Stanford Linear Accelerator Center
SLAC
P.O.Box 4349
Stanford, CA 94309

GLAST offline software

Raw data definition (Monte Carlo)

Document Edition:	0.7
Document Created:	February 2000
Document Date:	February 2002
Document Status:	Draft
Document Author:	T. Hansl-Kozanecka, ...

Abstract

This document proposes the definition of the classes for the raw data, which are the output of the Monte Carlo simulation and the input for reconstruction.

This is a living document and updates may be frequent. Please send comments to Traudl.Hansl-Kozanecki@cern.ch

Table 1 Document Status Sheet

Title:	GLAST offline software Raw data definition (Monte Carlo)		
ID:	[Document ID]		
Version	**Issue**	**Date**	**Reason for change**
0.7	5	04/02/02	Change MC to Mc in class names, eg McVertex, McParticle, etc.
0.6	5	11/25/00	Addition of Digi definitions for calorimeter, following discussion of the calo group; several class names changed; tidying up of text
0.5	5	11/11/00	Add discussion on Hit, Digi and channel identifiers
0.4	4	10/10/00	Decide on association mechanism for Hit and McParticle, add Technical remarks (Appendix)
0.1	3	02/02/00	First version

1 Introduction

This note is arranged as follows. Chapter 1 introduces the framework and the data model, and discusses the scope for the data definitions. Chapter 2 lists a summary of the requirements for the raw data. Chapter 3 discusses the Monte Carlo event and the raw event in more detail.

This note collects requirements, but gives also examples, which can be considered as ‘use cases’, and it proposes part of the implementation. To differentiate the importance, we follow the convention of requirements documents, where the meaning of ‘must’, ‘should’ and ‘may’ is defined as:

must	the feature has to be implemented,
should	if the feature is not implemented, this has to be justified,
may	implementation is desirable.

Appendix A discusses some technical points related to the standard template library (STL). Appendix B lists an example, the class definition of the Monte Carlo particle kinematics used for the Next Linear Collider Detector studies>may be replaced in later versions by the definitions chosen for GLAST<. This note is not intended to be a requirements document in the strict sense: it explains the requirements, but does not give an enumerated list to be followed up in future documents.

1.1 Model for data storage

We assume a framework, which implements the following design criteria:

- separation between ‘data’ and ‘algorithms’.
- separation between ‘persistent data’ and ‘transient data’.

The *data objects* will be limited to manipulations of internal data members. An *algorithm* will process *data objects* of some type and produce new data objects of a different type. Algorithms should not use directly the data in the persistency store but instead use transient objects. This is shown schematically in Figure 1. Algorithms communicate via the Transient Data Store (TDS). This architectural view was first promoted by LHCb (framework GAUDI) [1]>ref lhcbArchitecture<, and finds increasing acceptance in other experiments.

Data should be implemented by deriving from the `DataObject` base class. An algorithm should be designed as a class derived from the `Algorithm` base class. *Data services* provide the access to the transient data store for algorithms. *Converters* are responsable for retrieval and storage of the data objects between the persistency store and the transient store.

There are several types of data stores, which are distinguished mainly by the lifetime of their objects: event data, detector description (ideal geometry, numbering scheme, etc.), detector conditions (calorimeter calibration, alignment of tracker, dead and noisy channels, etc), parameters for algorithms, and histogramming and NTuple store.

Only objects derived from the DataObject base class may be placed directly within a data store. Most Monte Carlo objects and reconstructed objects will, however, for efficiency reasons be stored by putting a set of objects (themselves not derived from DataObject) into an object container, which is derived from DataObject. For the moment two 'concrete' container classes are implemented in the GAUDI framework: ObjectVector<T> and ObjectList<T>. For efficiency reasons only pointers are stored, and 'smart' data pointers facilitate the access to the data. For details see the GAUDI manual, Chapter 6 [2]>ref_gaudi<.

Objects which are to be registered into the transient store must be created on the heap (i.e. with the new operator). Once an object is registered into the store, the algorithm, which created it, looses the ownership. This is also true for objects, which are contained within other objects, such as those derived or instantiated from the ObjectVector class.

Different technologies could be chosen for the persistent data store. For the time being we envisage to use ROOT I/O [3]>ref_ROOT<. Other OODBs may replace ROOT or could exist in parallel, provided that the corresponding converters are available; working examples of the use of ROOT are the LHCb [2]>ref_gaudi< or the BaBar Kanga[4]>ref_kanga< access to data.

