STAR Offline Library Long Writeup

StMcEvent

User Guide and Reference Manual

Contents

Part I

User Guide

1 Introduction

StMcEvent[1] is a package that compliments the use of StEvent . The aim is for the user to be able to access and analyze Monte Carlo with the same object oriented approach as StEvent . The top class is *StMcEvent*, and a pointer to this class enables access to all the relevant Monte Carlo information. Again, the pointer is obtained through a Maker, in this case *StMcEventMaker*. An example invocation is found in sec. 7.

The information contained in StMcEvent is designed to be simply a reflection of the information already available in the existing g2t tables. The primary keys and foreign keys are replaced by associations between classes through pointers, like in StEvent . At the moment, not all the information found on the g2t tables is encapsulated in StMcEvent , for a full description of what is avaliable, please refer to sec. II. Also note that StMcEvent is a work in progress, and if additional information is needed, it can be added.

The goal of StMcEvent is to be used in conjunction with StEvent in order to have an OO model for both pure Monte Carlo data and DST data that has passed through the whole reconstruction chain. To be able to relate the 2 packages, there is an additional package that is run after both StEvent and StMcEvent are filled: StAssociationMaker . The aim of StAssociationMaker is to establish the relationships between the reconstructed and Monte Carlo information, so that users can easily check whether a particular reconstructed hit or track is found, and if that is the case, to directly obtain the associated Monte Carlo hit or track so that an assessment of the quality of the data can be made.

[1] We will adopt the convention used in StEvent and write StMcEvent when we refer to the package and *StMcEvent* when we refer to the class.

2 How to read this document

This document is divided in two parts, a user guide and a reference manual. In the first part we rather concentrate on basic questions and provide guidance to get started. Some of the information given here overlaps that one found in the StEvent documentation, and we provide it again here for completeness (We think that it is better to be redundant than to have to remember which information is in which manual). The reference section provides information on all available classes, their member functions and related operators. The class references contain one or more examples which demonstrate some features of each class and how to use them.

New users should **not** start with the Reference section. It is meant as a lookup when specific information is needed. Beginners should study the User Guide and should make themselves familiar only with those classes they will encounter more frequently: *StMcEvent* (sec. 10.6), *StMcTrack* (sec. 10.22), *StMcVertex* (sec. 10.23), *StMcTpcHit* (sec. 10.18). *StMcSvtHit* (sec. 10.13). and *StMcFtpcHit* (sec. 10.7).

Understanding the various examples certainly is the best way to get started. However, the examples are given for illustration purposes only, and are not complete programs.

3 Further documentation

StMcEvent makes use of various classes from the StarClassLibrary (SCL). To obtain the SCL documentation, the easiest thing to do is to go to the web page set up by Gene van Buren. Go to

```
http://www.star.bnl.gov/STARAFS/comp/root/special_docs.html
```

Here you can easily obtain the documentation several packages (this one for example).

4 Getting StMcEvent Sources

To access the complete source code proceed as follows:

StMcEvent is under **CVS** control at BNL. It can be accessed via *afs*:

1. Obtain an *afs* token: *klog -cell rhic.*
2. Make sure *$CVSROOT* is set properly:
   (i.e. *CVSROOT = /afs/rhic.bnl.gov/star/packages/repository*)
3. Check-out package into your current working directory:
   cvs checkout StRoot/StMcEvent

5 Coding Standards

StMcEvent tries to follow the STAR coding guidelines as described on the STAR → Computing → Tutorials → C++ coding standards web page:
http://www.star.bnl.gov/STAR/html/comp_l/train/standards.html.
Here we summarize the most relevant ones concerning the programmable interface:

- all classes, enumerations and functions start with the prefix **St**
- all member functions start with a lowercase letter
- header files have the extension **.hh**, source files have the extension **.cc**
- the use of underscores in names is discouraged
- classes, methods, and variables have self-explanatory English names and first letter capitalization to delineate words

StMcEvent also follows the following rules set forth in StEvent

- methods (member functions) which return a certain object have the same name as the referring object, i.e., without a preceding *get* prefix
  (as in *StMcEvent::primaryVertex()*)
- methods which assign a value to a data member (or members) carry the prefix *set* followed by the name of the member (as in *StMcEvent::setPrimaryVertex()*).
- integer variables which serve as counter or indices and never can take negative values are consistently declared as *unsigned*.
- Objects which are returned by pointer are not guaranteed to exist in which case a NULL pointer is returned. It is the user's responsibility to check for the return value and make sure the pointer is valid (non-zero) before she/he dereferences it. Objects which are guaranteed to exist are returned by reference or by value.

6 Conventions

6.1 Numbering Scheme

The information here is a reiteration of that found in the StEvent Manual. Both packages use the same conventions, but in order to stress the difference between the official STAR numbering scheme for subdetector components, and the C/C++ conventions for indexing the containers with which we represent those subdetectors, we repeat the cautions here. The C/C++ convention is that the first element in an array has the index 0. This applies to vectors, collections or other containers that allow indexing. The TPC sectors and padrows, the SVT layers, ladders and wafers, and the FTPC planes and sectors all start counting at 1.

So it is important to remember this when using the member functions that return the sector or padrow of a TPC hit, for example, to address the elements in a container. If you want to obtain the hit container in which a particular hit h is stored, you would do the following:

```
evt->tpcHitCollection()->sector(h.sector()-1)->padrow(h.padrow()-1).hits();
```

Note the subtraction of 1, in order to index the array properly. The functions that have this convention are:

- `StMcTpcHit::sector()`
- `StMcTpcHit::padrow()`
- `StMcFtpcHit::plane()`
- `StMcSvtHit::layer()`
- `StMcSvtHit::ladder()`
- `StMcSvtHit::wafer()`
- `StMcSvtHit::barrel()`
- `StMcRichHit::pad()`
- `StMcRichHit::row()`

As of October 2003, the volume Id of the FTPC hits also contains a sector. However the hits are still only contained by a collection based on the plane.

6.2 References and Pointers

Part of the goal of StMcEvent is the use of pointers and references as opposed to id's and foreign keys. However, the handling of pointers must be done with care. To reiterate wise words of caution: If a function returns an object by *reference*, the object is guaranteed to exist. If a function returns an object by *pointer*, then it is a good idea to always check if the pointer is `NULL`. Not doing this can be the cause of many a headache and sleepless nights. This is one of the most common "monsters under the bed" that can painfully bite, and crash code all over the place.

6.3 Units

All quantities in StMcEvent are stored using the official STAR units: cm, GeV and Tesla. In order to maintain a coherent system of units it is recommended to use the definitions in *SystemOfUnits.h* from the StarClassLibrary. They allow to 'assign' a unit to a given variable by multiplying it with a constant named accordingly (centimeter, millimeter, kilometer, tesla, MeV, ...). The value of the constants is thus that the result after the multiplication follows always the STAR system of units.

The following example illustrates their use:

```
double a = 10*centimeter;
double b = 4*millimeter;
double c = 1*inch;
double E1 = 130*MeV;
double E2 = .1234*GeV;

//
//   Print in STAR units
//
cout << "STAR units:" << endl;
cout << "a = " << a << " cm" << endl;
cout << "b = " << b << " cm" << endl;
cout << "c = " << c << " cm" << endl;
cout << "E1 = " << E1 << " GeV" << endl;
cout << "E2 = " << E2 << " GeV" << endl;

//
//   Print in personal units
//
cout << "\nMy units:" << endl;
cout << "a = " << a/millimeter << " mm" << endl;
cout << "b = " << b/micrometer << " um" << endl;
cout << "c = " << c/meter << " m" << endl;
cout << "E1 = " << E1/TeV << " TeV" << endl;
cout << "E2 = " << E2/keV << " keV" << endl;
```

The resulting printout is:

```
STAR units:
a = 10 cm
b = 0.4 cm
c = 2.54 cm
E1 = 0.13 GeV
E2 = 0.1234 GeV

My units:
a = 100 mm
b = 4000 um
c = 0.0254 m
E1 = 0.00013 TeV
E2 = 123400 keV
```

Further documentation can be found in the StarClassLibrary manual (see sec. 3).

6.4 Containers and Iterators

The containers used throughout StMcEvent

- are STL vectors (see below)6.4.1.
- store objects by pointer.

Because the are `vector`s, they allow random access as in:

```
pointer_to_object = container[index];
```

The containers are guaranteed to provide the following member functions: *size(), begin(), end().* All collections in StMcEvent have a referring iterator defined. The containers and iterators are all declared in the file `StMcContainers.hh`, for convenience. In addition, all the classes in StMcEvent are `#included` in the header `StMcEventTypes.hh`. The names given to the containers are meant to reflect the objects contained, as well as whether a container is a *structural* container or not. By a structural container we mean a container that owns the objects it contains. This means that when a structural container is deleted, the objects it holds get deleted as well. The naming convention is then:

- A **vec**tor of **p**oin**te**r**s** will carry the prefix `StPtrVec`
- A **s**tructural **vec**tor of **p**oin**te**r**s** will carry the prefix `StSPtrVec`

The name is completed with the type of the objects they contain. So the structural container of pointers to objects of type *StMcTpcHit* is *StSPtrVecMcTpcHit* where we drop the *St* of the object being contained.

In reality, whether a container is structural or not is of little importance in terms of its usage, the interface and methods of both are the same. The distinction is made to keep the objects organized, and to make sure everthing gets deleted, and deleted in only one place. But this mainly goes on behind the scenes. The main point one should keep in mind from all of this is that, unless you really know what you are doing, you should **NOT** delete a structural container. (Play music for "The Twilight Zone".)

