Successful living with PhAT 5.0

Contents

1 Introduction

The main purpose of the PHOBOS analysis framework PhAT is to provide the collaboration with common interfaces for all aspects of data handling, from raw data collected in the DAQ computers to online monitoring and signal processing to reconstruction and physics analysis. PhAT is written in C++ and based (and entirely dependend) on the ROOT class libraries. ROOT provides tools for persistent object I/O, container classes, graphics, histogramming and fitting. Perhaps most importantly it includes a C++ script interpreter that allows interactive handling of PhAT objects, rapid prototyping and development of flexible and powerful analysis scripts. In this document we'll try to introduce the most important concepts of PhAT from data objects to super modules and show how they are supposed to interact with each other.

This manual assumes some familiarity with C++ and ROOT. There are several web pages providing introductions to ROOT for various levels of C++ expertise. See

- *http://www-pat.fnal.gov/root/ (Good for beginners)*
- *http://glast.phys.washington.edu/ haas/Root.html*
- *http://www.rhic.bnl.gov/ markp/root/index.html*
- *and of course http://root.cern.ch/*

2 Using Phat 5.0

In this section we are going to describe the basic class categories within PhAT for the storage of event and run related data, implementation of analysis algorithms and the interaction between data and algorithms. This is neither a step-by-step introduction nor a complete reference guide. Rather some of the basic design choices and coding conventions for PhAT 5.0 are described, trying to make sure that a common look-and-feel for the overall package is maintained. For details on the class usage see the html reference guide at
http://www.rhic.bnl.gov./ phobos

We make extensive use of documentation that is written by the author of the class in the source code. ROOT provides facilities to take thata documentation and turn it into HTML pages that are suitable for finding out how to use a given class.

2.1 Basic Data Classes

Classes like TPhHit, TPhTrack provide a representation of the objects that the Phobos physics analysis will finally be concerned with. Other classes like TPhRawHitFECs0 are used to store the digitized charge information from the detector elements in its various forms.
Common to all the data classes is that they typically contain no or very little intelligence - i.e. a track knows how to apply a straight line fit to the hits connected with it, but the track finding algorithm is provided by a separate class based on TPhModule (see below). This separation was one of the PhAT design choices, mainly motivated by the hope of having greater flexibility in design changes and an easier transition from traditional procedural software development to the brave new world of OO.

2.2 TPhTopology – an introduction

TPhTopology is a means to describe a detector configuration which allows a virtual partition of a flat data array into segments that correspond to various levels. The only constraint is that all of the nodes at the "bottom" of the tree are at the same depth relative to the root node.

The bottom nodes comprise one or more channels. The nodes above it comprise all of the channels of the nodes beneath it.

The topology does not itself contain any data. Instead, it serves as merely the directory which allows the user to get access to the data by reference to a subcomponent, or even just simple access to all of the channels.

The important objects here are:

- *TPhTopology The root topology is a topology. However, each node of the root is itself a topology, as are all of its nodes, etc.*
- *TPhTopoIter This is an iterator to a topology at a certain depth. If t is a TPhTopology, then t-¿Begin(2) gives a TPhTopoIter at depth 2 (2 levels down from root, or 1 in the above diagram). The iterator is able to skip over the structures defined above it. Thus, one can iterate over all of the "1"'s without knowing the structure of the "2"'s above them.*
- *TPhTopoPtr This is a RefPtr (smart pointer) to a topology. This is used to place the responsibility of whether a topology should be deleted with the topology itself. Just think of this as a TPhTopology* and everything should be fine!*

Finally, keep in mind that you will not use TPhTopology directly. Rather, the TPhTopoData objects will access its topology for you by forwarding your *requests to it. See the documentation in TPhTopoData.cxx for some more explanation of this.*

2.2.1 TPhTopoData – an introduction

TPhTopoData is the means by which one can store and access data according to a structure (e.g. FEC structure, FB structure, etc.) defined by a TPhTopology.

