -> status  = lctphsetquality( pthit , iquality )
setRawData             -> status  = lctphsetrawdata( pthit , irawv, nraw )

id                     -> id      = lctphid( pthit )
getCellID              -> icellid = lctphgetcellid( pthit )
getTime                -> time    = lctphgettime( pthit )
getCharge              -> charge  = lctphgcharge( pthit )
getQuality             -> iquality= lctphgetquality( pthit )
getNRawDataWords       -> nraw    = lctphgetnrawdatawords( pthit )
getRawDataWord         -> iword   = lctphgetrawdataword( pthit, i )  (i=1,...,nraw)


class TrackerHit:

create                 -> ptrhit  = lctrhcreate()
delete                 -> status  = lctrhdelete( ptrhit )
```

```
setPosition            -> status  = lctrhsetposition( ptrhit, dposv )
setCovMatrix           -> status  = lctrhsetcovmatrix( ptrhit, covmxv )
setdEdx                -> status  = lctrhsetdedx( ptrhit , dedx )
setEDep                -> status  = lctrhsetedep( ptrhit , edep )
setEDepError           -> status  = lctrhsetedeperr( ptrhit , error )
setTime                -> status  = lctrhsettime( ptrhit, time )
setType                -> status  = lctrhsettype( ptrhit, itype )
setQuality             -> status  = lctrhsetquality( ptrhit , iquality )
addRawHit              -> status  = lctrhaddrawhit( ptrhit, prawh )

id                     -> id      = lctrhid( ptrhit )
getPosition            -> status  = lctrhgetposition( ptrhit, dposv )
getCovMatrix           -> status  = lctrhsetcovmatrix( ptrhit, covmxv )
getdEdx                -> dedx    = lctrhgetdedx( ptrhit )
getEDep                -> edep    = lctrhgetedep( ptrhit )
getEDepError           -> edeperr = lctrhgetedeperr( ptrhit )
getTime                -> time    = lctrhgettime( ptrhit )
getType                -> itype   = lctrhgettype( ptrhit )
getQuality             -> iquality= lctrhgetquality( ptrhit )
getRawHits             -> prawhv  = lctrhgetrawhits( ptrhit )


class Track:

create                 -> ptrk    = lctrkcreate()
delete                 -> status  = lctrkdelete( ptrk )
setTypeBit             -> status  = lctrksettypebit( ptrk, ibit, ival)
setOmega               -> status  = lctrksetomega ( ptrk, omega )
setTanLambda           -> status  = lctrksettanlambda( ptrk, tanlambda )
setPhi                 -> status  = lctrksetphi( ptrk, phi )
setD0                  -> status  = lctrksetd0( ptrk, d0 )
setZ0                  -> status  = lctrksetz0( ptrk, z0 )
setCovMatrix           -> status  = lctrksetcovmatrix( ptrk, covmxv )
setReferencePoint      -> status  = lctrksetreferencepoint( ptrk, refpointv )
setChi2                -> status  = lctrksetchi2( ptrk, chi2 )
setNdf                 -> status  = lctrksetndf( ptrk, ndf )
setdEdx                -> status  = lctrksetdedx( ptrk, dedx )
setdEdxError           -> status  = lctrksetdedxerror( ptrk, dedxerr )
setRadiusOfInnermostHit-> status  = lctrksetradiusofinnermosthit( ptrk, radius)
addTrack               -> status  = lctrkaddtrack( ptrk, ptrack )
addHit                 -> status  = lctrkaddhit( ptrk, phit )
subdetectorHitNumbers  -> status  = lctrksetsubdetectorhitnumbers( ptrk, intv, nintv)   (not in C++ API)