In order to keep your code independent of the underlying container types, people are encouraged to use *iterators* instead of indices. These permit a higher degree of flexibility, allowing to change containers without changing the application code using them. The following example demonstrates this:

Given an arbitrary StMcEvent collection *anyColl* of type *StAnyColl* which holds objects of type *obj* the following code is not guaranteed to work:

```
for (int i=0; i<anyColl.size(); i++)
    obj = anyColl[i];
```

A `list` for example, does not allow random access, so indexing is not supported for `lists`. For vectors and deques, indexing does work. Iterators however, work for all STL containers. The code above then, should be replaced by:

```
StAnyCollIterator iter;
for (iter = anyColl.begin(); iter != anyColl.end(); iter++)
    obj = *iter;
```

The names of the iterators defined in StMcEvent are just the name of the object contained + the word `Iterator`. For example, the iterator for the container *StPtrVecMcVertex* is *StMcVertexIterator.*

Note that all collections are by pointer, so they are *polymorphic* containers; that is, they are not restricted to collect objects of the base type only, but can store objects of any type derived from the base. This is especially important when dereferencing the iterator to access the objects in the collection. Polymorphism is not

used in StMcEvent(at least not as much as it could be used), but it is necessary in StEvent, where, for example, one has containers of tracks, but they can be Global or Primary. If a method returns a base class and one really wants a derived class, we have to use Run Time Type Information (RTTI), `dynamic_cast`ing and such. Also, make sure you know how to get the objects you want, depending on the type of objects in the container.

For a *by-pointer* collection:

```
StMcVertexIterator iter;
StSPtrVecMcVertex& vertices = event->vertices();
StMcVertex* vertex;
for (iter = vertices.begin(); iter != vertices.end(); iter++)
    vertex = *iter;                          // dereference once
```

Hint: Use pointers or references to collection elements wherever possible. Making local copies is often time- and memory-intensive. For example the same code above but written as:

```
StMcVertexIterator iter;
StMcVertexCollection* vertices = event->vertexCollection();
StMcVertex vertex;
for (iter = vertices->begin(); iter != vertices->end(); iter++)
    vertex = **iter;                          // dereference twice
```

invokes the *StMcVertex* assignment operator and creates a local copy. This method is only useful if you intend to modify an object locally but want to leave the original untouched.

6.4.1 Quick glance at `vectors`

As mentioned before, all containers used in StMcEvent are STL `vectors`. That means that one can use the methods supported by the `vector` template class. I'll give a brief overview of some basic elements of `vectors` to provide some common ground on which to walk on, for people unfamiliar with the STL. A LOT more information can be found in your favorite C++ book.

Common Member Types The type of the elements of the container is passed as the first template argument, and is known as its `value_type`. For example, the `value_type` of the *StPtrVecMcTrack* container, which is really a `vector<StMcTrack*>` is, of course, `StMcTrack*`. The type used for indexing into the container is known as `size_type`, and `difference_type` is the type of the result of subtracting two iterators. Like mentioned before, every container defines an `iterator` and a `const_iterator` for pointing to the elements of the container. One of the aims of these types is to allow the possibility to write code without knowing the actual types involved.

Public Member Functions The following methods illustrate some of the methods provided by the `vector` container. The list is by no means complete, but even so there are some more methods listed here than normally used. The counterpart `const` methods are omitted.

Iterator Methods

`iterator begin()`
Points to the first element of the container.

`iterator end()`
Points to the last-plus-one element.

`reverse_iterator rbegin()`
Points to the first element of the reverse sequence. Note that the type of the iterator is different than for `begin()`. Useful when going through a sequence backwards. For more information on `reverse_iterator` consult your favorite C++ reference book.

`reverse_iterator rend()`
Points to the last-plus-one element of the reverse sequence.

The following methods allow to access the elements of the container.

Useful Access Methods

`reference operator[](size_type n)`
Provides unchecked access to the elements of the container. Safe to use when there is a previous condition to guarantee that the argument is within bounds.

`reference at(size_type n)`
Provides range checked access to the elements of the container.
Throws `out_of_range` if the index is out of range.

`reference front()`
Reference to the first element of the container.

`reference back()`
Reference to the last element of the container.

`size_type size()`
Number of elements in the container.

`void push_back(const T& x)`
Adds the element to the end of the container, `size()` increases by 1.

`void pop_back(const T& x)`
Removes the last element of the container, `size()` decreases by 1.

`void clear()`
Erases all the elements.

6.5 Particles from Event Generators

StMcEventalso provides information on particles coming from the so called “particle table”. This is where all the particles produced by the event generator are stored, with all the necessary bookkeeping.

The particle table is useful for the cases where a particle that one is interested in, has been decayed by the event generator and not in GSTAR. It will therefore not appear in the g2t_track table. In order to find out if a track from the g2t_track table comes from our particle of interest in this case, we need to access the particle table.

All entries in the particle table are kept in the same container as the tracks from the g2t_track table. (We probably shouldn't call them tracks because these particles will most definitely NOT leave a track in the detector simulation, but we keep them along with the others for convenience.) There will of course be tracks that appear in both tables, namely the final state particles of the event generator. This case is already accounted for in StMcEvent, so that only one entry with the relevant information from both tables is kept in the container.

To properly use the additional information, it is important to know how to distinguish an *StMcTrack* that comes from the particle table and one that doesn't. This can be done in several ways, according to the following properties (or conventions, if you will):

- **Tracks with an event generator label > 0 have a corresponding entry in the particle table.**
- **Tracks with a key > 0 have a corresponding entry in the g2t_track table.**
- **Only tracks from the g2t_track table are assigned a start vertex.** This also means that the method *StMcVertex::daughters()* will **NOT** return any tracks from the particle table. It is important to realize this if one wants to asks for the daughters of the primary vertex. This will only return the tracks that actually have the primary vertex as their parent, according to the start_vertex_p entry of g2t_track the table. (These tracks are most probably also in the particle table, this scheme will of course not exclude them. But the point is that tracks that only appear in the particle table will **NOT** have a start vertex and they will not appear as daughters of any vertex, *not even the primary*).
- **To find out the genealogy of a particle, the *StMcTrack::parent()* method is provided.** This method will return a pointer to the parent track, if there is one, otherwise it will return a null pointer. Using this method, one can get to the parent particle of a decay daughter all the way into the particle table. The parentage is followed not just until we reach the particle table, but it is also followed *within* the particle table, i.e. down to the quarks and gluons of the event generator. It should be stressed, of course, that this further parentage is *only* event generator bookkeeping, and should not be used for more than this. For g2t_tracks, one can also find daughters by going through their stop vertex (if there is one), which then knows about its daughter tracks.

Let's illustrate this with a simple example. Suppose that we have an event generator and we want to look at $\phi \longrightarrow e^+e^-$ and the ϕ was decayed by the event generator. To get to them, we can do the following.

1. We get a pointer to the primary vertex (*StMcEvent::primaryVertex()*).
2. We get the daughters of the primary vertex (*StMcVertex::daughters()*).
3. We loop over the primary tracks thus obtained (Look in the containers section 6.4).
4. For each of primary tracks, we check whether it is an electron (*StMcTrack::particleDefinition()*).
5. If it is, we check whether its parent is a ϕ (*StMcTrack::parent()*, followed by *StMcTrack::particleDefinition()*. Be careful with NULL pointers!).
6. If it is a ϕ, then we found it and we can do whatever we want with it, e.g. fill a 2-D histogram of $p_\perp$ vs. y.

For more information on the specific methods to do this, refer to the relevant sections for *StMcVertex* 10.23 and *StMcTrack* 10.22.

7 How to use StMcEvent: The Macro and the Maker

7.1 StMcEventReadMacro.C

StMcEvent has now been a package in the STAR library for a while. The new version which this guide refers to is, as of this writing, available in starnew and stardev. Remember, that libraries get moved, so make sure you are using the right library for your purposes. This is very important, since to use StMcEvent along with StEvent one CANNOT combine the latest code with files produced before December 1999, since the dst tables and StEvent changed substantially.

To run the example macro StMcEventReadMacro.C, one can just run the default version directly from the library:

```
stardev
cd somewhere
root4star -b
.x StMcEventReadMacro.C
```

This will run the macro in the repository with the default settings: Process one event from the default geant.root file, load StMcEvent and print out to the screen some information on the event. If this step doesn't work, then something weird is going on. (Although make sure if you are using dev that Lidia is not rebuilding the whole library while you are trying to run one of the examples!) The default macros are meant to work "out of the box", so to speak, so this is the first step to check that things are working properly.

The next step is to actually check out the macro and edit it yourself for your purposes. To check out the macro, do the following:

```
klog
mkdir workdir
cd workdir
cvs co StRoot/macros/examples/StMcEventReadMacro.C
```

StMcEventReadMacro.C is found in *$CVSROOT/StRoot/macros/examples/StMcEventReadMacro.C*, and it runs a chain runs a chain of just one maker to load StMcEvent.

At the moment, it runs the chain on a ROOT file tree. That is, it uses the GEANT branches of the files it finds in the directory. This is important to realize if you want to run the macro on files other than the default, an activity one should quickly graduate to. The macro takes as arguments the number of events to process, and the name of the file to be used. An example invocation is:

```
.x StMcEventReadMacro.C(10,"/star/rcf/test/dev/tfs_Linux/Mon/year_2a/hc_standard/*.geant.root")
```

This invocation will process 10 events from the specified file. Now, to open the files, branches, etc. we rely on StIOMaker. This maker takes the path to look for files from the specified file, but the files it actually opens depend on what branches are activated in the macro. This is done inside the macro with the command *SetBranch*.

7.2 StMcEventMaker

StMcEventMaker is the code that actually does the opening of the geant.root file and uses the information there to set up the whole StMcEvent. If you want to use StMcEvent you have to make sure that *StMcEventMaker* is instantiated in the chain you are running. This will take care of all the setup. The data members and methods of the maker that you will find useful are:

Synopsis

```
#include "StMcEventMaker.hh"
class StMcEventMaker;
```

Public Data Members

`Bool_t doPrintEventInfo;`
Print or do not print info on the current StMcEvent event. (default=kFALSE). This produces a lot of output, and is meant for debugging. Every major class is dumped, the sizes of all collections, and the first element in every container. Don't use it for production.