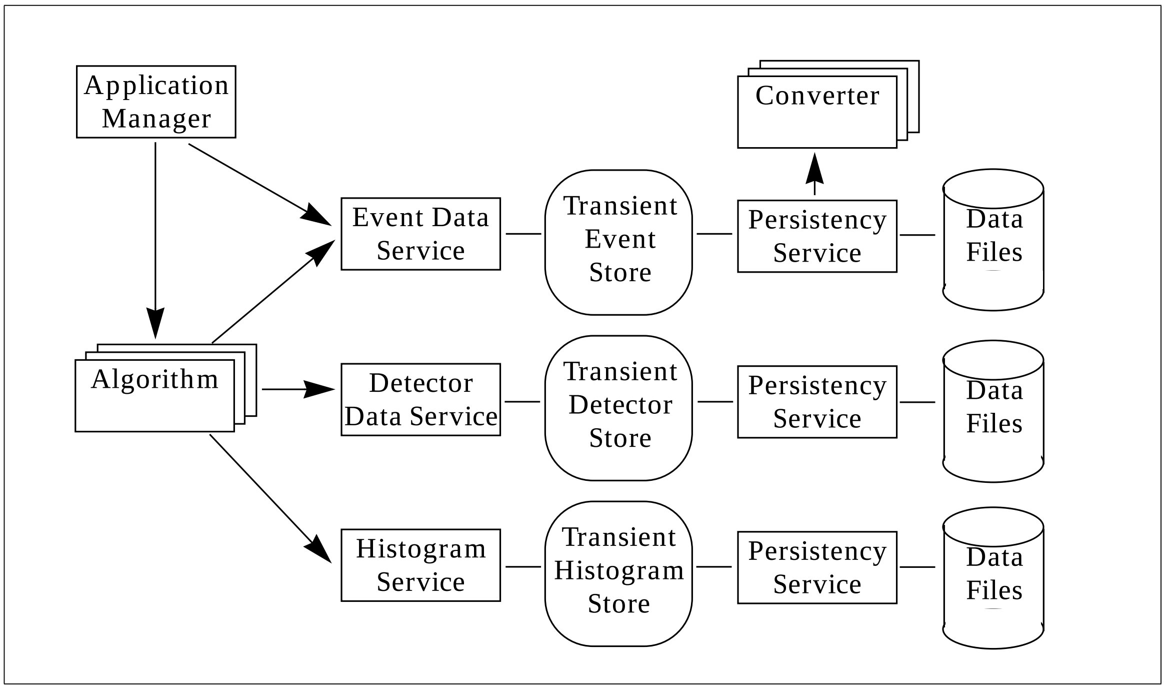

Figure 1 GAUDI object diagram: Access of algorithms to the transient data store via data services, and the conversion services between the transient store and the persistent store.

1.2 Scope

This note addresses the definition of the raw data, which are

- the output from the Monte Carlo detector simulation,
- the input to the reconstruction of simulated and real data,

- the input to the simulation of the level-1 trigger (LVL1) and the higher level triggers (HLT).

At present only the definition of data in the transient store will be addressed. The aim is to arrive at class definitions, which will be valid for the future software, though iterations will certainly be needed. This note does not discuss the format of the data in the readout buffers and in the persistency store. The requirements for these stores are different: whereas the transient store has to provide efficient access to the data for algorithms, the persistent store and the readout buffers require efficient packing to save memory space and minimize the amount of bytes to be transfered.

The testbeam data are a good testing ground for validating the class definitions. Though the essential class definitions should be available soon, the definitions will evolve with the experience gained from the testbeam.

For practical reasons it may be necessary to adapt the existing software in steps to the architecture presented here. Intermediate steps should then be documented also.

2 Requirements

This chapter collects the requirements for the raw data classes. Section 2.1 discusses the raw event and Section 2.1.1 the Monte Carlo (MC) event. The ‘hits’, from real data or data generated by Monte Carlo need identifiers, which describe their location in the detector. These identifiers should be ‘humanly readable’; they are discussed in Section 2.2.

The requirements formulated in the following subsections put constraints on the definition of the ‘raw’ data and on the MC propagator.

2.1 Raw event

The “raw event” contains the raw data collected by the data acquisition or generated by the simulation. The data are grouped per subdetector. If the data are generated by Monte Carlo, then a relationship to the particle(s), which caused the hit, should be provided for the diagnostic of the reconstruction algorithms.

2.1.1 Monte Carlo event

A schematic view of the *MC simulation* of the raw data is shown in Figure 2: >fig_MCsimulation< The *generator* produces particles according to the selected flux parameters. The *propagator* then tracks the particles through the spacecraft and the GLAST detector, possibly generating new particles in interactions or decays.

The MC event contains output data from the simulation: The particles, their origin and the signals in the sensitive parts of the detector. It should be possible to trace a signal hit back to the particle(s), which produced it, and to trace the particle to its ancesters, e.g. to the original particle from a flux generator. The volume over which the ancestry tree is recorded, may be settable via parameters (but has to be continuous).

Within the tracker volume each particle and vertex should be recorded, together with the association of hits to their particle of origin. This detail on hit association is also desirable for the spacecraft volumes surrounding the detector (more precisely, located between flux origin and detector).

The association of particles and hits should also be provided for charged tracks, which traverse (part of) the calorimeter without interaction. For em or hadronic showers in the calorimeter, however, it is neither useful nor practical to record individual trajectories. However, it is desirable to allow for recording the ‘hits’ as function of their ancestors, especially the electron and positron from the ‘first’ gamma conversion. This will help to study improvements of the association of calorimeter energy to tracks for the Kalman filter and the reconstruction of the gamma direction.

Some of the particles in a shower may be backscattered and reach the Anticoincidence Detector (ACD). It is desirable to distinguish the origin of ACD ‘hits’: ‘standard’ particles; backsplash from a shower in the calorimeter; or leakage of a shower, with possible interaction in the surrounding spacecraft and debris sent back into the ACD.

Hence several types of “origin” of the particle should be considered:

1. particles generated by the flux generator;
2. particles produced by interaction in the detector or its surrounding material;
3. decay products;
4. particles, which are ancestors of hadronic or electromagnetic (em) showers (showers, which originate in the calorimeter);
5. particles emerging as backsplash from a shower.