id                     -> id      = lctrkid( ptrk )
getType                -> itype   = lctrkgettype( ptrk )
getOmega               -> omega   = lctrkgetomega( ptrk )
getTanLambda           -> tanlam  = lctrkgettanlambda( ptrk )
getPhi                 -> phi     = lctrkgetphi( ptrk )
getD0                  -> d0      = lctrkgetd0( ptrk )
getZ0                  -> z0      = lctrkgetz0( ptrk )
getCovMatrix           -> status  = lctrkgetcovmatrix( ptrk, covmxv )
getReferencePoint      -> status  = lctrkgetreferencepoint( ptrk, refpointv )
getChi2                -> chi2    = lctrkgetchi2( ptrk )
getNdf                 -> ndf     = lctrkgetndf( ptrk )
getdEdx                -> dedx    = lctrkgetdedx( ptrk )
getdEdxError           -> dedxerr = lctrkgetdedxerror( ptrk )
getRadiusOfInnermostHit-> radius  = lctrkgetradiusofinnermosthit( ptrk )
subdetectorHitNumbers  -> status  = lctrkgetsubdetectorhitnumbers( ptrk, intv, nintv)
getTracks              -> ptrackv = lctrkgettracks( ptrk )
getTrackerHits         -> ptrhitv = lctrkgettrackerhits( ptrk )


class Cluster:

create                 -> pclu    = lcclucreate()
delete                 -> status  = lccludelete( pclu )
setTypeBit             -> status  = lcclusettypebit( pclu, ibit, ival)
setEnergy              -> status  = lcclusetenergy( pclu, energy )
setPosition            -> status  = lcclusetposition( pclu, posv )
setPositionError       -> status  = lcclusetpositionerror( pclu, poserrv )
setITheta              -> status  = lcclusetitheta( pclu, theta )
setIPhi                -> status  = lcclusetiphi( pclu, phi )
setDirectionError      -> status  = lcclusetdirectionerror( pclu, direrrv )
setShape               -> status  = lcclusetshape( pclu, pshapev )
addParticleID          -> status  = lccluaddparticleid( pclu, ppid )
addCluster             -> status  = lccluaddcluster( pclu, pcluadd )
addHit                 -> status  = lccluaddhit( pclu, pcalhit, weight )
setsubdetectorEnergies -> status  = lcclusetsubdetectorenergies( pclu, energiesv, nenergies)  (not in C++ API)

id                     -> id      = lccluid( pclu )
getType                -> itype   = lcclugettype( pclu )
getEnergy              -> energy  = lcclugetenergy( pclu )
getPosition            -> status  = lcclugetposition( pclu, posv )
```

```
getPositionError       -> status  = lcclugetpositionerror( pclu, poserrv )
getITheta              -> theta   = lcclugetitheta( pclu, theta )
getIPhi                -> phi     = lcclugetiphi( pclu, phi )
getDirectionError      -> status  = lcclugetdirectionerror( pclu, direrr )
getShape               -> pshapev = lcclugetshape( pclu )
getParticleIDs         -> ppidvec = lcclugetparticleids( pclu )
getClusters            -> pcluget = lcclugetclusters( pclu )
getCalorimeterHits     -> pcalhv  = lcclugetcalorimeterhits( pclu )
getSubdetectorEnergies -> pfloatv = lcclugetsubdetectorenergies( pclu )
getHitContributions    -> status  = lcclugethitcontributions( pclu, energiesv, nenergies)   (not in C++ API)



class ReconstructedParticle:

create                 -> prcp    = lcrcpcreate()
delete                 -> status  = lcrcpdelete( prcp )
setType                -> status  = lcrcpsettype( prcp, itype )
setMomentum            -> status  = lcrcpsetmomentum( prcp, xmomv )
setEnergy              -> status  = lcrcpsetenergy( prcp, energy )
setCovMatrix           -> status  = lcrcpsetcovmatrix( prcp, covmxv )
setMass                -> status  = lcrcpsetmass( prcp, xmass )
setCharge              -> status  = lcrcpsetcharge( prcp, charge ) ;
setReferencePoint      -> status  = lcrcpsetreferencepoint( prcp, refpointv ) ;
setGoodnessOfPID       -> status    lcrcpsetgoodnessofpid( prcp, goodns ) ;
addParticleID          -> status  = lcrcpaddparticleid( prcp, pid ) ;
addParticle            -> status  = lcrcpaddparticle( prcp, pparticle, weigth ) ;
addCluster             -> status  = lcrcpaddcluster( prcp, pclus, weigth ) ;
addTrack               -> status  = lcrcpaddtrack( prcp, ptrack, weigth ) ;