`Bool_t doPrintMemoryInfo;`
Switch on/off checks on memory usage of StMcEvent (default=kFALSE). In order to get a memory snapshot we use `StMemoryInfo` from the *StarClassLibrary*. A snapshot is taken before and after the setup of StEvent. The numbers in brackets refer to the difference. Not available on SUN Solaris yet.

`Bool_t doPrintCpuInfo;`
Switch on/off CPU usage (default=kFALSE). Tells you how long it took to setup StEvent. Timing is performed using `StTimer` from the StarClassLibrary.

`Bool_t doUseTpc;`
Load Tpc hits in the run (default = true). If you do not want to look at Tpc hits, switch this off.

`Bool_t doUseSvt;`
Load Svt hits in the run (default = true). If you do not want to look at Svt hits, switch this off.

`Bool_t doUseFtpc;`
Load Ftpc hits in the run (default = true). If you do not want to look at Ftpc hits, switch this off.

`Bool_t doUseRich;`
Load Rich hits in the run (default = true). If you do not want to look at Rich hits, switch this off.

`Bool_t doUseBemc;`
Load Bemc hits in the run (default = true). If you do not want to look at Bemc hits, switch this off. This loads the barrel towers.

`Bool_t doUseBsmd;`
Load Shower Max hits in the run (default = true). This loads hits from Bsmde, and Bsmdp detectors.

`Bool_t doUseCtb;`
Load CTB hits in the run (default = true). If you do not want to look at CTB hits, switch this off.

`Bool_t doUseTofp;`
Load TOFp hits in the run (default = true). If you do not want to look at TOFp hits, switch this off.

`Bool_t doUseTof;`
Load TOFr hits in the run (default = true). If you do not want to look at TOFr hits, switch this off.

`Bool_t doUsePixel;`
Load Pixel hits in the run (default = true). If you do not want to look at Pixel hits, switch this off. The pixel is under development (August 2003), so Pixel hits will not be found in general purpose simulation files for a while.

Public Member Functions

`StMcEvent* currentMcEvent();`
Returns a pointer to the current *StMcEvent* object.

Since the *StMcEvent* object you get when you invoke `currentMcEvent()` is held by pointer, dont forget to check if this is `NULL`. This would immediately signal something gone awry. Also, do not delete the objects you get from the StMcEvent methods, they will be deleted every time you read a new event.

In order to use the classes in StMcEvent in other Makers, one header file with *all* the include files is provided: `StMcEventTypes.hh`. Like with StEvent this makes life easier in terms of not having to remember which file you need where, you just care about one include file and that's it. However, convenience comes at a price, because if you #include all the files, you will certainly be creating dependencies on classes you won't need or care about. So if this is an issue, then proceed as usual and only #include the classes that you actually do depend on. The next step is the use of the StAssociationMaker package. This is described in its own manual.

8 Known Problems and Fixes

StMcEvent is changing as the software environment changes. From the last version, several things have been updated and the design revamped to follow StEvent . The creation of the vertex collection from the tables took its final form, once the g2t_vertex table was corrected. The g2t_event table was finally written out, so it is now available.

The FTPC and SVT associations between reconstructed and Monte Carlo hits are now included. Also, the problems StAssociationMaker had with the declaration of multimaps by the Solaris CC4.2 compiler (i.e. no Template Default Arguments, and ObjectSpace implementation of the STL) have been solved and now StMcEvent and StAssociationMaker compile and run on Linux gcc, Solaris CC4.2 and CC5 and HP aCC.

The particle table has also been added to StMcEvent, and this allows to find particles of interest that have been decayed by the event generator.

The switches to turn on/off the loading of the different detector hits have been added for convenience. The first version of the classes for the EMC hits implemented by Aleksei Pavlinov are now also included.

The TOF classes have been added in Aug 2003, as well as the development classes for the Pixel detector.

For corrections to this manual, typos, suggestions, etc. send an email to

`calderon@star.physics.yale.edu`.

Part II

Reference Manual

9 Global Constants

We use the constants defined in the two header files *SystemOfUnits.h* and *PhysicalConstants.h* which are part of the StarClassLibrary. The types defined therein are used throughout StMcEvent .

10 Class Reference

The classes which are currently implemented and available from the STAR CVS repository are described in alphabetical order.

Inherited member functions and operators are not described in the reference section of a derived class. Always check the section(s) of the base class(es) to get a complete overview on the available methods.

Note that some constructors are omitted, especially the ones which take tables as arguments. They are for internal use only (even if public).

Destructors, assignment operators and copy constructors are not listed. Macros and *Inline* declarations are omitted throughout the documentation, as well as the `virtual` keyword. For the most up-to-date reference of what is available, there is no substitute to looking directly at the class definition in the header file.

10.1 StMcCalorimeterHit

Summary *StMcCalorimeterHit* represents a Calorimeter hit.

Synopsis
```
#include "StMcCalorimeterHit.hh"
class StMcCalorimeterHit;
```

Description *StMcCalorimeterHit* does not inherit from *StMcHit.* This decision was taken by the EMC group based on the different functionality the Calorimeter Hit would need to have. For example, one of the main difference is that the calorimeter hits do not have a position in global coordinates. They do however, still keep a pointer to their parent track.

Persistence None

Related Classes The hits are kept according to module. They are stored by pointer in a container for each module (see 6.4). The way to get the hits for a certain module (0-119, 120 modules in total) starting from the top level *StMcEvent* pointer is: `mcEvt->emcHitCollection()->module(0)->hits()` Look in 10.4 and 10.5 for more information. Each instance of *StMcTrack* holds a list of Calorimeter hits which belong to that track. Depending on which detector they are, they will be in the appropriate containter: Bemc, Bprs, Bsmde, or Bsmdp. Look in 10.22 for more information.

Public Constructors

`StMcCalorimeterHit(int m,int e,int s,float de)`
Create an instance of *StMcCalorimeterHit* with module *p*, eta *e*, sub *s*, and energy deposition at hit *de.*

`StMcCalorimeterHit(int m,int e,int s,float de, StMcTrack* parent)`
Create an instance of *StMcCalorimeterHit* with module *p*, eta *e*, sub *s*, energy deposition at hit *de*, and parent track *parent.*

Public Member Functions

`int module() const;`
Returns the number of the module (1-120) in which a hit is found.

`int eta() const;`
Returns the eta in which a hit is found.

`int sub() const;`
Returns the sub in which a hit is found.

`float dE() const;`
Returns the energy deposition of the hit.

`StMcTrack*  parentTrack() const;`
Returns the pointer to the track which generated this hit.

Public Member Operators

`ostream&  operator<<(ostream& os, const StMcCalorimeterHit\&);`
Allows to write some relevant information directly to an output stream. `void operator`=(const StMcCalorimeterHit&);+

Allows to incrementally add energy to a calorimeter hit from another calorimeter hit. `void operator==(const StMcCalorimeterHit\&);` Compares two calorimeter hits, they're equal when the module, eta, sub and parent track pointers are found to be equal.

Examples

```
//
//  In this example, we use the methods of the class
//  to access the data.  For concreteness, we use them to fill
//  a Root histogram.
//
void
PositionOfHits(StMcTrack *track)
{
   StPtrVecMcCalorimeterHit hits = track->bemcHits();

   StMcCalorimeterHitIterator i;
   StMcCalorimeterHit* currentHit;
   TH2F* myHist = new TH2F("coords. MC","eta vs mod pos. of Hits",120, 1, 121, 100, -2, 2)
   for (i = hits.begin(); i != hits.end(); i++) {
      currentHit = (*i);
      myHist->Fill(currentHit->module(), currentHit->eta());
   }
}
```

10.2 StMcCtbHit

Summary *StMcCtbHit* represents a CTB hit.

Synopsis
```
#include "StMcCtbHit.hh"
class StMcCtbHit;
```

Description Like most of the hit classes, it represents the Geant hit its related detector, in this case the CTB.

Persistence None

Related Classes The class to hold these hits is *StMcCtbHitCollection*, which holds them in a flat array, no further hierarchy.

Public Constructors

`StMcCtbHit(const StThreeVectorF& x,const StThreeVectorF& p, const flo`
Create an instance of *StMcCtbHit* with position *x*, local momentum *p*, energy deposited *de*, pathlength *ds*, key and id for proper assignment of pointers and the pointer to the partent track..

`StMcCtbHit(g2t_ctf_hit_st*)`
Create an instance of *StMcCtbHit* from the geant tables, this is the standard way in *StMcEventMaker*.

Public Member Functions

`void get_slat_tray(unsigned int & slat, unsigned int & tray) const;`
Puts the corresponding slat and tray of the hit in the variables entered as arguments to the function.

`float tof() const;`
Returns the Time of Flight of the particle to the detector hit volume.

10.3 className

Summary

Synopsis
```
#include "className.hh"
class className;
```

Description

Persistence None

Related Classes

Public Constructors

Public Member Functions

Public Member Operators

Public Functions

Public Operators

Examples

```
//
//  What the example does
//
```

10.4 StMcEmcHitCollection

Summary *StMcEmcHitCollection* is a container for the modules of the EMC. The actual hits are stored according to the module they belong to.

Synopsis

```
#include "StMcEmcHitCollection.hh"
class StMcEmcHitCollection;
```

Description *StMcEmcHitCollection* is the first step down the hierarchy of hits for the EMC. It provides methods to return a particular module, and to count the number of hits in the EMC detector that it represents. The EMC is different than the rest of the detectors in that the same class is used for the Barrel Emc, the Pre-shower, and the Shower Max detectors (Eta and Phi).