The constructor to a TPhTopoData requires the existence of a TPh-Topology (see documentation in TPhTopology.cxx). If it is a topology, then **`TPhTopoData(t, depth)`** *allocates an array corresponding to the depth. If depth = -1 then the array has one entry per data channel. Otherwise, the array gets one entry for each node. For instance, one might want an entry for each chip in an FECDetector for a CMN calculation.*

Useful methods:

- *Begin(i) gets an iterator to channel i.*
- *End() gets an iterator pointing to one past the last element*
- *Begin(TPhTopoIter), Begin(TPhTopoPtr) gets data corresponding to the iterator. lets one get the data for a chip, string, etc.*
- *Depth() counts how many depths lie beneath the root node.*
- *Length() counts how many data channels lie beneath the root node.*
- *At(int i) gets data at offset i*
- *Get(i) creates a new topodata representing the data for the i'th object.*
- *GetDataFor(TPhTopoIter) gets a new topodata for the object pointed to by the topooiter*

2.2.2 TPhHitArrays

The interface between the online/raw data part of PhAT and the offline/reconstruction code is provided by the TPhHitArray class. The TPhHitArray contains the calibrated charge deposition in individual sensitive Phobos detector elements. For the silicon sensors, one hit array exists per sensor, providing a two-dimensional representation of energy values in GeV deposited in the sensor pixels. This forms the basis for creating containers of individual TPhHit objects representing the coordinates of impact of tracks on the sensors.

2.3 Data Containers

Many of the basic data classes - again hits and tracks are good examples - are usually stored in so-called container classes, with TPhObjectContainer for unnamed objects and TPhHashList for named objects being the most common ones. The container classes provide common I/O for all objects stored in them and allow a large collection of objects to be passed around between different analysis modules, usually by storing and retrieving from either the TPhEvent object or the TPhObjectManager object.
The data containers are named objects, allowing a storage of an arbitrary number of containers in the event or object manager, provided they are identified by a unique name.

2.3.1 Data persistence

TPhDataObjects and TPhObjectContainers provide the IsPersistent() member function, telling the event or object manager whether the object should be streamed out to mass storage or not, with the default being not to write the object to disk.

2.3.2 Object ownership

The behaviour of the TPhObjectContainer destructor can be controlled with the SetIsOwner() member function. By default, the container 'owns' the objects that are stored inside it, calling their destructors when the container destructor is called. As the same object can in principle be stored in several containers, the SetIsOwner() member function can be used to declare only one container as owning the objects, making sure they are only deleted once (the consequences of deleting the same objects twice can actually be fairly desastrous if the space has been reused in the meantime).

2.4 TPhEvent and TPhObjectManager

The TPhEvent and TPhObjectManager objects provide temporary storage in memory and persistent storage in root files. For most applications only one of each will exist in memory at any given time. As the name suggests, TPhEvent deals with TPhDataObjects (e.g. a TPhVertex object) or TPhObjectContainers (e.g. a TPhContainer containing TPhHits) representing data for one event.
The TPhObjectManager on the other hand stores data pertaining to a collection of many events (e.g. a run or a collection of runs). Typical data stored in the TPhObjectManager object are calibration constants or transformation

tables, cumulative histograms or counters.
Both TPhEvent and TPhObjectManager store named objects, i.e. access to the data objects is provided by a `Get("NameOfObject")` *member function. Both classes provide persistent I/O, i.e. all objects in the event (unless the user chooses otherwise) can be written to mass storage by one call to the Write() member function and likewise, all the objects in the event can be streamed back into memory by a reading the event object from mass storage. There is effectively no limit to the number of data objects stored in the event or object manager, provided they are identified by unique names. An event can for instance contain several TPhVertex objects found by different vertexing algorithms.*

Deleting an event or the object manager will also delete all the objects currently owned by them. [1] *The default is that any object added to TPhEvent is owned by it.*

2.5 Modules, Supermodules and Context

As mentioned above, as a general rule, the storage and handling of data and the algorithms themselves are separated. One of the most important changes in PhAT V5.0 was to formalize the basic design of the algorithmic classes by introducing the TPhModule class. All analysis classes are supposed to inherit from TPhModule. The common interface of all modules are five member functions that are called upon initialization, start of a new run, for each event, at the end of a run and at the end of the analysis.

```
virtual EReturnCode StartAnalysis();
virtual EReturnCode StartRun();
virtual EReturnCode Process(TPhEvent*);
virtual EReturnCode EndRun();
virtual EReturnCode EndAnalysis();
```

In general, not all of these functions have to be implemented by all modules.