In the cases 1 to 3 the meaning of “particle” and “vertex of origin” is obvious. In case 4 the “ancestor” should be chosen depending on the type of the shower:

- if the particle entering the tracker is a γ and the γ is converted in the tracker volume, then use the electron or positron ancestor, when recording the energies in the calorimeter;
- for all other cases use as ancestor the particle, which initiated the shower in the calorimeter.

In case 5, the vertex of the particle is the point at which an anonymous shower particle enters the first volume for which backsplash has to be recorded. The particle acquires from this point on identity, and its parent is the ancestor of the shower.

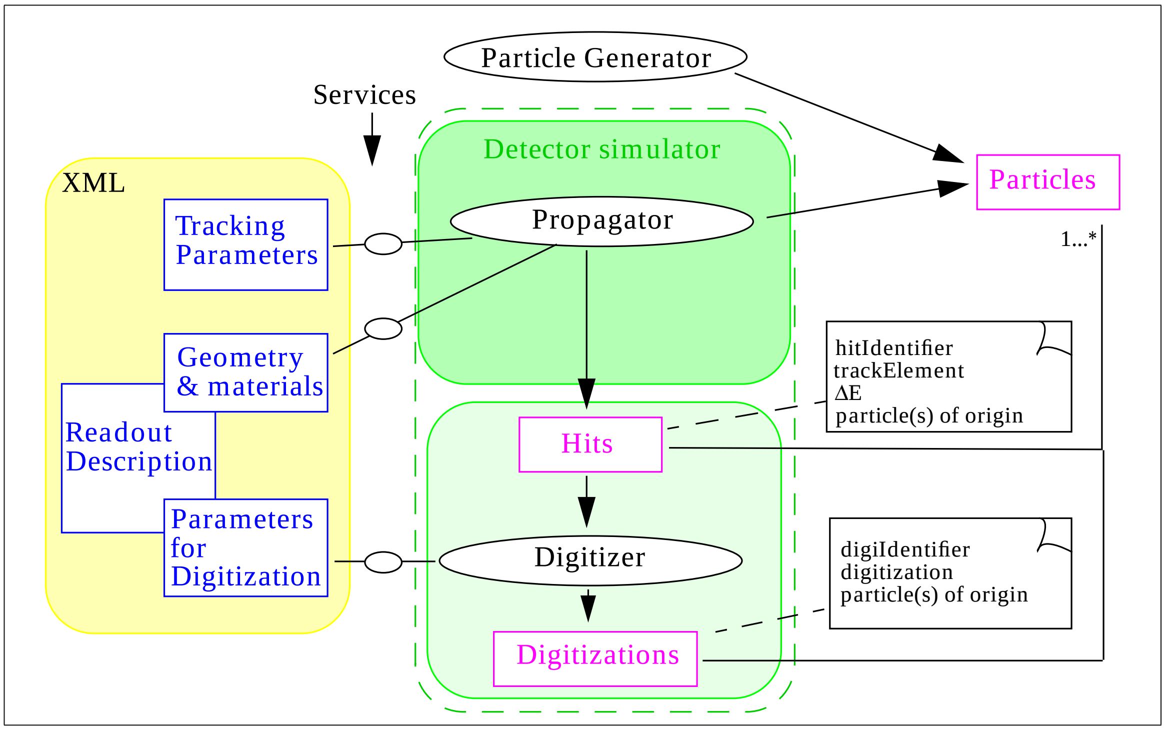

Figure 2 Detector simulation and its input and output. The relationship of particles to their ‘vertex’ of origin is not shown.

It is desirable that a probability can be assigned to a particle and propagated to the signals, which it produces. This may be an optional feature, controlled by settable parameters.

It should be possible to record hits in any volume of the detector, which is declared sensitive. This is useful to understand e.g. the loss of energy in dead material[1].

We have used the word “hit” until now in a sloppy way. The signal simulation should for several reasons proceed in two steps as sketched in Figure 2>fig_MCsimulation<. First a signal is simulated in an ideal detector. The resulting `Hit` object is then translated by the subdetector specific digitizer to the realistic digitization, the `Digi` object. The digitization merges `Hit` objects, adds electronic noise and applies uncertainties, resulting *e.g.* from calibration and alignment. Recording the signals at the level of `Hit` and `Digi`, allows to repeat the digitization for different parameters, starting from exactly the same `Hits`, e.g. to study the effect of different digitization parameters, or different noise levels. It also allows to add an additional signal stream (in `Hit` format); this is an efficient way of adding pre-generated background events.

MC data should provide information for diagnostic of the reconstruction. Such information is for example the position and direction of the track element in the sensitive volume and the deposited energy ΔE.

1. It is assumed that any volume can be declared as sensitive, i.e. as a volume, which registers Hit objects.

It should be possible to record the energy, which escapes the sensitive part of the detector. This can be achieved, either by surrounding the detector with a ‘dump’, which absorbs all energy, or by a ‘transparent dump’, which registers all energy, which passes through it. In both cases a granularity should be chosen, which is at least at the level of a tower and the major subsystems, calorimeter and tracker.

It should be possible to record the energy, which is deposited in dead material, for example between the towers. For this purpose several small volumes should be grouped into ‘dead material’ volumes, which collect the deposited energy like an active calorimeter.

It is not necessary to provide `Digi` classes for the case of `Hits` registered in dead material or the ‘dump’ volume.