Persistence None

Related Classes *StMcEmcHitCollection* has a container of type `StMcEmcModuleHitCollection` of size 120, each element represents a module. Look in *StMcEmcModuleHitCollection* 10.5. This class inherits from *TDataSet*, so that each hit collection also has a name to distinguish them (along with other inherited properties of *TDataSet*.

Public Constructors

`StMcTpcHitCollection();`
Create an instance of *StEmcHitCollection* and sets up the internal containers.

`StMcTpcHitCollection(char* name);`
Create an instance of *StEmcHitCollection*, where *name* is passed to the TDataSet constructor, and sets up the internal containers.

Enumerations

`enum EAddHit {kNull, kErr, kNew, kAdd};`
Enumerations for the return values of the addHit method to flag the different cases. See below.

Public Member Functions

`StMcEmcHitCollection::EAddHit addHit(StMcTpcHit*);`
Adds a hit to the collection (using *StMemoryPool* from the StarClassLibrary. If the hit is a new hit, kNew is returned. If the hit was already present and we just needed to add the information, kAdd is returned. If the hit has a wrong module or some other parameter that prevents its correct assignment, kErr is returned.

`unsigned long numberOfHits() const;`
Counts the number of hits of the for the referring EMC detector.

```
unsigned int  numberOfModule() const;
```
Returns the size of the module vector. Default is 120.

```
StMcEmcModuleHitCollection*       module(unsigned int);
```
Returns a pointer to the module specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples

```
//
// Look in StMcTpcModuleHitCollection for an example.
// These classes normally go hand in hand.
```

10.5 StMcEmcModuleHitCollection

Summary

StMcEmcModuleHitCollection is the class where the EMC hits are actually stored. It holds a vector of StMcCalorimeterHit pointers.

Synopsis

```
#include "StMcEmcModuleHitCollection.hh"
class StMcEmcModuleHitCollection;
```

Description

StMcEmcModuleHitCollection holds the EMC hits.

Persistence

None

Related Classes

StMcEmcModuleHitCollection has a container of type `StSPtrVecMcCalorimeterHit`, this means that this is the class that actually owns the hits. Look at the section on containers 6.4 for more information.

Public Constructors

`StMcEmcModuleHitCollection();`
Create an instance of *StEmcHitCollection* and sets up the internal container.

Public Member Functions

`unsigned long numberOfHits() const;`
Counts the number of hits of this module.

`float sum() const;`
Returns the summation of the energy deposition, *dE*, values of all the hits in the module.

`StSPtrVecMcEmcHit&  hits();`
Returns a reference to the container of the EMC hits.

Examples

```
//
// Print number of hits in EMC and plot 1D Histogram of eta of
// Hits
//
   StMcEmcHitCollection* emcHitColl = evt->bemcHits();
   cout << ‘‘There are :‘‘ << emcHitColl->numberOfHits() << ‘‘hits in the BEMC’’ << endl;
   TH1F* etaHist = new TH1F("etaHist","Eta pos. of Hits",100, -2, 2)
   for (unsigned int i=0; i<emcHitColl->numberOfModules(); i++) {
     StSPtrVecMcEmcHit& hits = emcHitColl->module(i)->hits();
     for (StMcEmcHitIterator i = hits.begin(); i != hits.end(); i++) {
       currentHit = (*i);
       myHist->Fill(currentHit->eta());
     }
   }
```

10.6 StMcEvent

Summary *StMcEvent* is the top class in the StMcEvent data model. It provides methods to access all quantities and objects stored in the g2t tables.

Synopsis
```
#include "StMcEvent.hh"
class StMcEvent;
```

Description Objects of type *StMcEvent* are the entry point to the g2t data. From here one can navigate to (and access) quantities stored in the g2t tables, but instead of going through foreign keys and Id's all relationships are established through pointers. The only `#include` file needed to access all the classes of StMcEvent is the `StMcEventTypes.hh` file. *StMcEvent* itself doesn't offer much functionality but rather serves as a container for the event data. Deleting *StMcEvent* deletes the whole data tree, i.e. all depending objects are deleted properly.

Persistence None

Related Classes None

Public Constructors

`StMcEvent();`
Default constructor. Creates an "empty" event. Normally not used. As with most other classes, the constructor used normally is based on g2t tables.

Public Member Functions

`unsigned long eventGeneratorEventLabel() const;`
Event Label from event generator, read from g2t_event.

`unsigned long eventNumber() const;`
Event number from monte carlo production, read from g2t_event.

`unsigned long runNumber() const;`
Run number, read from g2t_event.

`unsigned long zWest() const;`
Number of protons of particle coming from the "West", read from g2t_event.

`unsigned long nWest() const;`
Number of neutrons of particle coming from the "West", read from g2t_event.

`unsigned long zEast() const;`
Number of protons of particle coming from the "East", read from g2t_event.

`unsigned long nEast() const;`
Number of neutrons of particle coming from the "East", read from g2t_event.

`unsigned long eventGeneratorFinalStateTracks() const;`
As the name implies, this has the number of final stated tracks coming from the event generator, read from g2t_event.

`unsigned long numberOfPrimaryTracks() const;`
Number of primary tracks read from g2t_event. Note that this is not coming from `primaryVertex()->daughters().size()` so one can check whether these

numbers are actually the same (they should be). Note that the tracks that only appear in the particle table are NOT assigned either a start or a stop vertex, and they are NOT kept as daughters of the primary vertex either.

`unsigned long subProcessId() const;`
This is used for simulated Pythia pp events, where one can study a given reaction, e.g. gg-¿QQbar = 82, and gg-¿J/Psi+g = 86. Refer to the Pythia manual for the subprocess ids. Read from g2t_event.

`float impactParameter() const;`
Impact parameter of collision, read from g2t_event.

`float phiReactionPlane() const;`
Phi angle of the reaction plane of the collision, read from g2t_event.

`float triggerTimeOffset() const;`
Trigger time offset, read from g2t_event.

`unsigned long nBinary() const;`
Returns the number of binary collisions. Read from g2t_event.

`unsigned long nWoundedEast() const;`
`unsigned long nWoundedWest() const;`
Return the number of binary of wounded nucleons from each collision direction. Read from g2t_event.

`unsigned long nJets() const;`
Number of jets in the event. Read from g2t_event.

`StMcVertex* primaryVertex();`
Returns a pointer to primary vertex. The same object is stored in the *StMcVertexCollection*. Since the primary vertex is of high importance for most analysis steps, this method was added for convenience. Note that if no primary vertex is available this function returns a `NULL` pointer, so be careful!

`StSPtrVecMcVertex& vertices();`
Returns a reference to the vertex container, i.e., the `vector` of all event vertices.
`StSPtrVecMcTrack& tracks();`
Returns reference to the track container, i.e., the `vector` of all monte carlo tracks.

`StMcTpcHitCollection* tpcHitCollection();`
Returns a pointer to the TPC hit collection. Note that the collections are *N*OT just flat containers. The TPC hit collection is a container of sectors. The sectors are containers of padrows, where the TPC hits are stored.

`StMcSvtHitCollection* svtHitCollection();`
Returns a pointer to the SVT hit collection. The SVT hit collection is a container of layers. The layers are containers of ladders. The ladders are containers of wafers, where the SVT hits are stored.

`StMcFtpcHitCollection* ftpcHitCollection();`
Returns a pointer to the FTPC hit collection. The FTPC hit collection is a container of planes. Note that it is in the planes where the FTPC hits are stored. The

Monte Carlo hits of the FTPC know nothing about sectors, because these are only determined after reconstruction.

`StMcRichHitCollection* richHitCollection();`
Returns a pointer to the RICH hit collection. This is a flat collection of hits.

`StMcEmcHitCollection* bemcHitCollection();`
Returns a pointer to the BEMC hit collection. The calorimeter hit collection is containers of modules. This same class is used for the other calorimeter detector components, below.

`StMcEmcHitCollection* bprsHitCollection();`
Returns a pointer to the Barrel Pre-shower hit collection.

`StMcEmcHitCollection* bsmdeHitCollection();`
Returns a pointer to the Barrel Shower Max detector-Eta hit collection.

`StMcEmcHitCollection* bsmdpHitCollection();`
Returns a pointer to the Barrel Shower Max detector-Phi hit collection.

The following member functions are used to set data members of *StMcEvent.* They are shown only for completeness and shouldn't be used without a profound understanding of the relations between the different objects.

```
void setEventGeneratorEventLabel(unsigned long);
void setEventNumber(unsigned long);
void setRunNumber(unsigned long);
void setZWest(unsigned long);
void setNWest(unsigned long);
void setZEast(unsigned long);
void setNEast(unsigned long);
void setImpactParameter(float);
void setPhiReactionPlane(float);
void setTriggerTimeOffset(float);
void setPrimaryVertex(StMcVertex*);
void setTrackCollection(StMcTrackCollection*);
void setTpcHitCollection(StMcTpcHitCollection*);
void setSvtHitCollection(StMcSvtHitCollection*);
void setFtpcHitCollection(StMcFtpcHitCollection*);
void setRichHitCollection(setRichHitCollection);
void setBemcHitCollection(StMcEmcHitCollection);
void setBprsHitCollection(StMcEmcHitCollection);
void setBsmdeHitCollection(StMcEmcHitCollection);
void setBsmdpHitCollection(StMcEmcHitCollection);
void setVertexCollection(StMcVertexCollection*);
```

Public Member Operators

`int operator==(const StMcEvent&) const;`
Compares the event number, run number and type to determine whether 2 events are equal.

`int operator!=(const StMcEvent&) const;`
Uses `operator==` and negates it.

`ostream&  operator<<(ostream& os, const StMcEvent&);`
Allows to write some relevant information directly to an output stream.