id                     -> id      = lcrcpid( prcp )
getType                -> itype   = lcrcpgettype( prcp )
isPrimary              -> lprim   = lcrcpisprimary( prcp ) (lprim = type logical)
getMomentum            -> status  = lcrcpgetmomentum( prcp, xmomv )
getEnergy              -> energy  = lcrcpgetenergy( prcp )
getCovMatrix           -> status  = lctrkgetcovmatrix( prcp, covmxv )
getMass                -> xmass   = lcrcpgetmass( prcp )
getCharge              -> charge  = lcrcpgetcharge( prcp )
getReferencePoint      -> status  = lcrcpgetreferencepoint( prcp, refpointv )
getGoodnessOfPID       -> goodns  = lcrcpgetgoodnessofpid( prcp )
getParticleIDs         -> pids    = lcrcpgetparticleids( prcp )
getParticles           -> ppartv  = lcrcpgetparticles( prcp )
getClusters            -> pclusv  = lcrcpgetclusters( prcp )
getTracks              -> ptrkv   = lcrcpgettracks( prcp )


class ParticleID:

create                 -> ppid    = lcpidcreate()
delete                 -> status  = lcpiddelete( ppid )
setType                -> status  = lcpidsettype( ppid, idtype )
setPDG                 -> status  = lcpidsetpdg( ppid, ipdg )
setLikelihood          -> status  = lcpidsetlikelihood( ppid, xlogl )
setAlgorithmType       -> status  = lcpidsetalgorithmtype( ppid, itype )
addParameter           -> status  = lcpidaddparameter( ppid, param )

id                     -> id      = lcpidid( ppid )
getType                -> idtype  = lcpidgettype( ppid )
getPDG                 -> ipdg    = lcpidgetpdg( ppid )
getLikelihood          -> xlogl   = lcpidgetlikelihood( ppid )
getAlgorithmType       -> itype   = lcpidgetalgorithmtype( ppid )
getParameters          -> status  = lcpidgetparameters( ppid, paramv, nparam)


class MCParticle:

create                 -> pmcp    = lcmcpcreate()
delete                 -> status  = lcmcpdelete( pmcp )
addParent              -> status  = lcmcpaddparent( pmcp , pmcpp )
setPDG                 -> status  = lcmcpsetpdg( pmcp , ipdg )
setGeneratorStatus     -> status  = lcmcpsetgeneratorstatus( pmcp , istatus )
setSimulatorStatus     -> status  = lcmcpsetsimulatorstatus( pmcp , istatus )
setVertex              -> status  = lcmcpsetvertex( pmcp , dvtxv )
setEndpoint            -> status  = lcmcpsetendpoint( pmcp , dvtxv )
setMomentum            -> status  = lcmcpsetmomentum( pmcp , momv )
setMass                -> status  = lcmcpsetmass( pmcp , mass )
setCharge              -> status  = lcmcpsetcharge( pmcp , charge )

getNumberOfParents     -> number  = lcmcpgetnumberofparents( pmcp )
getParent              -> pmcpp   = lcmcpgetparent( pmcp , i )
getNumberOfDaughters   -> number  = lcmcpgetnumberofdaughters( pmcp )
getDaughter            -> pmcpd   = lcmcpgetdaughter( pmcp , i )  (i=1,...,number)
getPDG                 -> ipdg    = lcmcpgetpdg( pmcp )
```