TPhModule provides a default constructor for compatibility with the ROOT system - this should not be used to create a module in user code. Instead, the standard way to instantiate a module is the following constructor:

```
TPhModule(const Text_t* name, const Text_t* title,
TPhModuleContext* context);
```

This provides a name and a title to the module, allowing them to be stored in a ROOT database or within the hash list of a supermodule (see below). As a convention the 'title' of the module is set automatically to the module

[1] We may make this scheme more clever by employing smart pointers in the future.

class name, with the 'name' characterizing the specific instance. In a user macro, the creation of a module typically looks like this:

```
TPhVertexFinderMod *VertexFinder = new
        TPhVertexFinderMod("MyVertexFinder",context);
```

Besides creating a vertex finding module named `MyVertexFinder` *this constructor also provides the module with a pointer to a TPhModuleContext object (see below).*

2.5.1 Module Parameter Setting

Datamembers of a module that are user-settable are called parameters. They can be identified by adding a '$' in the comment field immediately following the '//' of the datamember declaration:

```
Float_t fProbCut; //$ Probability cut
```

Parameters have to be of basic type (`Int_t`, `Short_t`, `Float_t`, `Long_t`, `UInt_t`, `Bool_t`, `Char_t*`*) or* `TString`*. Arrays and* `enum` *are currently not supported, but will likely be added in the future.*

Parameters are identified automatically by the system and can be used in a number of ways:

- *Parameters can be set and got very easily by using the* `Set` *and* `Get` *memberfunctions, so that you don't need to write explicit memberfunctions for them:*

  ```
  root [0] TPhModuleContext *c = new TPhModuleContext;
  root [1] TPhObjectManager *o = new TPhObjectManager;
  root [2] c->SetObjectManager(o)
  root [3] TPhFindStraightTracksMod a("FindStraight", c)
  root [4] Float_t residual = 0.4;
  root [5] a.Set("fMaxResidual3", residual)
  root [6] Float_t copy_residual = 0;
  root [7] a.Get("fMaxResidual3", copy_residual)
  root [8] copy_residual
  (Float_t)4.00000005960464478e-01
  ```

- *All parameters of a given module can be viewed and changed at the same time by a popup dialog box. The dialog box can be invoked by the module member* `SetParameters()`*.*

- *Finally, the intention is that parameters from an analysis can be stored in an ASCII file for later retrieval or stored in the database. There* *are two memberfunctions for this purpose:* `Export(ofstream &)` *and* `Import(ifstream &)` *(see above example):*

```
root [10] ofstream f("my_analysis.txt")
root [11] a.Export(f)
root [12] .!  cat my_analysis.txt
Class:  TPhFindStraightTracksMod
Version:  1
#ei Name of hit container (Input)
TString fHitContainerName ""
#eo Name of track container (Output)
TString fTrackContainerName ""
# Number of spectrometer layers for track finding
Int_t fMaxLayer 4
# Minimum number of hits on valid straight track
Int_t fMinHits 3
# Max hit residual in 3rd plane
Float_t fMaxResidual3 0.4
# Max hit residual in 4th plane
Float_t fMaxResidual4 0.25
```

However, this mechanism is not yet fully developed.

2.5.2 Event Input/Output and Object Manager Input/Output

Event input and output container names are special parameters of a module. They should be identified by using the 'ei' and 'eo' tokens in the comment string immediately after the '$':

```
TString fHitContainerName; //$ei Hit container (Input)
TString fTrackContainerName; //$eo Track container (Output)
```

PhAT will provide mechanisms to check for the validity of the names and existance of input containers in the event in the future. Object Manager Input/Output containers should be identified similarly by using the 'oi' and 'oo' tokens.

2.5.3 TPhModuleContext

Another important feature of the modules is the TPhModuleContext, providing the modules with access to common services for storage and retrieval of long-lived objects in the object manager, retrieval of calibration constants and tables from databases, access to geometry and electronics setup information. TPhModuleContext also provides an interface to a common error *logger for the modules. These services a provided via the following member functions of TPhModuleContext:*

```
TPhRun*            GetRun();
TPhDetector*       GetDetector();
TPhFECDetector*    GetFECDetector();
TPhErrorLog*       GetErrorLog();
TPhFBDetector*     GetFBDetector();
TPhObjectManager*  GetManager();
```

Typically all modules within a given calibration, reconstruction or analysis job share a common context, assuring the consistency of information used between them.