Examples **Example:**

```
//
//   Prints some basic event quantities to stdout (cout).
//
//
void printEvent(StMcEvent *event)
{

     cout << "Event Generator Label = "
          <<  event->eventGeneratorEventLabel() << endl;
     cout << "Event Number = "
          <<  event->eventNumber() << endl;
     cout << "Run number = " << event->runNumber();
     cout << "# of tracks = "
          << event->trackCollection()->size() << endl;
     cout << "# of vertices = "
          << event->vertexCollection()->size() << endl;
     cout << "# of tpc hits = "
          << event->tpcHitCollection()->size() << endl;
     cout << "xyz of primary Vertex = "
          <<  event->primaryVertex()->position()/millimeter
          << " mm" << endl;
}
```

10.7 StMcFtpcHit

Summary *StMcFtpcHit* represents a FTPC hit.

Synopsis
```
#include "StMcFtpcHit.hh"
class StMcFtpcHit;
```

Description *StMcFtpcHit* inherits most functionality from *StMcHit.* For a complete description of the inherited member functions see *StMcHit* (sec. 10.10).

In the StMcEvent data model each track keeps references to the associated hits and vice versa. All hits have a pointer to their parent track. This is especially important for the making of the associations in StAssociationMaker.

Persistence None

Related Classes *StMcFtpcHit* is derived directly from *StMcHit.* The hits are kept according to plane. They are stored by pointer in a container for each plane (see 6.4). The way to get the hits for a certain plane (1-20) starting from the top level *StMcEvent* pointer is: `mcEvt->ftpcHitCollection()->plane(0)->hits()` Look in 10.8 and 10.9 for more information. Each instance of *StMcTrack* holds a list of FTPC hits which belong to that track.

Public Constructors
```
StMcFtpcHit(const StThreeVectorF& x,const StThreeVectorF& p,
          const float de, const float ds, const long key,
          const long id, StMcTrack* parent);
```
Create an instance of *StFtpcHit* with position *x*, local momentum *p*, energy deposition at hit *de*, path length (within padrow) *ds*, primary key *key*, volume Id *id*, and parent track *parent.* Not used. Standard use is to obtain all parameters from g2t_fpt_hit table.

Public Member Functions
```
unsigned long plane() const;
```
Returns the number of the plane (1-10 for West Ftpc, 11-20 for East Ftpc) in which a hit is found. `unsigend long sector() const;`
Returns the sector (1-6) in which a hit is found. The volume Id to allow this was introduced on Oct. 2003.

Public Member Operators
```
ostream&  operator<<(ostream& os, const StMcFtpcHit\&);
```
Allows to write some relevant information directly to an output stream.

Examples
```
//
//  In this example, we use the methods of the class
//  to access the data.  For concreteness, we use them to fill
//  a Root histogram.
//
void
PositionOfHits(StMcTrack *track)
{
   StPtrVecMcFtpcHit hits = track->ftpcHits();
```

```
   StMcFtpcHitIterator i;
   StMcFtpcHit* currentHit;
   TH2F* myHist = new TH2F("coords. MC","X vs Y pos. of Hits",100, -150, 150, 100, -150, 150
   for (i = hits.begin(); i != hits.end(); i++) {
      currentHit = (*i);
      myHist->Fill(currentHit->position().x(), currentHit->position().y());
   }
}
```

10.8 StMcFtpcHitCollection

Summary

StMcFtpcHitCollection is a container for the planes of the FTPC. The actual hits are stored according to planes.

Synopsis

```
#include "StMcFtpcHitCollection.hh"
class StMcFtpcHitCollection;
```

Description

StMcFtpcHitCollection is the first step down the hierarchy of hits for the FTPC. It provides methods to return a particular plane, and to count the number of hits in all the FTPC.

Persistence

None

Related Classes

StMcFtpcHitCollection has a container of type `StMcFtpcPlaneHitCollection` of size 20, each element represents a plane.

Public Constructors

`StMcFtpcHitHitCollection();`
Create an instance of *StFtpcHitCollection* and sets up the internal containers.

Public Member Functions

`bool addHit(StMcFtpcHit*);`
Adds a hit to the collection (using *StMemoryPool* from the StarClassLibrary.

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole FTPC.

`unsigned int  numberOfPlanes() const;`
Returns the size of the Array of planes. Default is 20.

`StMcFtpcPlaneHitCollection*  plane(unsigned int);`
Returns a pointer to the plane specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples

```
//
// Look in StMcFtpcPlaneHitCollection for an example.
// These classes normally go hand in hand.
```

10.9 StMcFtpcPlaneHitCollection

Summary *StMcFtpcPlaneHitCollection* is the class where the FTPC hits are actually stored.

Synopsis

```
#include "StMcFtpcPlaneHitCollection.hh"
class StMcFtpcPlaneHitCollection;
```

Description *StMcFtpcPlaneHitCollection* holds the FTPC hits. Note the difference with StEvent. The Monte Carlo doesn't know about the sectors of the FTPC, these are determined during reconstruction.

Persistence None

Related Classes *StMcFtpcPlaneHitCollection* has a container of type `StSPtrVecMcFtpcHit`, this means that this is the class that actually owns the hits. Look at the section on containers 6.4 for more information.

Public Constructors

```
StMcFtpcPlaneHitCollection();
```

Create an instance of *StFtpcHitCollection* and sets up the internal container.

Public Member Functions

```
unsigned long numberOfHits() const;
```

Counts the number of hits of this plane.

```
StSPtrVecMcFtpcHit&       hits();
```

Returns a reference to the container of the FTPC hits.

Examples

```
//
// Print number of hits in FTPC and plot 1D Histogram of r (cylindrical coord.) of
// Hits
//
   StMcFtpcHitCollection* ftpcHitColl = evt->ftpcHits();
   cout << ''There are :'' << ftpcHitColl->numberOfHits() << ''hits in the FTPC'' << endl;
   TH1F* myHist = new TH1F("Sector 10 ","R pos. of Hits",100, 0, 100)
   for (unsigned int i=0; i<ftpcHitColl->numberOfPlanes(); i++) {
     StSPtrVecMcFtpcHit& hits = ftpcHitColl->plane(i)->hits();
     for (StMcFtpcHitIterator i = hits.begin(); i != hits.end(); i++) {
       currentHit = (*i);
       myHist->Fill(currentHit->position().perp());
     }
   }
```

10.10 StMcHit

Summary *StMcHit* is the base class of all TPC, FTPC, SVT and RICH monte carlo hit classes.

Synopsis
```
#include "StMcHit.hh"
class StMcHit;
```

Description *StMcHit* provides the basic functionality for all derived classes which represent TPC, FTPC SVT and RICH hits. It provides information on position, energy deposition, path length within sensitive volume, and the track which generated this hit. It also has the primary key from the g2t tables and the volume Id, which are useful for debugging purposes. All information is taken from the appropriate *g2t_xxx_hit.idl* table, where xxx can be "tpc", "svt", "ftp", or "rch". *StMcHit* is a virtual class.

Persistence None

Related Classes The following classes are derived from *StMcHit*:

- *StMcTpcHit*
- *StMcFtpcHit*
- *StMcSvtHit*
- *StMcRichHit*

Public Constructors

```
StMcHit();
```
Constructs an instance of *StMcHit* with all values initialized to 0 (zero). Not used. Normally, all hits belong to a certain detector, so the appropriate table based constructor is used.

```
StMcHit(const StThreeVectorF& x, const StThreeVectorF& p,
      float de, float ds,  long k, long volId, StMcTrack* parent);
```

Create an instance of *StMcHit* with position *x*, local momentum *p*, energy deposition *de*, path length *ds*, primary key *k*, volume id *volId* and parent track *parent*. Used by the derived classes to initialize the base class data members.

Public Member Functions

```
const StThreeVectorF& position() const;
```
Hit position in global coordinates.

```
const StThreeVectorF& localMomentum() const;
```
Local momentum direction of the parent track at the position of the hit.

```
float dE() const;
```
Energy deposition.

```
float dS() const;
```
Path length within sensitive volume.

```
long key() const;
```
Primary key from the appropriate g2t table.

`long volumeId() const;`
Geant volume Id, without any decoding.

`StMcTrack* parentTrack() const;`
Pointer to the track that generated the hit.

The following functions should not normally be used. The relevant data members are set during construction or in *StMcEventMaker* so these are provided for completeness.

`void setPosition(const StThreeVectorF&);`
Set the hit position.

`void setLocalMomentum(const StThreeVectorF&);`
Set the local momentum.

`void setdE(float);`
Set the total energy deposition.

`void setdS(float);`
Set the path length within sensitive volume.

`void setKey(long);`
Set the primary key.

`void setVolumeId(long);`
Set the volume Id.

`void setParentTrack(StMcTrack*);`
Set the parent track.

Global Operators

`int operator==(const StMcHit&) const;`
Checks to see if 2 hits are equal. Uses position, dE and dS for the comparison.

`int operator!=(const StMcHit&) const;`
Uses operator== and negates the result.

`ostream&  operator<<(ostream& os, const StMcHit&);`
Prints hit data (position, dE, dS) to output stream *os.*

Examples

```
//
//  Create a random hit and print it.
//
void printArbitraryHit()
{
   HepJamesRandom engine;
   RandFlat       rndm(engine);
   RandGauss      rgauss(engine);
   const double   sigmadE = 1e-06;
   const double   sigmadS = 7*millimeter;
   StMcTrack* pTrack = new StMcTrack;

   StThreeVectorF pos(rndm.shoot(-meter,meter),
                            rndm.shoot(-meter,meter),
                            rndm.shoot(-meter,meter));
   StThreeVectorF mom(rndm.shoot(-GeV,GeV),
                            rndm.shoot(-GeV,GeV),
                            rndm.shoot(-GeV,GeV));
```

```
   float de = rgauss.shoot(2.0e-06,sigmadE);
   float ds = rgauss.shoot(1.6,sigmadS);
   StMcHit hit(pos, mom, de, ds, 0, 0, pTrack);

   cout << hit << endl;
}
```

Programs Output:

```
Id: 0
Position: -20.3111  -80.7543 76.8227
Local Momentum: .9042 -.4982 -.1873
dE: 1.0952e-06
dS: 1.3049
Vol Id: 0
```

10.11 StMcRichHit

Summary *StMcRichHit* represents a monte carlo RICH hit.