2.2 Numbering scheme

Numbering scheme at different processing levels

It is desirable that the data format and grouping of data in the transient store is the same for offline reconstruction algorithms, for the HLT offline simulation and possibly for the HLT online implementation. The format and grouping of transient data should also, for efficiency reasons, be close to the one of the readout buffers. The data format is, however, likely to be not exactly the same, because code ultimately works with data mapped onto words of hardware- and language-dependent length. Numbering conventions should, however, be identical for the transient store and the persistent store, and, where practical, for the readout buffers.

Relationship to detector description

The numbering scheme for the readout channels is closely related to the *detector description* (DD), which consists of two separate groups of parameters: the detector *geometry description* and the parameters for *digitization*. >which additional info for *readout numbering*?< These parameter files should contain sufficient information such that in the simulation the channel identifiers can be generated in a data driven way (no hard coding of channel identifiers). Similarly, the channel identifier, together with the DD parameters should contain all information to position the channel in the detector local and global coordinate system.

Hierarchical structure of channel identifier

The identifiers of the readout channels should reflect the structure of the subdetectors, and the grouping of the subdetectors into subdetector parts. >give example?<

Numbering and global coordinate system

Channel numbers should increase with increasing value of the coordinates (x, y, z) or with increasing azimuth (ϕ) in the global system. Numbers should run from 0 to N.

Requirements from higher level trigger(s)

The data ordering and data format can have important consequences for the HLT-implementation. Therefore requirements as a result of the HLT algorithms should be given preference.

Requirements from calibrations

The grouping of channels should take into account issues of alignment and calibration, such that corrections can be applied in an efficient way to groups of channels.

2.3 Relationship between simulation and reconstruction

The Digi objects are the output of the simulation and the input to the reconstruction. Simulation and reconstruction should use the same external parameter files to encode and decode the digitizations and transform to positions in local or global space. This is shown schematically in Figure 3>fig_sim+recon<. Simulation and reconstruction may, however, need different converters of the parameter files. > <

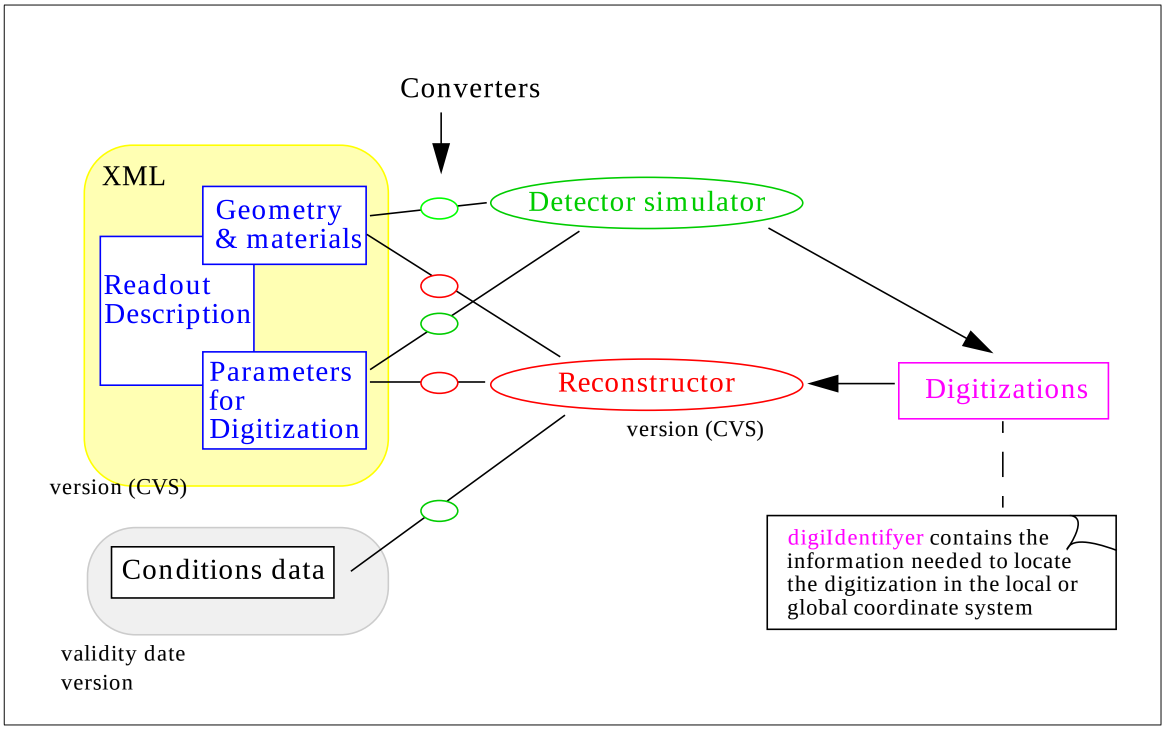

Figure 3 Parameters >fig_sim+recon< for simulation and reconstruction. Conditions data are *e.g.* data for calibration and alignment.

3 The (raw) event

To put the raw event and the MC event into context, we show in >fig_toplevel<Figure 4 a possible definition of the top event classes. The classes related to the raw and MC event classes are shown; the classes for the reconstructed event and the analysis event (not shown), are at the same level as the raw and MC event. The classes contain global run and event based information such as: runType, runNumber, eventNumber, timeStamp, triggerType, generatorType. Utility classes are used at several places: TimeStamp, RandomNumberSeed, ProcessingVersion. The latter identifies uniquely the release of the MC simulation code and the parameter files used.