2.5.4 Inter-module communication

Communication between modules is done via the exchange of named objects with the event and object manager. Modules therefore do not depend on specific other modules being available in memory or having been run before, but only on whether the necessary data objects have been stored in the event or the object manager. This allows us to completely specify a module by defining which objects or containers it expects as input from the event and object manager and which objects are produced as output in event and object manager. Any two objects using and producing the same objects should be completely interchangeable in running macros. The names of input and output objects for each module should be set to a default value in the modules constructor. They can be overwritten at any time, allowing the same module to be run on the same event with different tuning parameters producing different output containers. Using special tags in the comment fields of the module data member declaration allows the programmer to identify the names of input and output containers. This will be used to automatically check the availability of the containers needed by the module, check whether a particular chain of modules is likely to be consistent and provide automated module documentation. As a rule of thumb, modules should exit gracefully and with a meaningful error message if one of necessary input objects is not found in the event or object manager.

2.5.5 Multiple Algorithms

In general, modules should implement a single algorithm, controlled by a set of tuning parameters that get written out by the module streamer. Different algorithms with idential input and output object types should be implemented in seperate module classes (however, leaving interface and tuning parameters as similar as possible).

2.5.6 Side-effects

Modules are supposed to have no side-effects beyond modifying the content of event and object manager. In particular, they are not supposed to perform I/O by themselves, leave files in the user directory, perform their own persistent object management or use means of module-to-module communication other then via named objects in event or object manager.

2.5.7 Message Logging

In general, once they get beyond a basic debugging stage, modules in their default instance should perform their job quietly, reporting only severe error conditions. The level of reporting should be controlled via the `SetVerbosity()` *member function of the module, with higher values corresponding to more detailed messages or less severe errors. Error logging is provided through the module context, providing a consistent message management for all modules sharing a context. Within the module, the error logging proceeds via a call to the ErrorLog() function, which returns on ostream to which messages are written:*

```
ErrLog() << "No TPhEventHeader!" << endl;
```

At this point this is just a front-end to which a more powerful logging engine will be added as soon as possible.

2.5.8 Data ownership

Modules in general to not retain ownership of the data objects they use or produce, other than for temporary objects solely used internally. Modules give up ownership (i.e. the responsibility to delete after end of use) of all data objects once they are added to the event or object manager and thus visible to the outside world.

2.5.9 TPhSupermodule

In many cases a fixed set of modules will be called in sequence to perform a given analysis task. Running common analysis tasks is made simple by the use of supermodules, which provide a way of running a pre-defined collection of modules in a fixed order and with a fixed set of tuning or steering parameters. TPhSuperModule inherits from TPhModule and therefore also provides the StartAnalysis(),StartRun(),Process(), EndRun() and EndAnalysis() member functions. The supermodule member functions call the respective module member functions, in the order specified when setting up the supermodule.
Supermodules provide persistent I/O for the modules stored inside them,

allowing complete analysis packages to be set up by tuning the individual modules, then adding them to a supermodule and writing the supermodule to mass storage.
When in memory, individual modules within the supermodules can for testing purposes be replaced by functionally equivalent modules to allow testing/tuning of various analysis steps. Also, the order in which the modules are run by the supermodule can be changed for testing purposes.
As supermodules inherit from modules, they can in turn be combined into higher level super modules.
For testing purposes the modules within a supermodule can be individually enabled or disabled, using the SetEnabled() member function. By default, modules are enabled when adding them to the supermodule.

2.5.10 Module database

We will establish a database of collaboration standard module instances, distributed as a ROOT file stored in CVS. Standard macros will obtain the instances of macros from this file, making sure that these macros are reproducible and accountable. Standard modules are identified by their classname (`ClassName()`*) and name (as in* `GetName()`*) data members.*

2.6 TPhDetector

The TPhDetector object provides an interface to the information on sensor setup and electronics mapping of the experiment. The information is contained in two objects, TPhGeometry and TPhFECDetector.