Synopsis

```
#include "StMcRichHit.hh"
class StMcRichHit;
```

Description *StMcRichHit* inherits most functionality from *StMcHit.* For a complete description of the inherited member functions see *StMcHit* (sec. 10.10).

In the StMcEvent data model each track keeps references to the associated hits and vice versa. All hits have a pointer to their parent track. This is especially important for the making of the associations in StAssociationMaker.

Persistence None

Related Classes *StMcRichHit* is derived directly from *StMcHit.* The hits are kept according to row and pad. The hits are kept in an instance of *StSPtrVecMcRichHit* where they are stored by pointer (see 6.4). Each instance of *StMcTrack* holds a list of RICH hits which belong to that track.

Public Constructors

```
StMcRichHit(const StThreeVectorF& x,const StThreeVectorF& p,
      const float de, const float ds, const long key, const long id, StMc
```

Create an instance of *StMcHit* with position *x*, local momentum *p*, energy deposition *de*, path length *ds*, primary key *key*, volume id *id* and parent track *parent.*

Public Member Functions

```
unsigned short  pad() const;
```

Returns the number of the pad in which a hit is found.

```
unsigned short  row() const;
```

Returns the number of the row in which a hit is found.

```
float  tof() const;
```

Returns the time of flight from the g2t_rch_hit table.

Examples

```
//
//  Function which returns the RICH hits
//  used by a track and prints them.
//
void getHits(StMcTrack *track)
{
    StPtrVecMcRichHit& hits =
                    track->richHits();
    StMcRichHitIterator iter;
    StMcRichHit* hit;
    for(iter = hits->begin();
        iter != hits->end(); iter++) {
        hit = *iter;
        cout << *hit << endl;
    }

}
```

10.12 StMcRichHitCollection

Summary

StMcRichHitCollection is a container for the hits of the RICH. There is no hierarchy, this class holds the vector of hits.

Synopsis

```
#include "StMcRichHitCollection.hh"
class StMcRichHitCollection;
```

Description

StMcRichHitCollection holds the hits and the methods to add and count them.

Persistence

None

Related Classes

StMcRichHitCollection has a container of type `StSPtrVecMcRichHit` which holds the hits.

Public Constructors

`StMcRichHitCollection();`
Create an instance of *StRichHitCollection* and sets up the internal container.

Public Member Functions

`bool addHit(StMcRichHit*);`
Adds a hit to the collection (using *StMemoryPool* from the StarClassLibrary.

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole RICH.

`StSPtrVecMcRichHit& hits();`
Returns a reference to the vector of hits. This function also has a `const` version.

Examples

```
//
// Look in any of the other hit collections, the usage is identical.
//
```

10.13 StMcSvtHit

Summary *StMcSvtHit* represents a monte carlo SVT hit.

Synopsis

```
#include "StMcSvtHit.hh"
class StMcSvtHit;
```

Description *StMcSvtHit* inherits most functionality from *StMcHit.* For a complete description of the inherited member functions see *StMcHit* (sec. 10.10).

In the StMcEvent data model each track keeps references to the associated hits and vice versa. All hits have a pointer to their parent track. This is especially important for the making of the associations in StAssociationMaker.

Persistence None

Related Classes *StMcSvtHit* is derived directly from *StMcHit.* The way to get the hits for a certain layer-ladder-wafer starting from the top level *StMcEvent* pointer is: `mcEvt->svtHitCollection()->l`
The hits are kept in an instance of *StSPtrVecMcSvtHit* where they are stored by pointer (see 6.4). Each instance of *StMcTrack* holds a list of SVT hits which belong to that track.

Public Constructors

```
StMcSvtHit(const StThreeVectorF& x,const StThreeVectorF& p,
      const float de, const float ds, const long key, const long id, StMc
```

Create an instance of *StMcHit* with position *x*, local momentum *p*, energy deposition *de*, path length *ds*, primary key *key*, volume id *id* and parent track *parent.*

Public Member Functions

`unsigned long  layer() const;`
Returns the number of the layer [1-6] in which a hit is found.

`unsigned long  ladder() const;`
Returns the number of the ladder [1-8] in which a hit is found.

`unsigned long  wafer() const;`
Returns the number of the wafer [1-7] in which a hit is found.

`unsigned long  barrel() const;`
Returns the number of the barrel [1-3] in which a hit is found.

`unsigned long  hybrid() const;`
Returns the number of the hybrid.

Examples

```
//
//  Function which returns the SVT hits
//  used by a track and prints them.
//
void getHits(StMcTrack *track)
{
    StPtrVecMcSvtHit& hits =
                  track->svtHits();
    StMcSvtHitIterator iter;
```

```
    StMcSvtHit* hit;
    for(iter = hits->begin();
        iter != hits->end(); iter++) {
        hit = *iter;
        cout << hit << endl;
    }

}
```

10.14 StMcSvtHitCollection

Summary

StMcSvtHitCollection is a container for the layers of the SVT. The actual hits are stored according to wafers, 3 levels down the hierarchy.

Synopsis

```
#include "StMcSvtHitCollection.hh"
class StMcSvtHitCollection;
```

Description

StMcSvtHitCollection is the first step down the hierarchy of hits for the SVT. It provides methods to return a particular layer, and to count the number of hits in all the SVT.

Persistence

None

Related Classes

StMcSvtHitCollection has a container of type `StMcSvtLayerHitCollection` of size 6, each element represents a layer. Look in *StMcSvtLayerHitCollection* 10.16, *StMcSvtLadderHitCollection* 10.15, and *StMcSvtWaferHitCollection* 10.17,

Public Constructors

`StMcSvtHitCollection();`
Create an instance of *StSvtHitCollection* and sets up the internal containers.

Public Member Functions

`bool addHit(StMcSvtHit*);`
Adds a hit to the collection (using *StMemoryPool* from the StarClassLibrary.

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole SVT.

`unsigned int  numberOfLayers() const;`
Returns the size of the Array of layers. Default is 6.

`StMcSvtLayerHitCollection*  layer(unsigned int);`
Returns a pointer to the layer specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples

```
//
// Look in StMcSvtWaferHitCollection for an example.
// These classes normally go hand in hand.
```

10.15 StMcSvtLadderHitCollection

Summary

StMcSvtLadderHitCollection is a container for the wafers of the SVT. The actual hits are stored according to wafers, the next level down the hierarchy.

Synopsis

```
#include "StMcSvtLadderHitCollection.hh"
class StMcSvtLadderHitCollection;
```

Description

StMcSvtLadderHitCollection is the third step down the hierarchy of hits for the SVT. It provides methods to return a particular wafer, and to count the number of hits in the ladder.

Persistence

None

Related Classes

StMcSvtLadderHitCollection has a container of type `StMcSvtWaferHitCollection` of size 7, each element represents a wafer. Look in *StMcSvtLayerHitCollection* 10.16, *StMcSvtHitCollection* 10.14, and *StMcSvtWaferHitCollection* 10.17,

Public Constructors

`StMcSvtLadderHitHitCollection();`
Create an instance of *StSvtLadderHitCollection* and sets up the internal containers.

Public Member Functions

`unsigned long numberOfHits() const;`
Counts the number of hits of this ladder.

`unsigned int  numberOfWafers() const;`
Returns the size of the Array of wafers. Default is 7.

`StMcSvtWaferHitCollection*  wafer(unsigned int);`
Returns a pointer to the wafer hit collection specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples

```
//
// Look in StMcSvtWaferHitCollection for an example.
// These classes normally go hand in hand.
```

10.16 StMcSvtLayerHitCollection

Summary *StMcSvtLayerHitCollection* is a container for the layers of the SVT. The actual hits are stored according to wafers, 2 levels down the hierarchy.

Synopsis
```
#include "StMcSvtLayerHitCollection.hh"
class StMcSvtLayerHitCollection;
```

Description *StMcSvtLayerHitCollection* is the second step down the hierarchy of hits for the SVT. It provides methods to return a particular ladder, and to count the number of hits in the layer.

Persistence None

Related Classes *StMcSvtLayerHitCollection* has a container of type `StMcSvtLadderHitCollection` of size 8, each element represents a ladder. Look in *StMcSvtHitCollection* 10.14, *StMcSvtLadderHitCollection* 10.15, and *StMcSvtWaferHitCollection* 10.17,

Public Constructors

`StMcSvtLayerHitHitCollection();`
Create an instance of *StSvtLayerHitCollection* and sets up the internal containers.

Public Member Functions

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole SVT.

`unsigned int  numberOfLadders() const;`
Returns the size of the Array of ladders. Default is 8.

`StMcSvtLadderHitCollection*  ladder(unsigned int);`
Returns a pointer to the ladder specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples
```
//
// Look in StMcSvtWaferHitCollection for an example.
// These classes normally go hand in hand.
```

10.17 StMcSvtWaferHitCollection

Summary *StMcSvtWaferHitCollection* is the class where the SVT hits are actually stored.

Synopsis
```
#include "StMcSvtWaferHitCollection.hh"
class StMcSvtWaferHitCollection;
```

Description *StMcSvtWaferHitCollection* holds the SVT hits.

Persistence None

Related Classes *StMcSvtWaferHitCollection* has a container of type `StSPtrVecMcSvtHit`, this means that this is the class that actually owns the hits. Look at the section on containers 6.4 for more information.

