The MINOS Off-line Software User's Manual
Release R1.11

October 18, 2004

Contents

Chapter 1

Job Control

Last significant change: 2002/06/21

The job control package provides the interface to the data for reconstruction and analysis modules. The package defines the basic unit for reconstruction and analysis, the "**JobCModule**" and provides several standard modules which allow users to organize the various modules into reconstruction/analysis jobs.

JobCModules allow physicist programmers to plug their reconstruction and analysis code into the framework. A **JobCModule** consists of a header (`.h`) and an associated (`.cxx`) file. A module calls algorithms which interface with the data to perform reconstruction. Several **JobCModules** are provided by the framework which perform basic reconstruction and analysis of MINOS data. These can be organized into reconstruction and analysis jobs using `ROOT` macros. Instructions for configuring the **loon** executable using `ROOT` macros are given in Section 1.1.

In addition to the modules provided by the framework, users can provide their own **JobCModules** and plug them into **loon**. Instructions for how to do this are provided in Section 1.2

1.1 Running and Configuring `loon`

The **loon** executable is configured via a `ROOT C++` macro. The basic command is:

> *loon* `-[options]` *macro.C -o out.root in1.root in2.root in3.root ...*

where "macro.C" is the ROOT macro which configures the job, and "out.root" is the default name to use for data output via the output module. The list of input data files are listed as "in1-in3.root". The possible command line options are:

-b : *Run in batch mode without graphics.*
-n : *Do not execute ROOT logon and logoff macros.*
-q : *Quit the job after completing the commands in the macro.*
-h : *Print usage help summary.*
-H[module] : *Print help for a specified module.*
-d[url]: *Specify database URL.*
-u[user]: *Specify database user.*
-p[passwd]: *Specify database password.*
-t[time]: *Set the time limit for the job (eg. -t'10 minutes').*
-r[n]: *Specify the maximum number of records to evaluate before quitting.*
-o[out]: *Set the default name to use for output file.*

The most common options are "-bq" which specify batch running of **loon**. **loon** can also be run interactively (just leave off the "-bq" option). In this mode the user will be given essentially a `ROOT` prompt and can execute C++ statements, `ROOT` macros and commands.

1.1.1 Writing a JobControl Macro

Job control macros provide several functions. The macros are used to 1) organize the analysis and reconstruction modules into "paths" which define the sequence the modules are applied to the input data, 2) alter the configuration of jobs modules and framework services, and 3) trigger module commands. Several example job control macros are provided in the **Demo** package which is a good source to see working examples of macros and **JobCModules**.

Creating job paths

The basic function of the job control macro is to define the list of reconstruction and analysis steps to perform on the data. The following example macro defines a very simple job path:

```
{
  JobC j;
  j.Path.Create("Demo",
// if MC        "RerootToRawDataModule::Get "
                "DigitListModule::Get "
                "DigitListModule::Reco "
                "UserAnalysis::Ana "
                "CheezyDisplayModule::Ana "
                "EventDump::Ana");
  j.Path("Demo").Run();
}
```

The first step in the macro, JobC j, creates a new job. A new reconstruction/analysis path, named "Demo" is created within this job using the `C++` command `j.Path.Create`. The first argument to this command in the name of the path ("Demo" in this case) which is used to reference this path later in the macro. The second argument is a list of modules (`EventDump` eg.) and methods (`Ana`, eg.) to add to the path. The example, shows the typical "event preparation" steps (`RerootToRawDataModule::Get` does the data format conversion for Monte Carlo events, `DigitListModule` builds the candidate digit lists which hold the hit strip data), as well as some modules provided by the framework to print and display information about events. Finally, this macro runs all the data from the input files through the "Demo" path via the command, `j.Path("Demo").Run()` .

While the "Create" command is the most common, the "Path" module provides several other commands to configure job paths. The complete list can be viewed in source file `JobCPathModule.h` which defines the interface to the "Path" module; several commands are listed here:

`j.Path("Demo")` *// return the path named "Demo"*
`j.Path.Delete("Demo");` *// delete the job path named "Demo"*
`j.Path.Add("Demo","Module::Method");` *// Append to a job path.*
`j.Path.Attach("A","B");` *Link the end of path A to the beginning of path B*
`j.Path.Report();` *// Print a summary of all active job paths.*
`j.Path.Reset();` *// Delete all active paths*

Running job paths

Job paths can be run in several ways:

`j.Path("Demo").Run();` *// Run until all input data has been processed*
`j.Path("Demo").Run(10);` *// Run until 10 records have been read in*
`j.Path("Demo").RunNin(10);` *// Same as Run(10)*
`j.Path("Demo").RunNpass(10);` *// Run until 10 records pass all filters set in the path. Filters are discussed in Section 1.1.1.*
`j.Path("Demo").RunNfail(10);` *// Run until 10 records have been failed by the path.*

To get a summary of the run, use the `Report()` command:

`j.Path("Demo").Report();`
This produces output like this:

```
Demo(10 in 1 out 9 filt.)
  1) +DigitListModule::Get      n=10    (     10/     0) t=(  179.53/     7.13)
  2) +DigitListModule::Reco     n=10    (     10/     0) t=(   56.71/     1.46)
  3) +UserAnalysis::Ana         n=10    (      1/     9) t=(    0.01/     0.00)
  4) +EventDump::Ana            n=1     (      1/     0) t=(    0.01/     0.00)
```

This report indicates that the path "Demo" has processed 10 records, 1 of which passed the analysis, 9 of which were filtered. The path is made of 4 nodes (a `Get`, a `Reco`, and two `Ana`'s). The number of event processed by each node is listed. In this example, the node `UserAnalysis::Ana` saw 10 records, passed 1 of those and filtered 9. The last two columns list the total cpu and system time (in seconds) spent inside the node.

Filters

Each time a module performs some reconstruction or analysis on a data record it has the option to pass or fail the record depending on some criteria. For example, a muon track fitter, "FitTrack::Reco" might fail events if the goodness of fit is below some criteria. In normal running, data records process down the job paths until the record is failed, at which point the processing stops and a new record in read in.

The handling of filter settings can be configured in several ways. First, the user can opt to turn off all the filters in a path off. In this case processing will continue even for events which fail. All the filters in a path can be switched on and off using a command like:

```
j.Path("Demo").SetAllFilters(false);  // Turn all filters off
j.Path("Demo").SetAllFilters(true);  // Turn all filters on
```

Further, filtering at specific points in the analysis chain can be configured. For example,

```
j.Path("Demo").Node("MyModule::Ana").FilterOn();
j.Path("Demo").Node("MyModule::Ana").FilterOff();
```

Users can also reverse the meaning of the pass/fail decisions made by elements of the analysis chain. For example, if one wanted to output (ie. "pass") all events which failed the reconstruction path "Demo":

```
j.Path("Demo").ReverseAllFilters();
```

Alternately, the filters can be switched one-by-one, eg.:

```
j.Path("Demo").Node("MyModule::Ana").ReverseFilter();
```

Users can view the status of filters set in a given path with the "Path.Report()" command:

```
  1) +DigitListModule::Get      n=10    (    10/     0) t=(  179.53/    7.13)
  2)  DigitListModule::Reco     n=10    (    10/     0) t=(   56.71/    1.46)
  3) !UserAnalysis::Ana         n=10    (     1/     9) t=(    0.01/    0.00)
  4)  EventDump::Ana            n=1     (     9/     0) t=(    0.01/    0.00)
```

The markers to the left of each node label that the filter is on ("+"), on and reversed ("!"), or off (blank). In this example, the filter for the `UserAnalysis::Ana` node has been reversed. As you can see from the report, 9 events were failed by this node, and 9 events were processed by the next node in the path. In the case that a node makes no pass/fail decision all records are passed regardless of the on/off/reverse setting.

1.1.2 Job Configuration

There are several things that can be configured from a job control macro. These include the modules which perform the reconstruction as well as various services provided by the **loon** program.