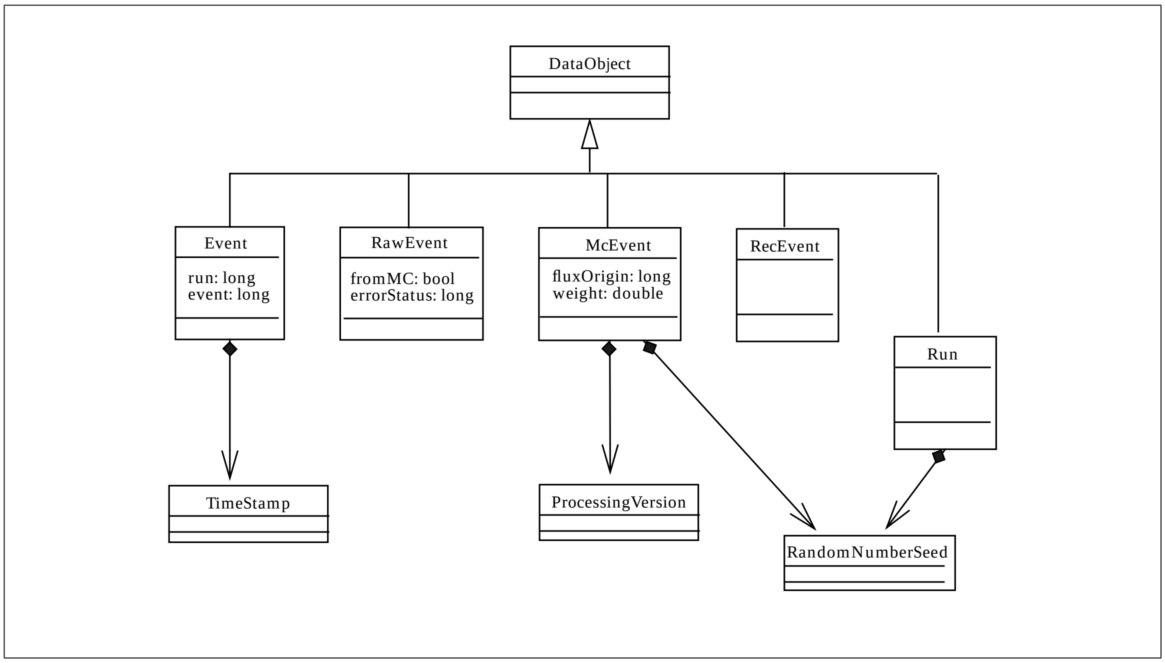

Figure 4 Class diagram >fig_toplevel< of the proposed top level event classes. The class for the analysis event is not shown. The class members shown are only indicative (inspired by [2]>ref_gaudi<, Figure A.1).

3.1 Monte Carlo event

The Monte Carlo event (Figure 5>fig_MCevent<) contains output data from the simulation. The data model consists of the classes, which describe the event kinematics (particles and pseudo-vertices), the MC hit and the MC digitization.

We distinguish two types of hits:

- Hits of type 'tracker hit' (McTkrHitBase) have a 1-to-1 relationship to their track of origin. Time of flight will be recorded. >ToF useful?<
- Hits of type 'calorimeter hit' (McCaloHitBase) may be caused by several 'tracks', hence there is a 1-to-n relationship between hit and tracks.

For the CsI calorimeter it is not practical to store the hits corresponding to each shower particle. To be able to repeat the digitisation, *i.e.* the generation of the signals recorded by the diodes, the following is proposed:

1. subdivide each crystal longitudinally into cells;
2. store per cell the sum of the deposited energy, the energy-weighted position and the second moment of the energy weighted position.
3. it is desirable that the information in 2) is available per ancestor.

We expect that the storage of the longitudinal position is sufficient, but foresee to store for special studies the three-dimensional position. The number of cells per crystal and their size (if cells have different size) should be settable parameters.

The CsI calorimeter, the diodes, and if required, the dead material and 'dump' hits are stored as calorimeter hits.

Few tracks will travers the ACDs, therefore the `Hit` objects can be stored like tracker hits. In case of the testbeam, the neutron counter hits can be stored the same way. The advantage is that the ACD `Hits` can be traced back to their origin, e.g. to an incoming particle or the backsplash from a shower. > < >fig_MCevent<