Public Constructors
`StMcSvtWaferHitCollection();`
Create an instance of *StMcSvtWaferHitCollection* and sets up the internal container.

Public Member Functions
`StSPtrVecMcSvtHit&  hits();`
Returns a reference to the container of the SVT hits.

Examples
```
//
// Print number of hits in SVT and plot 2D Histogram of x,y of
// Hits
//
StMcSvtHitCollection* svtHitColl = evt->svtHits();
cout << ''There are :'' << svtHitColl->numberOfHits()
     << ''hits in the SVT'' << endl;
TH2F* myHist = new TH2F("SVT","pos. of Hits",100, -10, 10, 100, -10, 10)
for (unsigned int ly=0; ly<svtHitColl->numberOfLayers(); ly++) {
  for (unsigned int ld=0; ld<svtHitColl->layer(ly)->numberOfLadders(); ld++) {
    for (unsigned int w=0;
         w<svtHitColl->layer(ly)->ladder(ld)->numberOfWafers();
         w++) {

      StSPtrVecMcSvtHit& hits =
              svtHitColl->layer(ly)->ladder(ld)->wafer(w)->hits();

      for (StMcSvtHitIterator i = hits.begin(); i != hits.end(); i++) {
           currentHit = (*i);
           myHist->Fill(currentHit->position().x(),
                        currentHit->position().y());
      }
    } // wafer  loop
  } // ladder loop
} // layer  loop
```

10.18 StMcTpcHit

Summary *StMcTpcHit* represents a monte carlo TPC hit.

Synopsis

```
#include "StMcTpcHit.hh"
class StMcTpcHit;
```

Description *StMcTpcHit* inherits most functionality from *StMcHit.* For a complete description of the inherited member functions see *StMcHit* (sec. 10.10).

In the StMcEvent data model each track keeps references to the associated hits and vice versa. All hits have a pointer to their parent track. This is especially important for the making of the associations in StAssociationMaker.

Persistence None

Related Classes *StMcTpcHit* is derived directly from *StMcHit.* The hits are kept in an instance of *StMcTpcHitCollection* where they are stored by pointer (see 6.4). Each instance of *StMcTrack* holds a list of TPC hits which belong to that track.

Public Constructors

```
StMcTpcHit(const StThreeVectorF& x,const StThreeVectorF& p,
      const float de, const float ds, const long key, const long id, StMc
```

Create an instance of *StMcHit* with position *x*, local momentum *p*, energy deposition *de*, path length *ds*, primary key *key*, volume id *id* and parent track *parent.*

Public Member Functions

```
unsigned long  sector() const;
```

Returns the number of the sector [1-24] in which a hit is found.

```
unsigned long  padrow() const;
```

Returns the number of the padrow [1-45] in which a hit is found.

Examples

```
//
//  Function which returns the TPC hit
//  that is closest to the
//  start vertex of the track.

StMcTpcHit* getFirstHit(StMcTrack *track)
{
    StThreeVector<float> vtxPosition = track->startVertex();
    StMcTpcHitCollection *hits =
                  track->tpcHits();
    StMcTpcHitIterator iter;
    StMcTpcHit* hit;
    float minDistance = 20*meter;
    for(iter = hits->begin();
        iter != hits->end(); iter++) {
        if (abs(*iter->position() - vtxPosition) < minDistance) {
           minDistance = abs(*iter->position() - vtxPosition);
           hit = *iter;
        }
    }
```

```
    return hit;
}
```

10.19 StMcTpcHitCollection

Summary *StMcTpcHitCollection* is a container for the sectors of the TPC. The actual hits are stored according to padrows, 2 levels down the hierarchy.

Synopsis
```
#include "StMcTpcHitCollection.hh"
class StMcTpcHitCollection;
```

Description *StMcTpcHitCollection* is the first step down the hierarchy of hits for the TPC. It provides methods to return a particular sector, and to count the number of hits in all the TPC.

Persistence None

Related Classes *StMcTpcHitCollection* has a container of type `StMcTpcSectorHitCollection` of size 24, each element represents a sector. Look in *StMcTpcSectorHitCollection* 10.21, and *StMcTpcPadrowHitCollection* 10.20,

Public Constructors

`StMcTpcHitCollection();`
Create an instance of *StTpcHitCollection* and sets up the internal containers.

Public Member Functions

`bool addHit(StMcTpcHit*);`
Adds a hit to the collection (using *StMemoryPool* from the StarClassLibrary.

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole TPC.

`unsigned int  numberOfSectors() const;`
Returns the size of the Array of sectors. Default is 24.

`StMcTpcSectorHitCollection*  sector(unsigned int);`
Returns a pointer to the sector specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples
```
//
// Look in StMcTpcPadrowHitCollection for an example.
// These classes normally go hand in hand.
```

10.20 StMcTpcPadrowHitCollection

Summary *StMcTpcPadrowHitCollection* is the class where the TPC hits are actually stored.

Synopsis
```
#include "StMcTpcPadrowHitCollection.hh"
class StMcTpcPadrowHitCollection;
```

Description *StMcTpcPadrowHitCollection* holds the TPC hits.

Persistence None

Related Classes *StMcTpcPadrowHitCollection* has a container of type `StSPtrVecMcTpcHit`, this means that this is the class that actually owns the hits. Look at the section on containers 6.4 for more information.

Public Constructors
`StMcTpcPadrowHitCollection();`
Create an instance of *StMcTpcPadrowHitCollection* and sets up the internal container.

Public Member Functions
`StSPtrVecMcTpcHit&  hits();`
Returns a reference to the container of the TPC hits.

Examples
```
//
// Print number of hits in TPC and plot 2D Histogram of x,y of
// Hits
//
StMcTpcHitCollection* tpcHitColl = evt->tpcHits();
cout << ''There are :'' << tpcHitColl->numberOfHits() << ''hits in the TPC'' << endl;
TH2F* myHist = new TH2F("TPC","x y pos. of Hits",100, -10, 10, 100, -10, 10)

for (unsigned int s=0; s<tpcHitColl->numberOfSectors(); s++) {

  for (unsigned int pr=0; pr<tpcHitColl->sector(s)->numberOfPadrows(); pr++) {

    StSPtrVecMcTpcHit& hits = tpcHitColl->sector(s)->padrow(pr)->hits();

    for (StMcTpcHitIterator i = hits.begin(); i != hits.end(); i++) {

      currentHit = (*i);
      myHist->Fill(currentHit->position().x(), currentHit->position().y());
    }
  } // padrow  loop
} // sector  loop
```

10.21 StMcTpcSectorHitCollection

Summary

StMcTpcSectorHitCollection is a container for the sectors of the TPC. The actual hits are stored according to padrows, the next level down the hierarchy.

Synopsis

```
#include "StMcTpcSectorHitCollection.hh"
class StMcTpcSectorHitCollection;
```

Description

StMcTpcSectorHitCollection is the second step down the hierarchy of hits for the TPC. It provides methods to return a particular padrow, and to count the number of hits in the sector.

Persistence

None

Related Classes

StMcTpcSectorHitCollection has a container of type `StMcTpcPadrowHitCollection` of size 45, each element represents a padrow. Look in *StMcTpcHitCollection* 10.19, and *StMcTpcPadrowHitCollection* 10.20,

Public Constructors

`StMcTpcSectorHitHitCollection();`
Create an instance of *StTpcSectorHitCollection* and sets up the internal containers.

Public Member Functions

`unsigned long numberOfHits() const;`
Counts the number of hits of the whole TPC.

`unsigned int  numberOfPadrows() const;`
Returns the size of the Array of padrows. Default is 45.

`StMcTpcPadrowHitCollection*  padrow(unsigned int);`
Returns a pointer to the padrow specified by the argument to the function. Recall that this is for indexing into an array, so the argument should go from 0 - (size()-1). Look in the numbering scheme conventions 6.1for more information.

Examples

```
//
// Look in StMcTpcPadrowHitCollection for an example.
// These classes normally go hand in hand.
```

10.22 StMcTrack

Summary *StMcTrack* describes a Monte Carlo track in the STAR detector.

Synopsis
```
#include "StMcTrack.hh"
class StMcTrack;
```

Description *StMcTrack* describes a monte carlo track in the STAR detector. *StMcTrack* provides information on the hits which belong to the track, its momentum and the vertices of the track. All tracks stored on the g2t_track table are collected in a *StSPtrVecMcTrack* which can be obtained through the *StMcEvent::tracks()* method. The tracks from the particle table are also stored in this container. For a track that appears in both tables, the information is merged so that only one entry per track appears in StMcEvent. In addition, the information about the particle that produced the track is accessible.

The information on the relation between hits and tracks in StMcEvent is kept both in *StMcTrack* and in *StMcHit.* Each track knows about the hits it is associated with, and vice versa. This was needed in order for the StAssociationMakerto begin its job by building hit associations and from those create the track associations.

Persistence None

Related Classes All monte carlo tracks are kept in an instance of *StSPtrVecMcTrack* where they are stored by pointer (see sec. 6.4).

Public Constructors

```
StMcTrack();
```
Constructs an instance of *StMcTrack* with all values initialized to 0 (zero). Not used. The tracks are constructed using the g2t_track table.

Public Member Functions

```
const StLorentzVectorF&      fourMomentum() const;
```
Returns the four-momentum of the track.

```
const StThreeVectorF&        momentum() const;
```
Returns the momentum of the track.

```
float                        pt() const;
```
Returns pt of the track.

```
float                        rapidity() const;
```
Returns rapidity of the track.

```
float                        pseudoRapidity() const;
```
Returns pseudoRapidity of the track.

```
StMcVertex* startVertex();
```
Returns a pointer to the start vertex of the track.

```
StMcVertex* stopVertex();
```
Returns a pointer to the stop vertex of the track.

`StMcTrack* parent();`
Returns a pointer to the parent of the track, if it exitsts. Otherwise, returns a null pointer.