2.6.1 TPhGeometry

Geometry objects *The parameters of geometrical properties of individual sensors are stored in Phat as objects of classes derived from the TPhSensor class. Different types of sensors are represented by objects of different class: TPhSpectSensor for sensors of spectrometer, TPhTOFSensor for sensor of TOF detector and so on. Phat defines also sets of groups of sensors by an object of class TPhSensorSet. A number of predefined TPhSensorSet objects is created when geometry is initialized. Sensors are grouped in subdetector, layer, plane and module object types, defined in header files of the corresponding classes (TPhSubDetector, TPhSensorLayer ...). A user can create any number of additional sets of sensors, that his analysis requires. Both sensors and sensor sets are geometrical objects. Their classes derive from TPhSensorNode class which in turn derive from TNode class that is used to specify position of the sensor in space and provides facilities to build a physical tree of unique connections between sensors and sensor sets. Within this tree, a connection means that the position of a given node (sensor) is* *given with respect to the parent node, so the position of the sensor in the main reference frame is defined by a full path of connections from a given sensor to a volume in a master reference frame. In Phat this volume is called MOTHER. A parallel tree of connections between sensors and sensor sets is built using objects of TPhSensorSet class. This class in addition to being able to participate in the tree of nodes, has also additional lists of sensors and sensor sets. Using them one can define a logical tree of sensor sets. The tree of sensor sets is built during initialization of the geometry. The pointer to the tree is stored in the object of TPhGeometry class. In addition to this link, the TPhGeometry object has two separate lists of all sensors and sensor sets for fast access to these objects by their name.*

The tree of geometry objects can be either Print()'ed or viewed using TBrowser object. After reading in the geometry from the ROOT file, a geometry can be found in the Geometries subdirectory of the TBrowser canvas.

Geometry methods *Methods of the geometry classes build the geometry objects, read and change their properties and modify the predefined geometry tree of objects.*

The top class that provides access to geometry is TPhGeometry. Using it one can obtain quickly a pointer to a sensor or a sensor set object. This is also the place where top functions referring to global properties of the geometry can be found. TPhGeometry derives from the ROOT TGeometry class so the mechanism of geometry visualization in ROOT canvas is available and the TPhGeometry is integrated with the ROOT environment.

The TPhSensorSet class defines a group of sensors. Different sensor set types are recognized by sensor set name, class name and by user type. The class provides iterators over its sensors and subsets of sensors and also allows to find them by name. The classes derived from TPhSensorSet, specialized for a specific detector, like TPhOctagonDet may provide additional methods to retrieve sets of sensors belonging to certain parts of the detector (GetLadder, GetPhiStrip). The order of objects in the lists can be changed by sorting them according to several predefined criteria. The class overrides methods of geometrical transformations (Shift, Rotate) in order to make sure that all sensors it contains are actually connected by their nodes to the sensor set object and are thus affected by the change.

The TPhSensor class refers to a single sensor entity. Most of the geometrical properties of the sensor are reflected in the part of the class that is derived from TPhSensorNode. Additional methods providing current spatial properties of the sensor are provided by specialized classes.

Both TPhSensorSet and TPhSensor derive from TPhSensorNode class. This is an extension of the ROOT TNode class. This class provides methods for geometrical calculations not present in the TNode class or specific to Phobos requirements.

Geometry properties *Summarizing, sensors are stored in objects of classes that derive from a single, common base class — TPhSensorSet. This makes it easy to write processing functions that take a TPhSensorSet object as parameter to define a geometric scope of processing operations. Sensors can be stored in many different sensor sets which makes possible to define multiple, user defined logical sets of sensors that are most convenient for the user from the point of view of particular application. A tree of predefined logical structures consisting of subdetectors, layers, planes and modules is created and stored together with sensor properties in the geometry file.*

Both sensors and sensor sets are TNode type classes that derive from a common TPhSensorNode class. The geometrical parameters of all sensors are stored in the appropriate objects of classes that derive from a common class TShape. TPhSensorNode class provides basic geometrical and sorting operations . The geometrical transformations can be applied not only on a sensor by sensor level, but moving sensor sets, just as it happens in reality. The changes in a position of a sensor set are automatically propagated to all sensors (and their subsets) in the current sensor set.