Module configuration

Each job module carries with it a set of configuration parameters. To see the set of configuration parameters using the "Report" command:

```
loon [0] gSystem->Load("libDemo.so");
loon [1] JobC j;
loon [2] j.Path.Create("A","EventDump::Ana");
loon [3] j.Path("A").Mod("EventDump").Report();
EventDump configured with: Registry: 'EventDump.config.default', 13 entries.
     [U|L] DumpAll = 0
     [U|L] DumpCandDigits = 0
     [U|L] DumpFragments = 1
     [U|L] DumpObjectTable = 0
     [U|L] DumpRawDaqSnarlHeader = 1
     [U|L] DumpRawData = 0
     [U|L] DumpRawHeader = 1
     [U|L] First = 0
     [U|L] Freq = 1
     [U|L] Last = 999999999
     [U|L] PrintSeparator = 1
     [U|L] RawBlockList =
     [U|L] Wait = 0
```

Here the "EventDump" module has been placed into the path "A". The "EventDump" module is currently configured using a configuration names "EventDump.config.default", which has 13 entries with the values shown. Any of these parameters can be changed using a "Set" command:

```
loon [4] j.Path("A").Mod("EventDump").Set("Freq=10");
loon [5] j.Path("A").Mod("EventDump").Report();
EventDump configured with: Registry: 'EventDump.config.default', 13 entries.
     [U|L] DumpAll = 0
     [U|L] DumpCandDigits = 0
     [U|L] DumpFragments = 1
     [U|L] DumpObjectTable = 0
     [U|L] DumpRawDaqSnarlHeader = 1
     [U|L] DumpRawData = 0
     [U|L] DumpRawHeader = 1
     [U|L] First = 0
     [U|L] Freq = 10
     [U|L] Last = 999999999
     [U|L] PrintSeparator = 1
     [U|L] RawBlockList =
     [U|L] Wait = 0
```

From the report you should be able to see that the frequency parameter "Freq" has been changed from 1 to 10.

Modules support a second interface ("Cmd" instead of "Set"). These commands vary from module to module. They should be documented in the help for the module. For example, `j.Path("").Mod("Module").Help();`.

Framework services

Several of the framework services can be configured. For example, message print thresholds can be altered using the "Msg" module. Here are some examples:

`j.Msg.SetLevel("Plex","Error");` *// Raise the threshold from emitting messages from the Plex package so that only error messages are printed.*
`j.Msg.SetLevel("MyPackage","Verbose");` *// Lower the threshold for "MyPackage" so that all debugging messages are printed.*
`j.Msg.Attach("MyPkg::Error","mypackage.err");` *// Attach the file "mypackage.err" to the "MyPkg::Error" stream.*
`j.Msg.Format("MyPkg::Debug","File,Line",off);` *// Remove the file name and line number tag from messages printed to "MyPkg::Debug" stream.*

`j.Msg.Stats();` *// Print statistics about message stream use. Useful to see if any warnings or errors were produced during your run.*

1.1.3 Job modules provided by the framework

This section reviews some basic modules provided by the framework and how to use them.

Input module

The most important module provided by the framework is the input module. One input module is created when you create a job:

```
loon [0] JobC j;
loon [1] j.Input.Report();
INPUT configured with: Registry: 'INPUT.config', 9 entries.
```

```
[U|L] DDSDataSource = Daq
[U|L] DDSKeepUpMode = FileKeepUp
[U|L] DDSMaxSyncDelay = 15
[U|L] DDSOffLine = 0
[U|L] DDSPort = 9090
[U|L] DDSServer = daqdds.minos-soudan.org
[U|L] DDSTimeOut = 120
[U|L] Format = input
[U|L] Streams = DaqMonitor,DaqSnarl,LightInjection
```

As you can see from the report the input module has several configuration parameters. Most of these are for controlling the interface to the DataDispatcher (DDS), which is not the most common use of the input module. Generally, the input module talks to files on a local disk. The two parameters you need to set in that case are "Format" which is in general either "input" for detector data, "reroot" for Monte Carlo. If you want to take data from the DataDispatcher you would `j.Input.Set("Format=dds")`. The second important parameter is the setting of which data streams you would like to subscribe to. For example, if you only wanted to see "DaqSnarl" records you would `j.Input.Set("Streams=DaqSnarl");`

The input module has the following commands:

`j.Input.Next(n)` *// Advance n records in the stream (default is 1).*
`j.Input.Prev(n)` *// Go back n records in the stream (default is 1).*
`j.Input.GoTo(run,event)` *// Go to specified run event number.*
`j.Input.List()` *// List all files atteched to the input module*
`j.Input.AddFile("filename")` *// Add a file to the list*
`j.Input.RemoveFile("filename")` *// Remove a file from the list*
`j.Input.NextFile(n)` *// Advance n files in the list (default is 1)*
`j.Input.PrevFile(n)` *// Go back n files in list (default is 1)*
`j.Input.GoToFile(n)` *// Go to the ith file in the list*
`j.Input.GoToFile("file")` *// Go to the file "file" in the list*
`j.Input.Select("stream","cut")` *// Set a selection cut on a stream*

When a `JobC` object is created the `Input` module automatically loads files from the **loon** command line.

Other modules

Here I just list some of the other modules that exist in the framework:

- `CheezyDisplayModule` — Draw an event using the CheezyDisplay
- `DigitListModule` — Builds Raw and MC Truth CandDigitLists from RawRerootRecord
- `EventDump` — A module to print text info. about a data record.
- `Output` — Write data to output streams
- `RerootToRawDataModule` — Builds RawRecord with RawDigitDataBlock
- `RerootToTruthModule` — Builds RawRecord with RawDigitDataBlock
- `Root` — A module to handle ROOT commands
- `RootCommand` — A module to handle ROOT commands

1.2 How to Make a Job Module

This section will outline how to make a job module and link it into **loon**.

Automatic generation of source code

To plug into the framework, users must write a new class which will contain the analysis and reconstruction code. The `JobControl` package provides a simple program which will prompt the user with questions about the module they intend to write, and then automatically generate template code for the ḣ and `.cxx` which define the user's `JobCModule`.

To use the generation program, type:
> *gen_module*
.

The program will prompt the user for several fields:

- **Class name** What is the `C++` name of the class you are writting. Some examples are "DemoEventDump", and, "IoInputModule".
- **Package name** What package are you working in? Examples, are "Demo", and "JobControl".

- **Human readable name** What name should people use to access this module from their jobs. For example, the class "DemoEventDump" is known just as "EventDump" (dropping the `C++`-y package prefix).
- **Short description** What does your module do? Keep it short and simple.
- **Your e-mail address** You're the author, take credit for your work!
- **What methods will you implement?**
  - **Ana** - Is this an analysis module?
  - **Reco** - Does the module do event reconstruction?
  - **BeginJob** - Is there any start-of-job setup that needs to be done?
  - **EndJob** - Is there any end-of-job clean up to do?
  - **BeginFile** - Do you need to know when a new file has been opened?
  - **EndFile** - Do you need to know when a file has been closed?
  - **BeginRun** - Do you need to know when a new has started?
  - **EndRun** - Do you need to know when a run has ended?
  - **Get** - Do you need to do event data input? (Usually just I/O modules)
  - **Put** - Do you need to do event data output? (Usually just I/O modules)
  - **Help** - Do you want to provide some help about how to use your module?
  - **Report** - Do you want to override the default Report command?
  - **Reset** - Do you want to provide a way to wipe your module's memory clean?
  - **HandleCommand** - Are then any commands you expect to support besides those that just change the module's configuration parameters?
  - **Configuration parameters** - You can setup the module's configuration by filling these forms. This will setup the module's default configuration and generate some template code to adjust the modules's configuration using the `Set("")` method.

Once complete, `gen_module` will write the `.cxx` and `.h` files for the class. These classes will require some tweaking before they compile. For example, the `Config()` method will need to be hooked up before it will work. Also, comments will need to be added (search for the tag "`FILL_IN`". Oh, and you'll have to fill in your analysis and reconstruction code...

Once the module compiles, you'll need a `GNUmakefile` if you already have one. For most packages this is simple:

```
#
# GNUmakefile for simple package
#
ROOTCINT := YES
LIB := lib$(PACKAGE)
LIBCXXFILES := $(wildcard *.cxx)
#########################################################################
include SoftRelTools/standard.mk
include SoftRelTools/arch_spec_root.mk
#
```

A "gmake" should get your module compiled into the package library and the library installed in your test release.