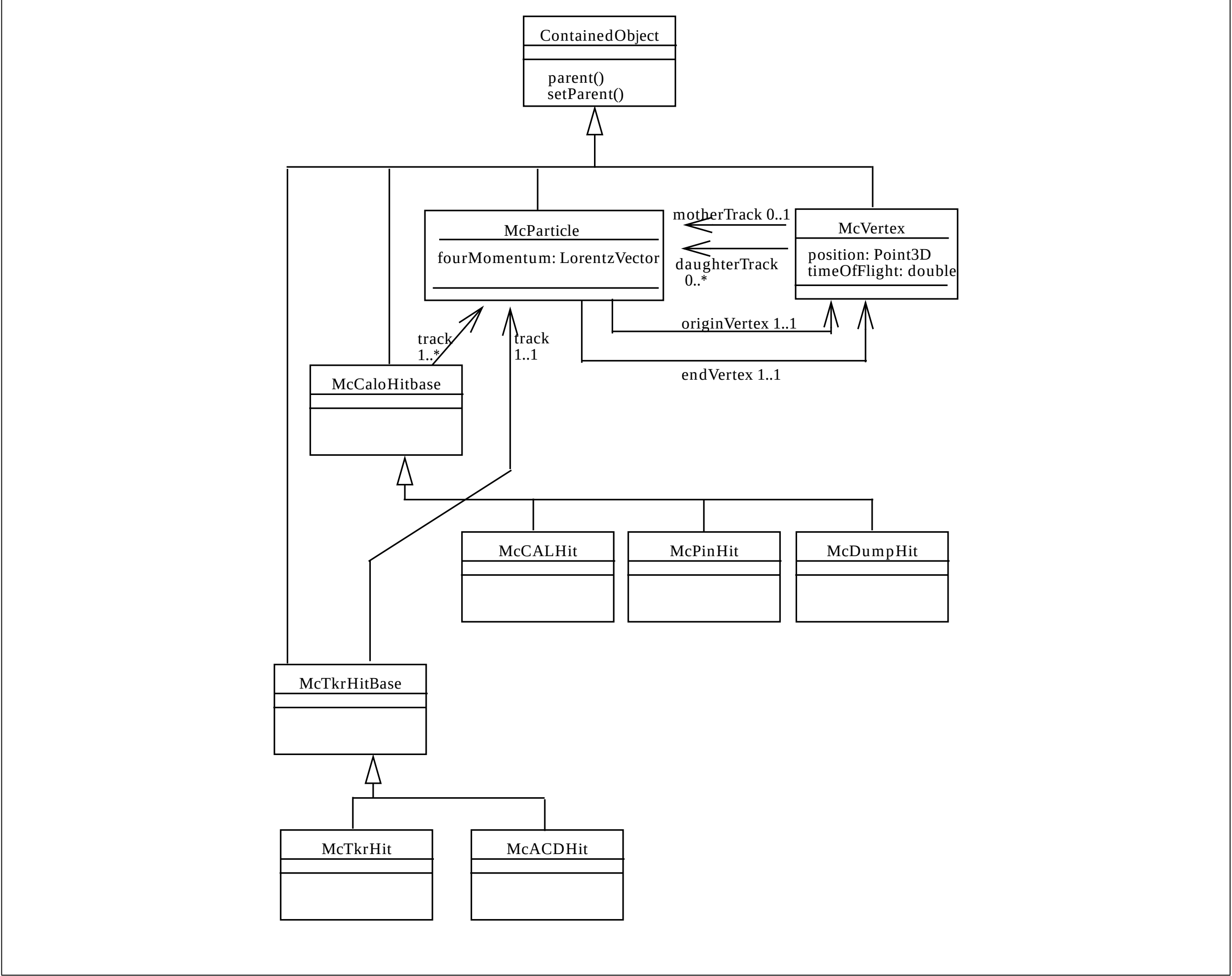

Figure 5 Definitions of the MC classes. >fig_MCevent< Only few of the class members are shown.

3.1.1 Monte Carlo kinematics

McParticle contains the data members: fourMomentum, particleId, and references to its origin-vertex, and end-vertex, where ‘vertex’ stands for a pseudo-vertex of the type listed in Section 2.1.1. In the following text this class may also be called shortly McParticle.

McVertex contains the data members: position, timeOfFlight, vertexType (see Section 2.1.1) and references to its mother track (pointer to McParticle) and its daughter tracks (vector of pointers to the class McParticle).

3.1.2 Monte Carlo Hits

All of the following classes inherit (directly or indirectly) from the class ContainedObject. >we may end up not using baseclasses <

3.1.2.1 ‘Calorimeter’ hits

McCaloHitBase is a hit base class for the ‘calorimeter’ hits. It contains the data members `cellId` and a vector of energy information. The energy information has two members: reference to the ancestor McParticle and the energy deposited by this McParticle. This class is common to four `Hit` classes: the energy deposited in the CsI calorimeter, the PIN diodes and (if requested) in the dead material and / or the “dump”.

McCALHit inherits from McCaloHitBase. In addition it contains as data members the energy-weighted first and second moments of the position, per ancestor.

McPinHit inherits from McCaloHitBase. No additional members are needed.

McDeadMaterialHit inherits from McCaloHitBase. No additional members are needed.

McDumpHit inherits from McCaloHitBase. No additional members are needed.

3.1.2.2 ‘Tracker’ hits

McTkrHitBase is a hit base class for the ‘tracker’ hits. It contains data members `entry` (entry point), `depositedEnergy` and `timeOfFlight`, and the reference track (pointer to McParticle). It is common to all of the following `Hit` classes. In contrast to the McCaloHitBase the deposited energy corresponds to a single track and there is a 1-to-1 relationship between Hit and McParticle.

McTkrHit inherits from McTkrHitBase. In addition it contains the data member `exit` (the exit point).

McACDHit inherits from McTkrHitBase. No additional members are needed.

3.1.3 Monte Carlo digitizations

The digitization information is the same for MC data and for true raw data (see Section 3.2), except that for MC digitizations there may in addition be the information, which relates Digi to ‘McParticles’. Different solutions can be envisaged to express this relationship, depending whether one insists on the following requirements:

1. The Digi classes for MC and true raw hits should be the same.
2. The Digi objects should be usable independently from the Hits objects.
3. It should be possible to mix MC and true raw data.