More detailed description of classes and objects that the geometry in Phat is composed of can be found at
`http://phobos-srv.chm.bnl.gov/~olszewski/Active\_geometry.html`

2.6.2 TPhFECDetector

2.7 Database access

Within PhAT, database functionality is provided by ROOT and Oracle. A common interface to storage and retrieval of arbitrary PhAT objects using simple keys is provided by the TPhDBKeyManager class.

2.7.1 Keys, KeyPtr's and the Key Manager

This style of database access is intended to have been created by experts for use by various users, and the interface therefore has little or no direct reference to the actual database. All reference to objects residing in the database is by means of a key, an object deriving from TPhDbKey, which encapsulates the SQL concept of a primary key to a table – it is an object which refers to exactly one 'object' somehow stored in Oracle.[2]

2.7.2 Creating TPhDbKeys

There are two ways to obtain a key. One is directly, through a class's constructor. See for example the constructor of TPhCalKey or TPhRunKey.

[2]Though that object need not be actually stored in one uniquely referenced row of a single table.

This requires you to know, exactly, all the information that the Oracle tables are keyed on.

The second way is to use a key class's Select function. This function is designed to return "the best" key[3] satisfying some specification. See for example TPhCalKey::Select. This would be used in the following way:

```
TPhCalKey::Select("SIPED",2121,"PHAT5");
```

where this will return you the "best" (most recent) calibration key which is a silicon pedestal set, for method PHAT5 (the currently used one) and also valid for run 2121.

2.7.3 TPhKeyPtr's

The TPhKeyPtr is a template class. Objects of the class function for all intents and purposes (to the user) just like a pointer. One specifies what the TPhKeyPtr is a pointer to by the template parameter, thus:

```
TPhKeyPtr<TPhRun> run;
```

declares 'run' as a key pointer to a TPhRun object. If one has a pointer to a valid object (which the above declaration does not do, more below), it could refer to the actual run object thus:

```
run->GetDate();
```

2.7.4 Reading TPhKeyPtr's

The first way of making a TPhKeyPtr is to read in an object which already exists in the database. To do this, you just ask for that object by its key:

```
TPhRunKey runkey(1543);
TPhKeyPtr<TPhRun> run(runkey);
```

would make run a pointer to a TPhRun object for run 1543. One can condense this syntax and use temporaries:

```
TPhKeyPtr<TPhRun> run(TPhRunKey(1543));
TPhKeyPtr<TPhPedestalSet>
  peds(TPhCalKey::Select("SIPED",2121,"PHAT5");
```

where the first statement does the same as in the previous example, and the second statement creates a pointer to a pedestal object referred to by the key

[3] It actually returns a lazy-evaluation object called a TPhDbKeySelector, but usually you do not need to worry about this distinction.

discussed in the example in the previous section – the "best" set for run 2121.

There are two advantages to using KeyPtr's in this way. First, when you stream an object out to a Root file, any KeyPtr's in the object will write only the key – not the whole object pointed to. When the object is read back in, the key manager will consult the database and recover the object by its stored key, and reattach the pointer. Second, the key manager performs caching of objects, so that objects will generally not be read twice from the database.

2.7.5 Writing TPhKeyPtr's

The second way of making a TPhKeyPtr is to write out an object which is not yet in the database to the database, defining its key in the process. The way this is expresses is by making a pairing of a Key and its object – which does an implicit write to the database.[4] *For example:*

```
TPhPedestalSet *mypeds;
// I've just calculated mypeds for run 2119 somewhere else.
TPhKeyPtr<TPhPedestalSet> peds(TPhCalKey("SIPED",2119,"PHAT5"),mypeds);
```

This is very similar to the previous, except for the explicit specifying of an actual key, not asking for the "best" key – note that the previous Select function could well have returned this pedestal set, if it had been in the database and no other sets were in for runs 2120 or 2121 – and the explicit specifying of the object pointed to, in this case mypeds. It is the fact that these are explicitly specified that makes the system write the new association into the database.

2.7.6 Caveats to using KeyPtr's

Once you create a KeyPtr referring to an object, the TPhKeyManager thinks it 'owns' the object, and might delete it out from under you (if you've discarded your KeyPtr), or might give someone else a pointer to that object without telling you. Here are some rules to follow when working with KeyPtrs:

- *Don't make KeyPtr's to objects which are not now or soon to be stored in the database.*
  The current implementation forbids this, but that may change in the future.
- *Don't change an object pointed to by a KeyPtr unless you know what you must.*

[4] This may be made more explicit in the future.