The next step is to attach your module to the **loon** program. This step is done by updating a file which lives in the SRT test and/or base release which stores the map from job module names to the libraries that provide them. To update this file, use this command:
> *makemodulemap*
That should be all there is to it. When you ask to add a module to a job path, **loon** will consult this file (`$SRT_PRIVATE_CONTEXT/tmp/jobmodules.txt`), load the required library, and build your module.

Job modules: A closer look

This section takes a closer look at what goes on inside `JobCModule`s using the `JobControl/JobCDemoModuleA` as an example. The header file for `JobCDemoModuleA` is shown in Figure 1.1. A sketch of the implementation for this module is shown in Figure **??**.

User modules must inherit from the base class `JobCModule` as shown in line 019 in the figure. The user module can optionally provide implementations for one or more of the following methods:

```
////////////////////////////////////////////////////////////////////////
// $Id: job_control.tex,v 1.9 2004/06/14 19:38:45 west Exp $
//
// A dummy module used for testing
//
// messier@huhepl.harvard.edu
////////////////////////////////////////////////////////////////////////
#ifndef JOBCDEMOMODULEA_H
#define JOBCDEMOMODULEA_H

#ifndef JOBCMODULE_H
#include "JobCModule.h"
#endif

class MomNavigator;
class JobCommand;

//......................................................................

class JobCDemoModuleA : public JobCModule
{
public:
  JobCDemoModuleA();
  ~JobCDemoModuleA();

  void BeginJob();
  void EndJob();
  void BeginFile();
  void EndFile();
  void BeginRun();
  void EndRun();

  void Get(MomNavigator *mom);

  JobCResult Ana(const MomNavigator *mom);
  JobCResult Reco(MomNavigator *mom);
  JobCResult AnaNonPhysics(const MomNavigator *mom);
  JobCResult RecoNonPhysics(MomNavigator *mom);

  void HandleCommand(JobCommand *command);
  void Help();
  void Report();
  void Reset();
private:
};

#endif // JOBCDEMOMODULEA_H
////////////////////////////////////////////////////////////////////////
```

Figure 1.1: Header file for a dummy job module.

- `BeginJob()` - called at the start of a job
- `EndJob()` - called at the end of a job
- `BeginFile()` - called when a new file begins
- `EndFile()` - called when the current file ends
- `BeginRun()` - called when the data run number changes
- `EndRun()` - called when the data run ends.
- `Get()` - called to add data to an event from an I/O stream
- `Put()` - called to write the data to an I/O stream
- `Ana()` - called to perform read-only analysis of data
- `Reco()` - called to perform reconstruction of data
- `AnaNonPhysics()` - called for non-physics data records
- `RecoNonPhysics()` - called for non-physics data records
- `HandleCommand()` - provides a user interface to the module
- `Help()` - prints a help message
- `Report()` - prints a report of the module status
- `Reset()` - set the state of the module back to its "begin run" status.

JobCResult

The various methods that look at data records are allowed to make pass/fail decisions on the event by returning values of type `JobCResult` which can hold several pieces of information about the result of your reconstruction or analysis. The possible results fall into a few categories:

- Selection criteria Modules have the option of filtering events according to selection criteria. For example, if an event satisfies some cuts you might want to "pass" the event, or if some reconstruction code fails to reconstruct an event you might want to "fail" it. Please note, that if "failed" is a possibility for your method, "passed" should be a possibility also so that records either pass or fail. These results are set via the `JobCResult::SetFailed()` and `JobCResult::SetPassed()` methods. Alternatively, one can return `JobCResult::kPassed` or `JobCResult::kFailed` and use the implicit conversion of this enumeration to a `JobCResult`. One final note, a result cannot hold both "passed" and "failed", so, for example, the `SetFailed()` method, clears results of the `SetPassed()` method and vice-versa.

- Error conditions If a module encounters some error which prevents it from doing its work, or detects conditions that cast doubt on whether the event processing is proceeding correctly, it can return an error result. These conditions are set via, `JobCResult::SetWarning()`, `JobCResult::SetError()`, and, `JobCResult::SetFatal()`. Error conditions will stop the event processing; fatal conditions will cause the program to abort.

- Status changes These are to be used by I/O modules and shouldn't concern the "physicist programmer". These results can flag the end of the input stream, the start of a new file, and other conditions encountered when reading from the input data stream.

The following are a few simple examples for using `JobCResult`:

```
JobCResult Module1::Ana(const MomNavigator* mom) {
// Simplest case - no errors, no pass/fail decisions
  return JobCResult::kAOK;
//
// This would also work:
//
// JobCResult r;
// return r;
}

//.........................................................................

JobCResult Module2::Ana(const MomNavigator* mom) {
//
// Another simple case: A single pass/fail decision
//
```

```
  if (this->PassesCuts(mom)) return JobCResult::kPassed;
  // else
  return JobCResult::kFailed;
}

//..........................................................................

JobCResult Module3::Ana(const MomNavigator* mom) {
//
// A more complicated case: Apply selection criteria and flag error
// conditions
//
  JobCResult r; // The result of this method's analysis of mom

  SomeDataObject* d = this->GetSomeDataObject(mom);

  // Do some error checking on the object d (for example)
  if (d->GetSomeNumber() > this->GetValidRange()) {
    r.SetError();  // Flag this condition as an error
    r.SetFailed(); // Fails selection criteria
  }

  // Apply selection criteria
  if (d->PassesCuts()) {
    r.SetPassed();
  }
  else {
    r.SetFailed();
  }

  // Return final result, could contain both ``failed'' and ``error''
  return r;
}
```

HandleCommand

In general, when users talk to modules, they are just chaning the module's configuration. If you have implemented the `Config()` method, then you are all set; the framework will trigger the `Config()` method when the `Set()` command is issued from a job control macro or from the **loon** prompt and the module will reconfigure itself based on the new set of parameters.

However, in some cases, you may need more specialized than the `Set()` command. In this case, you will want to talk to the module using the `Cmd()` command. This interface allows users to issue a large variety of commands to the module. For example,

```
loon [0] JobC j;
loon [1] j.Path.Create("A","AModule::Ana");
loon [2] j.Path("A").Mod("AModule").Cmd("Draw timeHisto chargeHisto");
```

This command might cause histogram windows to pop-up containing time and charge information.

If you find that you need to be able to talk to your module with more flexibility than is provided via `Set()` you will need to implement the `HandleCommand` method. This method receives a parsed version of the string command typed by the user as the argument to the `Cmd()` method. This parsed version is called a `JobCommand`.

`JobCommands` are of the form `/c1/c2/c3/... o1 o2 o3 ...` where `c1, c2, c3, ...` specify the command path (through menus and submenus) and `o1, o2, o3...` are the options associated with the command. Module can navigate the command path and options using the methods `JobCommand::PopCmd()`, `JobCommand::PopOpt()`, `JobCommand::PopIntOpt()`, `JobCommand::PopFloatOpt()`. For example,

```
UserModule::HandleCommand(JobCommand *cmd)
{
  std::string c1 = cmd->PopCmd();
  if (c1 == "Draw") {
    std::string opt = cmd->PopOpt();
    if      (opt == "Top")  this->DrawTopView();
    else if (opt == "Side") this->DrawSideView();
    else { ... print warning... }
  }
  if (c1 == "Dump") {
    std::string c2 = c2 = cmd->PopCmd();
    if (c2 == "Summary") {
      std::string opt = cmd->PopOpt();
```

```
        if (opt == "Muon") {...}
        else if (opt == "Electron") {...code...}
      }
      else if (c2 == "Stats") {
        ...code...
      }
    }
  }
```

handles the commands:

```
j.Path("A").Mod("UserModule").Cmd("Draw Top);
j.Path("A").Mod("UserModule").Cmd("Draw Side");
j.Path("A").Mod("UserModule").Cmd("Dump/Summary Muon");
j.Path("A").Mod("UserModule").Cmd("Dump/Summary Electron");
j.Path("A").Mod("UserModule").Cmd("Dump/Stats");
```

///

Chapter 2

Message Service

Last significant change: 2002/07/17

2.1 Introduction

The message service serves two main functions. The first is to provide a convenient way to output formatted numerical data as the C++ "<<" operator is lacking in this regard. The second is to provide a utility to filter and direct text output such as debugging, warning, and error messages. The MINOS C++ coding conventions state that users are to use the message service rather than `cout` and `cerr` directly so that these messages are processed in a uniform way.