These requirements can be fulfilled, when using a STL map or by defining objects, which describe the association between two objects, e.g. the Hit and the McParticle. We propose to use the latter, because it is a more general solution, see Appendix A.2. >> details of implementation have still to be understood <<

3.1.3.1 `Hit`, `Digi` and channel identifiers

The requirements for the numbering scheme were discussed in Section 2.2. The `Hit` identifier can be built from the volume names and numbers, which are available during the MC propagation of the particle through the material. For example for a tracker `Hit`, the number of the tower, the layer and silicon wafer are known at the tracking step. The position of the `Hit` in the local coordinate system allows in the digitization step, to determine which strip(s) responded. The identification scheme is shown in Table 2. The corresponding readout channel number could be ‘identical’ to the Digi identifier (except for compression). If this is not possible, then a translation procedure has to be provided (look-up-table or function). The `Digi` identifier and the channel identifier are equivalent informations, but the channel identifier reproduces more realistically the raw data.

The identifiers used in the TDS should be classes, with methods to return the ‘elements’ which describe the Hit or Digi, e.g. method tower(), or methods to return the position in local or global space. ><

Table 2 Elements of the tracker `Hit` and `Digi` identifiers.

Tracker Object	Elements for Hit or Digi identification				
Hit	tower	layer	plane	ladder	wafer
Digi	tower	layer	plane	strip	

The digitization step may merge several `Hits` into one `Digi` (strip), or one `Hit` may result in more than one strip digitization. Normally the relationship between `Hits` and `Digi` will not be stored; if needed for special studies, it can either be stored in an `STL` multi-map or can be reconstructed from the `Hit` and `Digi` identifiers and the local `Hit` position.

The proposed identification for the CsI calorimeter is shown in Table 3. ><

Table 3 Elements of the calorimeter `Hit` and `Digi` identifiers.

Calo Object	Elements for Hit or Digi identification			
Hit	tower	layer	crystal	pseudo-cell
Digi	tower	layer	crystal	face

3.1.3.2 Ordering of Hits

`Hits` are generated per MC particle and along the trajectory. For later use by the digitization a different order is needed, which facilitates e.g. merging of the hits and storing of the Digis in an order similar to the order in which the readout buffers provide the raw data. This ordering can be achieved using the identifier as described in Section 3.1.3.1, and the local hit position. The hits should therefore preferably be stored in an ‘ordered collection’ (see Appendix A.1) or otherwise be ordered before digitization.

3.2 Raw event

The raw event contains the (quasi) raw data collected by the data acquisition. These classes contain the digitizations of the subdetectors: tracker, CsI calorimeter and ACD. Classes for detector conditions (position, orientation, temperature measurements, etc.) are not considered yet.

The data members listed in the next sections represent the data in the transient data store. All `Digi` classes inherit from ContainedObject. The containers presently available in Gaudi are `STL`-like vectors and lists.

The tables in the following sections list only the data members, or the elements from which a data member is built. The details are left to the implementation. For example, for the identifiers it is left open, whether integers or strings are used and how to combine the elements.

3.2.1 Digi for tracker

> the following is very tentative, just a start for the tracker group<

> Which container to choose?

1. one container for all strips, digi ordered per tower ,layer, strip. This gives most freedom: 1) iterator per tower and layer, 2) iterator per layer

   DigiId: needs all elements
2. container per layer -> DigiId has only to contain the strip no
3. vector of strips per layer

> How to store the association strip -> McParticle?

1. Map SmartRef<TkrDigi>(=key), SmartRef<McParticle>
2. SmartRef<McParticle> member of TkrDigi

<

The data are zero-suppressed. The information is readout per strip and per strip-layer. No clustering of adjacent strips is performed. The data are grouped per tower. All Digi are stored in one container and iterators provided per tower / layer / strip and per layer(all towers) / strip. The information per layer (TkrLayerDigi) is stored in a container and iterators provided per tower / layer. ><

Table 4 Data members of class TkrDigi for the silicon tracker data.

Type	Data member	Comment
TkrId	tkrId	see Table 2
SmartRef<McParticle>	rMcParticle	here or in separate map?

><

Table 5 Data members of class TkrLayerDigi for the silicon tracker data.

Type	Data member	Comment
TkrLayerId	tkrLayerId	see Table 2
int	tot	time over threshold
bool	statusController	true = OK

3.2.2 Digi for calorimeter

The data are grouped per tower, and within a tower by layer and crystal. For each end-face of the crystal all four ADC values (LEX4, LE, HEX8, HE) are recorded for the MC data together with the energy range, which the DAQ would choose for real data.

>zero supression or all data? If all data are transfered, then the position in the array determines the log number, no need for CalId<

><

Table 6 Class CalDigi for the CsI calorimeter data.

Type	Data member		Comment
CalId	calId		see Table 3
int	adc	[2][4]	pulse height in ADC counts, per face and each diode and gain
int	selRange	[2]	

3.2.3 Digi for ACD

>similar questions as for calorimeter, assume ordering according to Steves numbering<

><

Table 7 Class AcdDigi for the anticoincidence counter.