The current implementation also forbids this, but this may change since there are too many ROOT functions which are not const-correct. If it is changed, changing the object a KeyPtr points would cause others hard-to-find bugs.

- *Don't take pointers to the object pointed to by the keyptr without keeping the KeyPtr itself around. Remember that the KeyPtr is destroyed when you leave scopes!*
  If you do this, it is possible that the Key Manager will think you're done with the object and delete it. Or it might not, if someone else has a KeyPtr to it somewhere. This is another source of hard-to-find bugs.

2.7.7 Using SQL directly

One can use SQL directly, if one understands the structure of the database itself (discussion of which is currently beyond the scope of this document). One uses the interface supplied by three abstract base classes to SQL: TPhSQLDatabase, TPhSQLConnection, and TPhSQLStatement.

In order to execute a statement, one must follow several steps. First, one must initialize the database (once per Phat session). The normal way of doing this is:

```
gPhDb = new TPhDatabase();
```

TPhDatabase uses TPhSQLDatabase, and is aware that the (current) Phobos database uses Oracle.

Second, one must authenticate yourself to this database. You must do this whenever you want to change the user you're logging in to Oracle as. To do this:

```
gPhDb->Authenticate("userid","password");
```

Third, one gets a connection. You should only keep this connection as long as you need to, since otherwise multiple calls to the database can interfere with each other. The usual way of doing this is:

```
TPhDatabase::TConn activate(gPhDb);
TPhSQLConnection *conn = gPhDb.Connection();
```

The first statement creates a local object that ensures that the connection is released when you leave the scope of the function you are in. It only needs to be done once in a function, even if you refer to the connection more than once in a function. If for some reason you do not want the connection to be released upon leaving the function, you must explicitly use the functions

TPhDatabase::Activate and ::Release – and be careful about early returns from your function!

Fourth, one creates a statement. This is obtained from the connection:

```
TPhSQLStatement *s =
  conn->NewStatement("SELECT X FROM TABLE");
```

Note that the statement must be deleted by the user.

2.7.8 Binding and Defining

See the header files.

2.8 CINT macros

The main purpose of CINT within PhAT is to provide the 'glue' for assembling compiled classes into a running analysis macro. After initial prototyping, a typical analysis macro should contain at most a couple of hundred lines of code. All algorithms should be contained in compiled classes, all data stored in instances of compiled classes. Typically the macro should just instantiate a set of modules or supermodules, open a connection to a data stream and implement a run/event loop, providing a straightforward sequential flow of data.
Nested macros (macros calling other macros) should be avoided.

2.9 Job Manager

With a simple analysis macro, you want to read in a file(s), do some processing, and then write out a file(s) (such a grouping of files, is refered to as a file set). To read in and write out the files you have to tell the macro which ones you want (i.e you pass the macro the directory and run number of an input file and similarily for the output file). This works well for one user, using their own files on their own machine, running over just on set of inputs and outputs. But when you want to run on other machines (like the farms on RCF) or over multiple sets of input and output files, then you would have to change the macro to suit the environment you are running in + do other work to get it to run. Then you have many slightly different versions of essentially the same macro =¿problems occur when tring to compare results etc.

The solution to this problem is the job manager object(TPhJobManager). It makes the macro an invarient, and puts all the changes into the jobmanager which is used as the input to an analysis macro. The job manager knows how to convert a machine independent reference name, that is used in the macro, to the specfic file name required on the place you are running. (e.g. "RawData" -¿means "/mnt/data/4/PhoRaw002306s000.dat" on one *machine but " phobsink/engrun/data/PhoRaw002306s000.dat" on another etc..). This way you do not have to change your analysis macro in any way when you change where you run (but you do change the job manager!). The other feature of the job manager is that it can contain multiple file sets that you can run over within your single analysis macro, ensuring that the macro itself does not change if you run over one file set or multiple file sets.*

The details of how to use the job manager can be found at http://phobos-srv.chm.bnl.gov/Detectors/Computing/Documentation/jobmanager_doc.html/ , or goto Computing-¿Basic Docs-¿Job Manager Documentation (under Miscellaneous heading)