2.2 Formatting Numerical Data

The message service provides a simple class to allow users to specify the width and precision of numerical output in much the same way as provided by the C routine `printf`. For example:

```
#include "MessageService/MsgService.h"
#include "MessageService/MsgFormat.h"

int main(void)
{
  int    i = 12345;
```

```
  double x = 123.456;
  MsgFormat ifmt("%9d");
  MsgFormat ffmt("%9.3f");

  // Don't worry about the MSG thing - its covered in the
  // next section. Just think of it as a tricky "cout"
  MSG("XXX",Msg::kInfo) << "[" ifmt(i) << "]" << endl;
  MSG("XXX",Msg::kInfo) << "[" ffmt(x) << "]" << endl;
}
```

will print the number 12345 using a width of 9 characters and 123.456 with a width of 9 characters with a precision of 3 places beyond the decimal point. Output looks like this:

```
[____12345]
[__123.456]
```

where the underscores mark blank spaces. The formating options available are summarized in table 2.1.

-	Left justify in field
+	Always show sign
O	Pad left with zeros
o	Octal format
x,X	Hexadecimal format
f	Fixed format
e,E	Scientific format
g,G	General format

Table 2.1: Format flags for use with class `MsgFormat`.

2.3 Text Messages

2.3.1 Basics

The MessageService also allows users to send text messages to output streams. Each message is associated with a given package (specified by the prefix chosen for the package) and a priority level. The simplest use looks like this:

```
#include "MessageService/MsgService.h"
.
.
.
// The CVSID line is required so that the message service can tag
// error messages with the version number attached to this file
CVSID("$Id: message_service.tex,v 1.4 2004/06/14 19:38:45 west Exp $");
.
.
.
{
  MSG("Pkg", Msg::kVerbose) << "This is a VERBOSE message\n";
  MSG("Pkg", Msg::kDebug)   << "This is a DEBUG message\n";
  MSG("Pkg", Msg::kInfo)    << "This is an INFO message\n";
  MSG("Pkg", Msg::kWarning) << "This is a WARNING message\n";
  MSG("Pkg", Msg::kError)   << "This is an ERROR message\n";
  MSG("Pkg", Msg::kFatal)   << "This is a FATAL message\n";

  MsgStream& m = MSGSTREAM("Pkg", Msg::kInfo);
  m << "This construct bypasses test to see if stream is active\n";
}
```

which sends text messages from package `Pkg` with various priorities ranging from lowest to highest. The threshold for printing can be set package by package. For example, if you wanted to see all the `Debug` messages but not all the `Verbose` messages from the package `Pkg` you would set the threshold for the package `Pkg` to `Debug`. The default threshold is `Info`. Eventually the `JobControl` package will provide a simple interface for doing this without recompiling, but for now it must be done manually with a line of code like:

```
MsgService::Instance()->GetStream("Pkg")->SetLogLevel(Msg::kDebug);
```

Note that messages that are below print threshold are not formatted. This means that unprinted messages cause only a negligible performance hit. On my machine (Pentium 350 MHz) the performance hit from 100000 unprinted messages was a total 0.15 seconds.

	Priority	Prefix	Date	File	CVS Version	Line #
Verbose	x	x				
Debug	x	x				
Info						
Warning	x	x				
Error	x	x	x	x	x	x
Fatal	x	x	x	x	x	x

Table 2.2: Default information attached to each message.

2.3.2 Advanced Features

The following outlines some of the more advanced features of the `MessageService`. Note that many of the code fragments shown will eventually be replaced by commands available from the `JobControl` package.

- **Writing message service streamers for your classes** The message service can write output for any class that has an operator `ostream& operator<<` defined. So, to write a class "C" out:

  ```
  #include <iostream>
  class C
  {
    friend class ostream& operator<<(ostream& os, const C& c) {
      return os << c.fData1 << "," << c.fData2;
    }
  };
  ```

  Any class that can be writen via "`cout << c`" to also use "`MSG(...) << c;`" without any modification.

- **Sending Messages to Multiple Output Streams** The message service allows messages to be send to multiple output streams. For example, if you wanted messages from package `Pkg` to go to both `cout` and a file `PkgInfo.txt` you would use:

  ```
  MsgService::Instance()->
    GetStream("Pkg")->AttachOStream(Msg::kInfo,"PkgInfo.txt");
  ```

- **Merging of Two Output Streams** The output of two streams can be merged like this:

```
MsgService::Instance()->
  GetStream("PkgA")->AttachOStream(Msg::kInfo,"merge.txt");
MsgService::Instance()->
  GetStream("PkgB")->AttachOStream(Msg::kInfo,"merge.txt");
```

  In this case `Info` messages from both `PkgA` and `PkgB` will be merged in the file `merge.txt`.

- **Concatenation of Two Output Streams** The output of two streams can be concatenated automatically like this:

```
MsgService::Instance()->
  AddCatStream("cat.txt","PkgA",Msg::kInfo);
MsgService::Instance()->
  AddCatStream("cat.txt","PkgB",Msg::kInfo);
```

  In this case `Info` messages sent from package `PkgA` and `PkgB` will be concatenated together in the file `cat.txt`.

- **Changing Format** By default, the message service will attach additional information to each message printed. The defaults are listed in table 2.2. To modify the defaults follow these examples:

```
MsgService::Instance->
  GetStream("PkgA")->AddFormat(Msg::kInfo,Msg::kName);
MsgService::Instance->
  GetStream("PkgA")->RemoveFormat(Msg::kInfo,Msg::kName);
MsgService::Instance->
  GetStream("PkgA")->SetFormat(Msg::kInfo,Msg::kName+Msg::kFile);
```

- **Statistics** To view the number of messages printed by package and file use:

```
MsgService::Instance()->PrintStatistics();
```

Chapter 3

Modules, Algorithms and Candidates

Last significant change: Printing candidates: 2004/04/26 Remainder: 2001/02/20

3.1 Introduction

This chapter will tell you how event reconstruction works in our framework and how you write reconstruction code.

At its heart of the reconstruction is simple: input data-like objects are processed to create new output data-like objects which can subsequently be used as input for further processing. In terms of this framework, the processing is performed by Algorithm objects and the outputs are Candidate objects. The framework structures a job into a set of modules each of which is responsible for passing input to a Algorithm and returning the results to the framework. The framework's primary responsibilities are to:-

- Allow the job user to determine what modules get used and in what sequence.
- Allow the job user to configure the Algorithms that get used. In general Algorithms will have sets of constants that control their operation. Defaults set are drawn from the database but the framework allows the job user to fine tune them.

- Prevent memory leaks by maintaining control over all the major objects in the reconstruction process.
- Record a full processing history so that subsequently it would be possible to determine how each Candidate was built and each Algorithm configured.

Your primary responsibility, as an Algorithm developer, is simply to write the single member function that performs reconstruction. Typically individual member functions are procedural so, although set in an OO framework, the task is not so very different from the equivalent job in a Fortran framework.

In this chapter we will focus on reconstruction. However the machinery applies equally well to those phases of analysis that involve the creation of further objects. The remainder of this chapter is divided into the following sections:-

Concepts The concepts that underpin Modules, Handles, CandContexts, MOM, Algorithms and Candidates.

Naming Convention The naming convention used for Modules, Handles, Algorithms and Candidates.

A Simple Example: DigitListModule A walk through a simple reconstruction process.

Working with Modules How to design a Module.

Working with Candidates How to design, create and use Candidates.

Working with Algorithms How to design, create and use Algorithms.

3.2 Concepts

There is a UML diagram in Cand section of the The MINOS Class User Guide tha draws together most of the classes introduced in this section.

3.2.1 Module

A Module is the basic building block of a job. Eventually there will be modules for I/O, calibration, reconstruction, analysis etc. As part of job configuration, the job user tells the framework what Module objects it must create and use.

All modules inherit from JobCModule which is essentially an abstract interface, leaving all the real work to its subclasses. Modules are informed of state changes such as job begin or file end. They have member functions that are called to set them to work. For example, reconstruction modules are called via their Reco method to perform reconstruction on the current event.