Type	Data member	Range of values	Comment
AcdId	acdId		
int	adc		

3.3 Preprocessed event

Preprocessing of the event is the first step in reconstruction. Application of calibrations, clustering of TKR hits, position corrections per tray, etc. are applied. The reconstruction can use the pre-processed data only, the raw data and the calibration files are no longer needed. If in addition an event filter is applied, the resulting data volume is in general much smaller.

4 References

1 *GAUDI, LHCb Data processing applications framework*,LHCb 98-064 COMP (Nov 1998) >ref_lhcbArchitecture<

2 *GAUDI User guide*, http://lhcb.cern.ch/computing/Components/html/GaudiMain.html >ref_gaudi<

3 *ROOT, An object oriented data analysis framework*, http://root.cern.ch/ >ref_ROOT<

4 *BaBar Analysis framework KANGA*, http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Kanga/index.html >ref_kanga<

5 ATLAS EDM Working group, *The StoreGate, A data Model for the ATLAS Software Architecture*, http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/Athena/edm/StoreGate.pdf

A Some technical points

A.1 Ordered collections

Ordering of sets can be achieved with the `STL` set collection and a user supplied ‘`Compare`’ template argument. Normally this second template argument defaults to a generic ‘less<>’, operating on the objects entered into the set. However, by defining the ‘`Compare`’ such that it returns true, if the first of two compared objects is ‘higher’ than the second, and false otherwise, the desired ordering can be achieved. With such a function, attached to the set, the compare is only executed at the time of inserting the object, and not when later iterating through the collection. Thanks to Christopher T.Day (CTDay@lbl.gov) for pointing this out.

A.2 Association of objects

To describe the association of objects, e.g. of `Hits` to the `McParticles` from which they originate, we propose to use ‘association objects’. For example an instance of a `McParticleAssociatedHit` would contain a pair of DataHandles referring to a `McParticle` and a `Hit` associated to it. This follows the discussion of Ref. [5]>ref_storegate<.

B MC kinematic information used by LCD

Listing B.1 Definition of the MC particle class McPart for the LCD experiment. The definition includes track and vertex (from R. Dubois). Note that this note proposes to have separate classes for particles and vertices.

```
// Class MCPart.h

#ifndef MCPART
#define MCPART

#include “TMath.h”
#include “TObject.h”
#include “TLorentzVector.h”
#include “TVector3.h”

class McPart: public TObject {
private:
  Int_t m_type;                 //  particle type number
  Int_t m_status;               //  status code
  McPart *m_parnt;             //  Pointer to McPart object which is parent of particle
  TLorentzVector* m_InitMom;    //  Initial momentum     (x,y,z,tot)
  TLorentzVector* m_TermPos;    //  Termination position (x,y,z,time)
  Float_t m_charge;             //  Particle charge

public:
  McPart();
  McPart(Int_t ID);
  McPart(McPart* part);
  ~McPart();



  void SetUpParticle(Int_t           type,
                     Float_t         charge,
                     Int_t           status,
     McPart*         parntptr,
                     TLorentzVector* tmom,
                     TLorentzVector* tpos);

  Int_t   GetType()    {return m_type;};
  Int_t   GetStatus()  {return m_status;};
  Float_t GetCharge()  {return m_charge;};
  McPart* GetParnt()   {return m_parnt;};

  void     SetMomentum(TLorentzVector* momentum);
  void     SetPosition(TLorentzVector* position);
  void     SetCharge(Float_t q)  {m_charge = q;};
  TLorentzVector* GetMomentum()  {return m_InitMom;};
  TLorentzVector* GetPosition()  {return m_TermPos;};

ClassDef(McPart,1)            // Monte Carlo particle object
};
#endif
```

Listing B.2 The implementation of the class McPart for the LCD simulation (from R. Dubois).

```
#include “McPart.h”
#include “TMath.h”

// The class McPart is the class containing the Monte Carlo particle
// information such as the particle ID, a pointer to the parent McPart
// object if it exists,the charge, the position and momentum initially
// and at the calorimeter. All particles recorded in the detector get an
// McPart object assigned to

ClassImp(McPart)

McPart::McPart() {
  // Default constructor
  m_type    = 0;
  m_parnt   = 0;
  for(int i = 0; i < 4; i++){
    m_InitMom = 0;
    m_TermPos = 0;
  }
}

McPart::~McPart() {
  // Default destructor
    delete m_InitMom;
    delete m_TermPos;
}

McPart::McPart(Int_t ID) {
  // Create an McPart object with article ID
  m_type = ID;
}

McPart::McPart(McPart* part){
  // Copy constructor, still has to be finished
  m_type   = part->GetType();
  m_status = part->GetStatus();
  m_parnt  = part->GetParnt();
  m_charge = part->GetCharge();
}

void McPart::SetUpParticle(Int_t type,Float_t charge,Int_t status,
   McPart* parntptr, TLorentzVector* tmom,
   TLorentzVector* tpos) {
  // Set all the attributes of the particle
  m_parnt  = parntptr;
  m_charge = charge;
  m_status = status;
  m_type   = type;
  SetMomentum(tmom);
  SetPosition(tpos);
}
```

Listing B.2 The implementation of the class McPart for the LCD simulation (from R. Dubois).

```
void McPart::SetMomentum(TLorentzVector *momentum){
  // Store the initial momentum of the particle
    m_InitMom = momentum;
}

void McPart::SetPosition(TLorentzVector *position){
  // Store the initial position of the particle
    m_TermPos = position;
}
```