```
getGeneratorStatus     -> istatg  = lcmcpgetgeneratorstatus( pmcp )
getSimulatorStatus     -> istats  = lcmcpgetsimulatorstatus( pmcp )
getVertex              -> status  = lcmcpgetvertex( pmcp , dvtxv )
getEndpoint            -> status  = lcmcpgetendpoint( pmcp , dvtxv )
getMomentum            -> status  = lcmcpgetmomentum( pmcp , dmomv )
getEnergy              -> denergy = lcmcpgetenergy( pmcp )
getMass                -> dmass   = lcmcpgetmass( pmcp )
getCharge              -> charge  = lcmcpgetcharge( pmcp )



utility:

LCIntVec, LCFloatVec, LCStrVec classes:

getLengthofIntVector   -> nelem   = lcivcgetlength( pintvec )
getIntAt               -> int     = lcivcgetintat( pintvec , i ) (i=1,...,nelem)
getLengthofFloatVector -> nelem   = lcfvcgetlength( pfloatvec )
getFloatAt             -> float   = lcfvcgetfloatat( pfloatvec , i ) (i=1,...,nelem)
getLengthofStringVector-> nelem   = lcsvcgetlength( pstrvec )
getStringAt            -> string  = lcsvcgetstringat( pstrvec , i ) (i=1,...,nelem)

stl vector class (int, float, string, pointer):

IntVectorGetLength     -> nelem   =  intvectorgetlength( pintvec )
IntVectorGetElement    -> int     =  intvectorgetelement( pintvec , i ) (i=1,...,nelem)
FloatVectorGetLength   -> nelem   =  floatvectorgetlength( pfloatvec )
FloatVectorGetElement  -> float   =  floatvectorgetelement( pfloatvec , i ) (i=1,...,nelem)
StringVectorGetLength  -> nelem   =  stringvectorgetlength( pstrvec )
StringVectorGetElement -> string  =  stringvectorgetelement( pstrvec , i ) (i=1,...,nelem)
PointerVectorGetLength -> nelem   =  pointervectorgetlength( ppointervec )
PointerVectorGetElement-> pointer =  pointervectorgetelement( ppointervec , i ) (i=1,...,nelem)
```

The extended Fortran API to LCIO

Remarks:

The return value of the functions and the meaning of arguments are either:
∗ pointers denoted by a name beginning with the **letter p**
∗ character strings denoted by **...name** or **string**
∗ integers denoted by **status** or a variable name starting with **i** or **n**
∗ double precision variables denoted by a name starting with **d**
∗ arrays denoted by a name ending with **v**
∗ reals **else**
If arguments of the type **array of character strings** are used the last argument has to be the length of a character string in the array.
Integers starting with n are also used to give the length of an array (input/output argument for get... functions, input: dimension of the array, output: number of values stored).

```
for class LCReader:

lcrdropenchain         -> status = lcrdropenchain(preader, filenamesv, nfiles, len(filenamesv(1)))

for class LCRunHeader:

writeRunHeader         -> status  = lcwriterunheader( pwriter, irun, detname, descrname, sdnamev, nsdn, len(sdnamev(1)) )
readNextRunHeader      -> pheader = lcreadnextrunheader( preader, irun, detname, descrname, sdnamev, nsdn, len(sdnamev(1)) )

for class LCEvent:

setEventHeader         -> status  = lcseteventheader ( pevent, irun, ievt, itim, detname )
getEventHeader         -> status  = lcgeteventheader ( pevent, irun, ievt, itim, detname )
dumpEvent              -> status  = lcdumpevent ( pevent )
dumpEventDetailed      -> status  = lcdumpeventdetailed ( pevent )

for class SimTrackerHit:

addSimTrackerHit       -> status  = lcaddsimtrackerhit( pcolhitt, icellid, dposv, fdedx, ftime, pmcp )
getSimTrackerHit       -> status  = lcgetsimtrackerhit( pcolhitt, i, icellid, dposv, fdedx, ftime, pmcp )

for class SimCalorimeterHit:

addSimCaloHit          -> phit    = lcaddsimcalohit( pcolhitc, icellid0, icellid1, energy, posv )
addSimCaloHitMCont     -> status  = lcschaddmcparticlecontribution( phit, pmcp, energy, time, ipdg )  (from basic f77 API)
getSimCaloHit          -> phit    = lcgetsimcalohit( pcolhitc, i, icellid0, icellid1, energy, posv )
getSimCaloHitMCont     -> status  = lcgetsimcalohitmccont( phit, i, pmcp, energy, time, ipdg )

for class MCParticle:

getMCParticleData      -> status  = lcgetmcparticledata ( pmcp, ipdg, igstat, isstat, dvtxv, momv, mass, charge, ndaughters )

for class HEPEVT (Fortran interface  to COMMON /HEPEVT/ ):

toHepEvt               -> status  = lcio2hepevt( pevent )
fromHepEvt             -> status  = hepevt2lcio( pevent )

for the user extension classes (LCIntVec, LCFloatVec, LCStringVec):

createIntVector        -> pvec    = lcintvectorcreate( intv, nint )
createFloatVector      -> pvec    = lcfloatvectorcreate( floatv, nfloat )
createStringVector     -> pvec    = lcstringvectorcreate( stringsv, nstrings, len(stringsv(1)) )

getIntVector           -> status  = lcgetintvector( pvec , intv, nintv )
getFloatVector         -> status  = lcgetfloatvector( pvec , floatv , nfloatv )
getStringVector        -> status  = lcgetstringvector( pvec , stringv , nstringv, len(stringv(1)) )

for the stl vector class (int, float ,string, pointer):

IntVectorCreate        -> pveci   = intvectorcreate( intv, nintv )
IntVectorDelete        -> status  = intvectordelete( pveci )
FloatVectorCreate      -> pvecf   = floatvectorcreate( floatv , nfloatv )
FloatVectorDelete      -> status  = floatvectordelete( pvecf )
StringVectorCreate     -> pvecs   = stringvectorcreate( stringsv, nstrings, len(stringsv(1)) )
StringVectorDelete     -> status  = stringvectordelete( pvecs )
PointerVectorCreate    -> pvecp   = pointervectorcreate( pointerv, npointerv )
PointerVectorDelete    -> status  = pointervectordelete( pvecp )

getIntVector           -> status  = getintvector( pveci , intv, nintv )
getFloatVector         -> status  = getfloatvector( pvecf , floatv , nfloatv )
getStringVector        -> status  = getstringvector( pvecs , stringv , nstringv, len(stringv(1)) )
get PointerVector      -> status  = getpointervector( pvecp , pointerv, npointerv )
```

References

[1] LCIO: A persistency framework for linear collider simulation studies.
By Frank Gaede (DESY), Ties Behnke (DESY & SLAC), Norman Graf, Tony Johnson (SLAC). SLAC-PUB-9992, CHEP-2003-TUKT001, Jun 2003. 5pp. Talk given at 2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03), La Jolla, California, 24-28 Mar 2003. e-Print Archive: physics/0306114 / LC Note LC-TOOL-2003-053

[2] LCIO Homepage:
http://lcio.desy.de

[3] SIO documentation:
http://www-sldnt.slac.stanford.edu/nld/new/Docs/FileFormats/sio.pdf

[4] AID Homepage:
http://java.freehep.org/aid/index.html

[5] javadoc Homepage:
http://java.sun.com/j2se/javadoc

[6] doxygen Homepage:
http://www.stack.nl/~dimitri/doxygen

[7] root Homepage:
http://root.cern.ch/

[8] Pythia Homepage:
http://www.thep.lu.se/~torbjorn/Pythia.html

[9] cernlib Homepage:
http://cernlib.web.cern.ch/cernlib

[10] AIDA Homepage:
http://aida.freehep.org

[11] JAS3 Homepage:
http://jas.freehep.org/jas3/index.html CMake Homepage:
http://www.cmake.org

[12]