3.2.2 Handle

A handle gives you controlled access to an object. If you are familiar with Fortran memory managers such as BOS or ZEBRA, then a handle is a bit like a link. Just as with links you can have multiple handles to the same object. In our framework we use handles for Algorithm and Candidate objects. The primary reasons why we use them are:-

Avoiding Memory Leaks If you directly create any object then it is your responsibility to destroy it when no longer needed, or find another object that is prepared to take on that responsibility! That's why local (stack-based) objects, i.e. objects declared locally within a function are so good; as soon as you leave the function they get destroyed. However this option won't work for event reconstruction! It is not possible to determine, at creation time, how long an individual reconstruction object will live; for later reconstruction may need them even if the creator no longer does. The solution is to use what are called smart pointers. The idea is that any number of smart pointers can point to an object but they must be the only means of access. Once all smart pointers to an object are deleted the object itself can safely be deleted, as it has become inaccessible. What prevents smart pointers from being a source of memory leaks themselves is that they are lightweight and ideally suited to be local objects. However they can also be passed back out of function by value so that its rather like a baton in a relay race; so long as it is passed between functions the object remains alive, but drop the last baton and the object dies.

Some of our handles are smart pointers and from the explanation given above, it follows that they should normally be local objects.

Object Protection Reconstruction objects are initially modifiable but subsequently they become frozen as subsequent reconstruction becomes dependent on them. In our framework the way we prevent objects from being modified is as follows:-

1. First we move all its get and set methods into a companion handle which is made a friend. Now, even if you have an object, you cannot change it, you must have its handle to do that.
2. The handle can determine if a modifying access is allowed and if not either forbid it or create a clone copy of the object which the user can safely modify.

3.2.3 CandContext

A CandContext object is a lightweight object, typically created on the stack that identifies which part of the current event is being used in the role of input. CandContexts are created by Modules and passed to Algorithms so that they know what to operate on.

3.2.4 MOM

MOM is the Minos Object Mapper (O.K. so the name is a bit contrived). Its function is to provide access to all the major components, or fragments, of an event. All I/O works at the fragment level so MOM is really an IO stream manager. Currently MOM is still in prototype design and actually owns all its fragments. A single object reconstruction consists of the following stages:-

1. Ask MOM for an input fragment.
2. Create a CandContext that points to this fragment.
3. Create a new reconstruction object specifying this CandContext and an Algorithm.
4. Hand the resulting object back to MOM.

3.2.5 Algorithm

All Algorithm objects inherit from AlgBase. The role of an Algorithm is to initialise or modify a Candidate i.e. a reconstruction object. Your job is to create a subclass and write the overloaded member function RunAlg which performs this operation.

Conceptually Algorithms are stateless objects although in reality what this means is that during its lifetime changes in its state do not influence the way its RunAlg works. In fact Algorithms have several types of state:-

- Configuration or extrinsic state. This is the set of all constants that control the way it operates. It is called extrinsic because it is not held in the Algorithm object at all but in a companion AlgConfig object. In this way the same Algorithm can be made to operate in multiple ways by associating it with a different AlgConfigs. Currently AlgConfig is just a dummy placeholder but eventually AlgConfig objects will get default state from the database with additional configuration from the job user.

- Implementation or intrinsic state. For example a neural net Algorithm may hold a object structure produced as a result of training. This state does determine how it operates but doesn't change with time.

- Performance state. For example you might want to keep statistics on how well the Algorithm is performing. Strictly speaking this is a second form of intrinsic state but it does not influence operation in any way. Currently the framework has no support for this type of state.

Algorithms are accessed via AlgHandles. This is both to avoid a memory leaks (see section 3.2.2 on handles) and to allow an Algorithm and an AlgConfig to be packaged up together into a single operational unit. The user asks AlgFactory for a specific Algorithm and is handed back an AlgHandle.

3.2.6 Candidate

All reconstruction objects are Candidates and inherit from CandBase.

The name is meant to convey the concept that such objects could in principle be tentative and that alternatives are possible. For example there may be several ways to assign hits to a Track and a Shower object if they overlap. The Candidate package has support machinery to allow multiple reconstructions from the same data, using different configured Algorithms and includes the hooks for equality and overlap testing. Of course this facility will not be used at every stage of reconstruction, but stands ready for use where appropriate. Other built-in support includes:-

- Recording of reconstruction history. Currently dummy.
- Memory leak and object state protection by the use of CandHandles. See the above discussion 3.2.2 of how Candidate objects are protected by devolving their get and set methods to their companion CandHandles.
- I/O. Currently just that provided by ROOT. The ability to distribute Candidate objects over multiple TTrees has yet to be addressed.

All Candidates have a list of daughter Candidates, although strictly speaking the list is of CandHandles to the daughters. The term "daughter" is somewhat misleading as it implies dependence. Sometimes this implication is correct, for example a CandDigitList *owns* a list of CandDigits; if the CandDigitList is destroyed, so are its CandDigits. However later we will want objects, say CandPhysicsEvent that are built from pre-existing CandTrack and CandShower objects but will not own them. Its is better to think of Candidates as being of two types:-

Simple The daughter list is empty. The Candidate's state is to be found in its other data members.

Compound The daughter list is not empty. The Candidate's state is to be found only in its daughters. Of course the daughter list might be empty on occasions, but that is still the only state.

Candidates are created by an associated factory. The user passes in an AlgHandle and a CandContext and receives a CandHandle.

3.3 Naming Convention

In order to create an Xxx object the following classes have to be developed.

CandXxx The reconstruction object.

CandXxxHandle Its handle.

AlgXxx The Algorithm that creates it.

AlgXxxConfig The configuration for AlgXxx.

XxxModule The Module that gets called to create and which puts AlgXxx to work to create the Xxx object.

Often, Xxx is a actually a list of other Candidates, for example CandDigitList is simple a list of CandDigits. In such cases i.e where:-

```
Xxx = YyyList
```

there are a further set of Yyy classes except for YyyModule; the framework only knows about Yyy or lists of Yyy, not both.

3.4 A Simple Example: DigitListModule

3.4.1 Introduction

We shall see all of the above concepts brought into play by studying a simple reconstruction process: the creation of a list of CandDigits from the corresponding RawDigits.

3.4.2 DigitListModule code

At the framework level, what is being created is a CandDigitList and so, by the naming convention above, the Module that is called is DigitListModule.

Examining the source for this we see that most member functions are dummy. for example:-

```
 void DigitListModule::BeginJob()
{
  MSG("JobC", Msg::kInfo) << "DigitListModule::BeginJobn";
}
```

and

```
JobC::Filter_t DigitListModule::AnaNonPhysics(const MomNavigator *mom)
{
  MSG("JobC", Msg::kInfo) << "DigitListModule::AnaNonPhysicsn";
  return JobC::kNoDecision;
}
```

all the real work is done in:-

```
JobC::Filter_t DigitListModule::Reco(MomNavigator *mom);
```

It is passed mom and has to return a JobC::Filter_t. Eventually this return will be used as a way of filtering further processing. For now it is dummy. Stripping out all the comments and printout, the code is remarkably short:-

```
   RawRecord *rr = (RawRecord *) mom->GetFragment("RawRecord");
   AlgHandle adlh = AlgFactory::GetAlgDigitList();
   CandContext cx(this, mom);
   cx.SetDataIn(rr);
   CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
   CandDigitListHandle *onheap = new CandDigitListHandle(cdlh);
   mom->AdoptFragment(onheap);
   return JobC::kNoDecision;
```

We will now pick this apart, statement by statement.

```
   RawRecord *rr = (RawRecord *) mom->GetFragment("RawRecord");
```

First mom is asked to find the fragment called "RawRecord". It returns a TObject pointer so has to be cast up to a RawRecord pointer.

```
   AlgHandle adlh = AlgFactory::GetAlgDigitList();
```

Next a stack based AlgHandle is filled by asking the AlgFactory for a handle to a AlgDigitList. Note the very typical passing by value when using handles, they are so lightweight that is easier and safer not to use pointers.