`StPtrVecMcVertex& intermediateVertices();`
Returns a reference to the collection of intermediate vertices of the track. Note, that the collection can be empty.

`StPtrVecMcTpcHit& tpcHits();`
TPC hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcSvtHit& svtHits();`
SVT hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcFtpcHit& ftpcHits();`
FTPC hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcFtpcHit& richHits();`
RICH hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcRichHit& richHits();`
RICH hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcCalorimeterHit& bemcHits();`
BEMC hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcCalorimeterHit& bprsHits();`
Barrel pre-shower hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcCalorimeterHit& bsmdeHits();`
Barrel shower max-eta hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StPtrVecMcCalorimeterHit& bsmdpHits();`
Barrel shower max-phi hits belonging to the track. The returned container holds the hits by pointer. Note, that the collection can be empty. This will be the case if the option to load these hits is switched off.

`StParticleDefinition* particleDefinition();`
Returns a pointer to the only instance of the appropriate *StParticleDefinition* class, which holds information on the particle (name, charge, mass, spin, etc).

`int isShower();`
Returns 1 if the track is shower, 0 if not.

`long geantId();`
GEANT particle ID.

`long pdgId();`
PDG code particle ID (when the track comes from the particle table and has no GEANT ID). If the track comes from the g2t table, this will also be set but the numerical value will be equal to the geantId value. (That is what is in the g2t table...)

`long key();`
Primary key from g2t_track table.

`long eventGenLabel();`
Event Generator Label. This label should be different from zero if the track comes from the event generator. Useful also in determining the entry in the particle table (as index = label - 1).

`void setFourMomentum(const StLorentzVectorF&);`
Assigns the four-momentum of the track.

`void setStartVertex(StMcVertex*);`
Assigns the start vertex of the track.

`void setStopVertex(StMcVertex*);`
Assigns the stop vertex of the track.

`void setIntermediateVertices(StPtrVecMcVertex&);`
Assigns the intermediate vertex collection of the track.

`void setTpcHits(StPtrVecMcTpcHit&);`
Assigns the TPC hit collection of the track.

`void setSvtHits(StPtrVecMcSvtHit&);`
Assigns the SVT hit collection of the track.

`void setFtpcHits(StPtrVecMcFtpcHit&);`
Assigns the FTPC hit collection of the track.

`void setRichHits(StPtrVecMcRichHit&);`
Assigns the RICH hit collection of the track.

`void setBemcHits(StPtrVecMcCalorimeterHit&);`
Assigns the BEMC hit collection of the track.

`void setBprsHits(StPtrVecMcCalorimeterHit&);`
Assigns the BPRS (pre-shower) hit collection of the track.

`void setBsmdeHits(StPtrVecMcCalorimeterHit&);`
Assigns the BMDE (shower max-eta) hit collection of the track.

`void setBsmdpHits(StPtrVecMcCalorimeterHit&);`
Assigns the BMDP (shower max-phi) hit collection of the track.

`void setShower(char);`
Assigns the shower flag of the track.

`void setGeantId(long);`
Assigns the GEANT ID of the track.

`void setPdgId(long);`
Assigns the PDG ID of the track.

`void setKey(long);`
Assigns the primary key of the track.

`void setEventGenLabel(long);`
Assigns the event generator label of the track.

`void setParent(StMcTrack*);`
Assigns the parent track if it exists.

`void addTpcHit(StMcTpcHit*);`
Adds a TPC hit to the internal hit collection.

`void addFtpcHit(StMcFtpcHit*);`
Adds a FTPC hit to the internal hit collection.

`void addSvtHit(StMcSvtHit*);`
Adds a SVT hit to the internal hit collection.

`void addRichHit(StMcRichHit*);`
Adds a RICH hit to the internal hit collection.

`void addBemcHit(StMcCalorimeterHit*);`
Adds a BEMC hit to the internal hit collection.

`void addBprsHit(StMcCalorimeterHit*);`
Adds a BPRS hit to the internal hit collection.

`void addBsmdeHit(StMcCalorimeterHit*);`
Adds a BSMDE hit to the internal hit collection.

`void addBsmdpHit(StMcCalorimeterHit*);`
Adds a BSMDP hit to the internal hit collection.

`void removeTpcHit(StMcTpcHit*);`
Removes a TPC hit from the internal hit collection.

`void removeFtpcHit(StMcFtpcHit*);`
Removes a FTPC hit from the internal hit collection.

`void removeSvtHit(StMcSvtHit*);`
Removes a SVT hit from the internal hit collection.

`void removeRichHit(StMcRichHit*);`
Removes a RICH hit from the internal hit collection.

`void removeCalorimeterHit(StPtrVecMcCalorimeterHit&, StMcCalorimeterHit*)`
Removes a calorimeter hit from the specified vector. Used by the methods below.

`void removeBemcHit(StMcCalorimeterHit*);`
Removes a Calorimeter hit from the internal BEMC Emc hit collection. Calls removeCalorimeterHit.

`void removeBprsHit(StMcCalorimeterHit*);`
Removes a Calorimeter hit from the internal BPRS Emc hit collection. Calls removeCalorimeterHit.

`void removeBsmdeHit(StMcCalorimeterHit*);`
Removes a Calorimeter hit from the internal BSMDE Emc hit collection. Calls removeCalorimeterHit.

`void removeBsmdpHit(StMcCalorimeterHit*);`
Removes a Calorimeter hit from the internal BSMDP Emc hit collection. Calls removeCalorimeterHit.

Examples

Example 1:

```
//
//  Calculate the mean pt of all primary
//  tracks.

double meanPtOfTracks(StMcEvent *event)
{

    StPtrVecMcTrack& tracks = event->primaryVertex()->daughters();

    StMcTrackIterator iter;
    StMcTrack* theTrack;

    double sumPt = 0;
    int    n = 0;

    for (iter = tracks.begin(); iter != tracks.end(); iter++) {
        theTrack = *iter;       // careful here, pointer collection

        // add up pt
        sumPt += theTrack->pt();
        n++;
    }
    return n ? sumPt/static_cast<double>(n) : 0;
}
```

10.23 StMcVertex

Summary *StMcVertex* describes a vertex, i.e. a start or/and end point of one or several tracks

Synopsis
```
#include "StMcVertex.hh"
class StMcVertex;
```

Description *StMcVertex* describes a general vertex in the STAR detector. It uses the information stored in the *g2t_vertex.idl* table. *StMcVertex* has a position, a GEANT volume, a TOF and a GEANT process. Each vertex also has a parent track and 0...n daughter tracks. All are referenced by pointers. The container of all vertices in the event can be obtained through *StMcEvent::vertices()*.

Persistence None

Related Classes The StSPtrVecMcVertex container of *StMcEvent* owns the vertices. This container stores the vertices by pointer.

Public Constructors

`StMcVertex();`
Default constructor. Constructs an instance of *StMcVertex* with values initialized to 0. Not used. Construction is done using the g2t_vertex table.

`StMcVertex(float x, float y, float z);`
Constructs a vertex in position (x, y, z).

Public Member Functions

`const StThreeVectorF&  position() const;`
Vertex position in global coordinates.

`StPtrVecMcTrack& daughters();`
Collection of all daughter tracks. Note that the collection can be empty.

`unsigned int numberOfDaughters();`
Number of daughter tracks. Same as `daughters().size()`.

`StMcTrack* daughter(unsigned int i);`
Returns the `i`'th daughter track, or a NULL pointer if `i` is larger than or equal to the number of stored daughters. (Index runs from 0 to `numberOfDaughters()-1`).

`const StMcTrack* parent();`
Pointer to parent track.

`string geantVolume() const;`
Returns the GEANT volume read from g2t_vertex table.

`float tof();`
TOF read from g2t_vertex table.

`long geantProcess() const;`
GEANT process read from g2t_vertex table.

`long   key() const;`
Primary key, read from g2t_vertex table.

`long geantMedium() const;`
GEANT medium read from g2t_vertex table.

`long generatorProcess() const;`
Process from event generator, read from g2t_vertex table.

`void setPosition(const StThreeVectorF&);`
Sets vertex position.

`void setParent(StMcTrack*);`
Sets pointer to parent track.

`void addDaughter(StMcTrack*);`
Adds a daughter track to the daughter collection.

`void setGeantVolume(string);`
Sets the GEANT volume.

`void setTof(float);`
Sets the Time Of Flight.

`void setGeantProcess(int);`
Sets the GEANT process.

`void removeDaughter(StMcTrack*);`
Removes a track from the daughter container.

Public Member Operators

`int operator==(const StMcVertex&) const;`
Returns true (1) if two instances of *StMcVertex* are equal or false (0) if otherwise. The only data members used for the comparison are the position and the GEANT process.

`int operator!=(const StMcVertex&) const;`
Returns true (1) if two instances of *StMcVertex* are not equal or false (0) if otherwise. This operator is implemented by simply inverting *operator==*.

`int operator!=(const StMcVertex&) const;`
Returns true (1) if two instances of *StMcVertex* are not equal or false (0) if otherwise. This operator is implemented by simply inverting *operator==*.

`ostream& operator<<(ostream& os, const StMcVertex& v);`
Prints position, geant volume, time of flight and geant process to standard output.

Examples

Example 1:

```
//
//  Function to count # of tracks originating
//  from the primary vertex that have more than
//  10 hits in the TPC.
unsigned long countGoodPrimaryTracks(StMcEvent *event)
{
    unsigned long counter = 0;
    StPtrVecMcTrack& primaryTracks =
                event->primaryVertex()->daughters();
    StMcTrackIterator i;
    StMcTrack* track;
    for (i = primaryTracks.begin(); i != primaryTracks.end(); i++) {
```

```
        track = *i;
        if (track->tpcHits()->size() > 10)
            counter++;
    }
    return counter;
}
```

Index