Although you don't really need to know, there is nothing very mysterious going on here, if you look in AlgFactory. you will see that GetAlgDigitList is a static method (i.e. is called without an object):-

```
  AlgHandle AlgFactory::GetAlgDigitList()
{
   if (!fsAlgInstanceDigitList)
     fsAlgInstanceDigitList = new AlgDigitList();
   if (!fsAlgConfigDigitList)
     fsAlgConfigDigitList = new AlgConfigDigitList();
   return AlgHandle(*fsAlgInstanceDigitList, *fsAlgConfigDigitList);
}
```

which is simply creating static an AlgDigitList and an AlgConfigDigitList if needed and then creating an AlgHandle from them.

Returning to DigitListModule::Reco,

```
   CandContext cx(this, mom);
   cx.SetDataIn(rr);
```

The first statements creates a stack based CandContext which is passed this, a pointer to the DigitListModule, so that the reconstruction history can record who the creating module and the mom were. The second statement programs the CandContext to point to the RawRecord.

```
   CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
```

This is where the real action takes place! MakeCandidate is a static method of CandDigitList and it creates a CandDigitList object using the Algorithm handle and the CandContext. It returns a CandDigitListHandle by value. During execution of this statement the AlgDigitList is passed the newly created CandDigitList and the CandContext and uses the input data it has selected to fill out the CandDigitList.

```
   CandDigitListHandle *onheap = new CandDigitListHandle(cdlh);
```

This bit of fancy footwork is needed because we are about to give the new CandDigitList to mom who adopts objects by address. We cannot pass the address of cdlh, as this is on the stack and will shortly disappear. Instead we make a copy of the stack CandDigitListHandle on the heap so that it can be passed to mom.

```
   mom->AdoptFragment(onheap);
```

Now the CandDigitListHandle and associated CandDigitList are passed to mom. The word Adopt is important, its part of our Coding Convention (see NAM-7), and means that mom now owns the CandDigitListHandle and has to worry about its ultimate destruction.

```
  return JobC::kNoDecision;
```

Finally the function signals back to the caller. Currently this is dummy hence the kNoDecision value.

Its worth taking one last look at the code. Although it indirectly createda heap based CandDigitList and possibly a AlgDigitList, the code itself only created stack objects for the most part. The one exception was that last CandDigitList which it made mom's problem!

3.4.3 CandDigitList code

Now its time to take a closer look at the all important MakeCandidate:-

```
  CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
```

Looking at CandDigitList member function MakeCandidate:-

```
 CandDigitListHandle CandDigitList::MakeCandidate
                             ( AlgHandle &ah,
                               CandContext &cx)
{
  CandDigitListHandle cdlh;
  new CandDigitList(ah, cdlh, cx);
  return cdlh;
}
```

It creates a stack based CandDigitListHandle and then creates a heap based CandDigitList which is passed the AlgHandle, the local CandDigitListHandle and the CandContext. Passing the CandDigitListHandle to the CandDigitList is the way it gets connected. Finally the CandDigitListHandle gets returned by value.

Now we have reached the constructor for CandDigitList which contains:-

```
 CandDigitList::CandDigitList(AlgHandle &ah, CandHandle &ch,
                                                     CandContext &cx) :
  CandBase(ah)
{
```

```
 // Run Algorithm to construct Candidate
   {                                         // Start of scope.
     CandDigitListHandle cdlh(this);         // cdlh will go out of scope
     ch = cdlh;                              // after setting ch.
   }                                         // End of scope.
   ah.RunAlg(ch, cx);
}
```

The first few statements ensure that the CandBase class is initialised with the AlgHandle and that ch, the CandHandle that has been passed in by reference, gets connected to the new CandDigitList. The crucial line is:-

```
   ah.RunAlg(ch, cx);
```

for this is where the AlgDigitList::RunAlg gets called.

3.4.4 AlgDigitList code

For that we need to look at AlgDigitList. and in particular:-

```
 void AlgDigitList::RunAlg(AlgConfig &ac, CandHandle &ch,CandContext &cx)
{

// Iterate over DigitDataBlock or CandDigitList daughters.
   if (cx.GetDataIn()->IsA()->InheritsFrom("RawRecord")) {

// Iterate over input RawDigit's to fill input CandDigitList.

     AlgHandle ah = AlgFactory::GetAlgDigit();
     CandContext cxx(this, cx.GetMom());

// Iterate over RawRecord's RawDataBlockList to find DigitDataBlock.
     RawRecord *rr = (RawRecord *) (cx.GetDataIn());
     TIter rdbit = rr->RawDataBlockIter();
     TObject *tob;
     while ((tob = rdbit())) {
       if (tob->IsA()->InheritsFrom("DigitDataBlock")) {
         DigitDataBlock *rdb = (DigitDataBlock *) tob;
         TIter rdit = rdb->GetDatumIter();
         RawDigit *rd;
         while ((rd = (RawDigit *) rdit())) {
           cxx.SetDataIn(rd);
           CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
           ch.AddDaughterLink(cdh);
         }
       }
```

```
    }
  }
  else if (cx.GetDataIn()->IsA()->InheritsFrom("TCollection")) {
    TCollection *cdl = (TCollection *) cx.GetDataIn();
    TIter cdit(cdl);
    CandDigitHandle *cdh;
    while ((cdh = (CandDigitHandle *) cdit())) {
      ch.AddDaughterLink(*cdh);
    }
  }
}
```

The outer if statement:-

```
  if (cx.GetDataIn()->IsA()->InheritsFrom("RawRecord")) {

  ...


  else if (cx.GetDataIn()->IsA()->InheritsFrom("TCollection")) {

  ...

  }
```

demonstrates how a single Algorithm can be made to work on different sorts of input data. The CandContext is sent its GetDataIn message and responds by returning a TObject pointer to it. Then using standard ROOT services the TObject is asked for its TClass which in turn is asked if it InheritsFrom from "RawRecord" or "TCollection" Asking if it InheritsFrom allows the code to work even if the data are subclasses of the required classes. Now, in our case we are interested in the "RawRecord" branch as we know that back in DigitListModule::Reco the CandContext was programmed to point to a RawRecord. Taking this branch to bits:-

```
    AlgHandle ah = AlgFactory::GetAlgDigit();
```

This code should look familiar, its a replay of what happened in DigitListModule::Reco only now we want an AlgHandle for a AlgDigit rather than an AlgDigitList.

```
    CandContext cxx(this, cx.GetMom());
```

This too is familiar. Here a local CandContext object is created that will be used to select an individual RawDigit for AlgDigit to operate on.

Ignoring the details of the inner loops, the remainder of the code is as follows:-

```
// Iterate over RawRecord's RawDataBlockList to find DigitDataBlock.
     RawRecord *rr = (RawRecord *) (cx.GetDataIn());
     TIter rdbit = rr->RawDataBlockIter();
     TObject *tob;
     while ((tob = rdbit())) {
      ...
     }
   }
```

First the RawRecord is retrieved using the supplied CandContext. (i.e. then CandContext associated with the CandDigitList). The RawRecord is asked to provide an iterator over all its RawDataBlocks and this iterator is loop on until exhausted.

Taking the next loop:-

```
     while ((tob = rdbit())) {
     if (tob->IsA()->InheritsFrom("DigitDataBlock")) {
         DigitDataBlock *rdb = (DigitDataBlock *) tob;
         TIter rdit = rdb->GetDatumIter();
         RawDigit *rd;
         while ((rd = (RawDigit *) rdit())) {
          ...
         }
     }
```

If a particular RawDataBlock happens to be, or inherits from, a DigitDataBlock then this block is processed which involves iterating over its individual RawDigits.

The innermost loop starts:-

```
         while ((rd = (RawDigit *) rdit())) {
           cxx.SetDataIn(rd);
           CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
```

For each RawDigit the local CandDigit is programmed to point to it and the CandDigit factory method MakeCandidate is used to make a CandDigit from the CandContext and the AlgDigit. The factory returns a CandHandle to the CandDigit. Again this should look familiar from DigitListModule::Reco.

```
ch.AddDaughterLink(cdh);
```

In DigitListModule::Reco the CandHandle was given to mom, but in this case the CandDigit is handed, via its handle, to the CandDigitList's daughter list. Note that it is doing this not by sending messages directly to the CandDigitList, but to its handle.

3.4.5 AlgDigit code

All that remains to be analysed is:-

```
CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
```

Much of this is a replay of CandDigitList::MakeCandidate. A CandDigitHandle and CandDigit get created and connected and during construction of CandDigit, AlgDigit member function RunAlg get called to fill it.

This method is for the most part very straight forward:-

```
 void AlgDigit::RunAlg(AlgConfig &ac, CandHandle &ch, CandContext &cx)
{
   CandDigitHandle &cdh = (CandDigitHandle &) ch;
   RawDigit *digit = (RawDigit *) cx.GetDataIn();

// Use CandDigitHandle methods to set CandDigit member variables.
   cdh.SetChannelId(digit->GetChannel());
   cdh.SetCharge((Float_t) digit->GetADC()); // Set to 100. * p.e. count
   cdh.SetTime((Float_t) digit->GetTime());  // Set to 1. ns time units?

// For a PlexHandle we need a context
   MomNavigator *mom = (MomNavigator *) cx.GetMom();

   if (mom) {
     RawRecord *rawrec = (RawRecord *) mom->GetFragment("RawRecord");
     VldContext vldc = rawrec->GetRawHeader()->GetVldContext();

// Get a PlexHandle for PlexStripEndId to RawChannelId conversions
     PlexHandle ph(&vldc);

     PlexSEIdAltL altlist = ph.ChannelToStripEnds(digit->GetChannel());
     cdh.SetPlexSEIdAltL(altlist);
   }
}
```

This first part simply consists of transfering data from the RawDigit to the CandDigit. Again notice how modification goes via the CandDigitHandle.

The second part involves the creation of a complete list of possible strip ends as determined by the multiplexing for this digit. Here we are getting into a level of detail that is not appropriate for this chapter; the only point to note is that again CandDigit is being modified via its handle.

3.4.6 Summary

To summarise, we have seen how:-

- XxxModule::Reco is used to specify an input (CandContext) and a configured Algorithm (AlgHandle to a AlgXxx) and use them to build an output (CandXxxHandle to a CandXxx) that is then stored in MOM.
- The constructor of the CandXxx uses the AlgXxx::RunAlg to initialise it.
- AlgXxx:RunAlg modifies a new, or existing CandXxx via its CandXxxHandle.

Although this may all look a little daunting, design of Candidate classes will mostly be left to the core group and XxxModules can almost be created starting with an existing one and making a couple of global replaces, which just leaves AlgXxx and in particular AlgXxx:RunAlg, and that last really is the job of the algorithm developer!

George, I need an example of a AlgXxx::RunAlg modifying an existing CandXxx i.e. being call outside the constructor.

3.5 Working with Modules

Modules are informed of states changes such as the begin and end of the job, file and run by calls to the appropraie member function, for example:-

```
void JobCModule::BeginRun();
```

Modules also have member functions Get and Put for I/O and Ana and Reco for analysis and reconstruction processing e.g.:-

To be continued.

3.6 Working with Candidates

To be supplied.

3.6.1 Designing Candidates

To be supplied.

3.6.2 Creating Candidates

To be supplied.

3.6.3 Using Candidates

To be supplied.

3.6.4 Printing Candidates

One can prevail upon the Candidates to print themselves. Given a `CandHandle` one can use either of the following methods to induce a nested, formatted output.

```
virtual void Print(Option_t *option="nid3") const;
virtual std::ostream& FormatToOStream(std::ostream& os,
                                      Option_t *option="nid3") const;
```

In all cases the class name and number of daughters are printed. The `option` string controls various other components

"n" The object's name is printed.

"t" The object's title is printed.

"i" The object's UIDs are printed.

"d*n*" The object's daughters are printed to the *n*-th level in a nested format. Use "d0" to suppress recursing over the daughters. The lack of a *n* is taken to be a unlimited depth.

"v*n*" Verbosity level. Use "v0" to suppress any class specific data.

The class specific data portion is indented with the string "| " to help delineate the structure. This can be reset using the static method:

```
CandBase::SetDataIndent("");
```

3.6.5 Teaching Candidates to print themselves

Each Candidate class that adds new data members above and beyond those in the class it derives from must implement the method:

```
virtual std::ostream& FormatToOStream(std::ostream& os, Option_t *option="") const;
```

Here is a basic template for writing the method. Substitute the correct names for `CandDerived` and `CandDerivedFrom`. The loop over the daughter list is handled after the all the class data has been printed (and is done automagically); this doesn't handle other user additions of `TObjArrays` – the user should decide whether it is important to print those as well.

```
std::ostream& CandDerived::FormatToOStream(std::ostream& os, Option_t *option) const
{
  CandDerivedFrom::FormatToOStream(os,option);  // let base classes print their part first

  TString opt(option);
  if (!opt.Contains("v0")) { // v0 means suppress the data values
    const TString& indent = GetIndentString();

    os << indent << GetDataIndent()  // use this whenever starting a new line
                                     // for correct nesting
       << "DataVariableName " << fDataVariable  // this is (only) the data in CandDerived
       << " MoreVarNames " << fMoreVars         // CandDerivedFrom's is already printed
       << endl;
  }
  return os;
}
```

3.7 Working with Algorithms

To be supplied.

3.7.1 Designing Algorithms

To be supplied.

3.7.2 Creating Algorithms

To be supplied.

3.7.3 Using Algorithms

To be supplied.

Chapter 4

Conventions

Last significant change: 2002/12/04

4.1 Coordinate Systems and Plane Numbers

Coordinate systems used by MINOS are right-handed. When referenced to the outside world a coordinate system of x-y-z is normally used. This has a $+z$ direction that is the horizontal component of the beam direction, $+y$ that is locally vertical and $+x$ chosen to make it right-handed. For the near and far detectors it is convenient to refer to x-y-z as *detector west*, *up*, and *detector north* even though the axis directions aren't exactly compass west and north.

Plane numbers in each detector count steel planes and start at zero. Scintillator planes are referenced by the steel to which they are attached on the downstream side. Thus the correct pairing is *active+passive*. For the near and far detectors the first (zeroth) steel plane has no active plane upstream. The origin of the z axis is close-to but upstream of the front face of the first plane (steel or scintillator).

Each detector has a well defined u-v-z system, with the u-v origin at the center of the coil[1] as shown in Figure 4.1. Planes orientations are referred to by the axis which the individual strips map out (and not the axis along which they lie); strip numbers start with zero and increase with increasing u or v. Only in the CalDet does u-v correspond to x-y.

[1] The u-v origin is at the center of the plane for the CalDet which has no coil, and which differs from the other detectors in numerous other small ways.

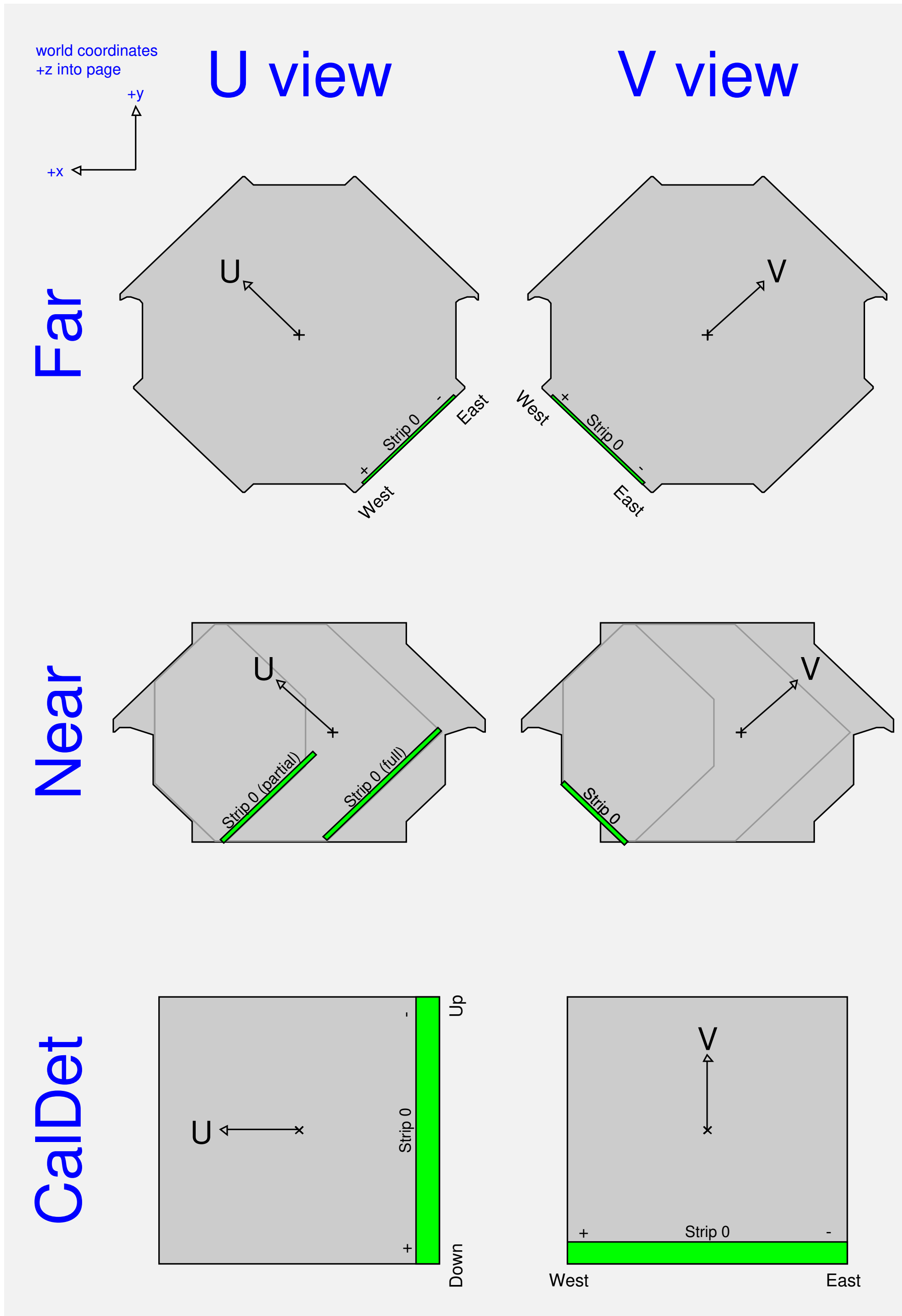

Figure 4.1: Definition of coordinate systems, plane views, strip numbering, and strip ends for all three MINOS detectors.

4.2 Enumerations

The offline code uses enumeration types to give specific user-friendly names to a related collection of values. Coding conventions require the enumerations to be put in a namespace or class and that the specific value names start with the letter 'k' (indicating a constant).

4.2.1 DetectorType

The enumeration for distinguishing between the three detectors is one of type `DetectorType::Detector_t`[2]. Variables of this type can take on the values: `DetectorType::kNear`, `DetectorType::kFar` and `DetectorType::kCalDet`. Code that needs to differentiate between the detectors should always use one of these symbolic values.

The specific values given are equivalent to bit masks but normal user code should **not** use this feature. In general user code should never make reference to the specific values that the symbolic names are assigned to; only the symbolic values should be used.

4.2.2 SimFlag

Often it is necessary to distinguish between data from the DAQ and simulation events, generally for purposes of database interactions where calibration, plex, geometry constants differ. The `SimFlag::SimFlag_t` type is used for this purpose with values: `kData`, `kDaqFakeData`, `kMC`, and `kReroot`. At this time `kDaqFakeData` and `kMC` are not used. The `kReroot` value indicates data that derives from the GEANT3 (fortran) based GMINOS that is post-processed via `rerootjob`.

4.2.3 Plane View and Coverage

Plane orientations form another enum list; the `PlaneView::PlaneView_t` values `kU`, `kV` are used by all three detectors. The CalDet also has some planes of the `kB` orientation with the plane horizontal and the strips running parallel to the beam direction.

[2]The definition of this type is defined in `Conventions/DetectorType.h`. Similar include files for the `SimFlag`, `PlaneView`, `PlaneCoverage`, `StripEnd`, `ElecType` and `ReadoutType` enumerations discussed below are also found in the Conventions package.

Scintillator planes provide different coverages of the steel to which they are attached. Bare steel planes take on a `PlaneCoverage::PlaneCoverage_t` value of `kUninstrumented`; `kNoActive` is provided as a synonym. All far detector and CalDet planes with scintillator attached have values of `kTotal` (`kComplete` is a synonym), while the near detector planes can be either `kNearFull` or `kNearPartial`.

The far detector veto shield is problematic in regard to both plane view and coverage as the orientations form no regular pattern *and* change over time. These necessitate an extension of the `PlaneView` and `PlaneCoverage` types to have special cases to map out all the veto shield alternatives. For further information on how the veto shield is handled see Section 5.7.

4.2.4 StripEnd

Each scintillator strip has a "–" (`kNegative`) and a "+" (`kPositive`) end. The symbolic constants are of type `StripEnd::StripEnd_t`. For the near and far detectors these are conventially referred to as the *east* (`kEast`) and *west* (`kWest`) end. For the CalDet convention refer to Figure 4.1. In the near and far detectors some *logical* strips are composed of two physical strips. These *subpart*s are referred to by the same `StripEnd` convention which also includes a `kWhole` value.

4.2.5 Electronics Readout Types

Not all raw electronics channels are attached to PMT pixels that recieve light from scintillator strips in normal planes. Electronics channels can be mapped (see Chapter 5) to an identifier of type `ReadoutType::Readout_t` that distinguishes between: `kScintStrip`, `kPinDiode`, `kFlashTrigPMT`, `kVACommonMode`, `kTriggerOR`, `kCerenkov`, `kTOF` and `kTimingFid` readouts. The last three should only ever occur in the CalDet data. Unexpected channels are tagged with `kUnknown`.

Electronics channel and pixel identifiers use the `ElectType::Elec_t` type found in `Conventions/ElecType.h` to differentiates between `kVA` and `kQIE` electronics.

4.3 Units and Physical Quantities

The `Conventions/Munits.h` (Minos units) header defines a namespace in which a number of conversion factors are given. Each value represents the scaling factor which converts a quantity in units of that type to the base unit of the same dimensionality (length, time, mass, etc.). In general the base units are the traditional S.I. MKS system. Users need to convert quantities into the base units only at the point they are importing or exporting the value to/from any series of calculations that are done within the system. These points of conversion include:

- Import:
  - hardcoded values in the source code.
  - values returned from packages that don't follow the convention; if the package supplying the values does use the convention then there is no need to perform any conversion.
  - input from the user or database.
- Export:
  - when calling another package outside of the convention.
  - when writing to a database or filling a histogramming.
  - when printing; the user should also specify the units being written as part of the output text string.

If these conventions are followed then there is no need to perform any conversions during calculations except where necessary to interface to non-conforming packages. Additionally, this consistency ensures that the result of any calculation will be independent of what the underlying base unit is. Geant4 uses two independent versions of `G4Units.h` to test their code – if the code is consistent then the choice of base units is irrelevant and one gets exactly the same output.

```
#include "Conventions/Munits.h"
//
// macroscopic quantities use MKS as base units
//
// Conversion of quantities from external sources to base units.
```

```
// One does this conversion when a value enters a calculation stream
// from a hardcoded values in the source code, user input,
// or the database.
//
Double_t rise       = 10.0 * Munits::cm;
Double_t run        =  4.0 * Munits::meter;
Double_t macro_mass =  1.0 * Munits::kg;
Double_t charge     = Munits::eplus; // unit charge in Coulombs
Double_t bfield     = 15 * Munits::kilogauss;
Double_t velocity   = 3.1e7 * Munits::m/Munits::s;
// if column in database table were stored in mm
Double_t offset     = dbiData->GetAlignmentOffset() * Munits::mm;
// if user config info was requested in cm
Double_t fAnaZmin   = command->PopFloatOpt() * Munits::cm;
// if xpkg package doesn't conform to using Munits
Double_t silly      = xpkgObj.GetDistance() * Munits::inch;

//
// use of quantities
//
Double_t angle_rad = TMath::ATan2(rise,run);
Double_t force = charge * velocity * bfield;

//
// conversion to alternative units for output, including
// printing, histogramming and writing to the database.
//
cout << "the 'run' distance in km is " << run / Munits::km << endl;
cout << "the angle in degrees is " << angle_rad / Munits::degree << endl;
// fill a histogram who's axis is in mm
myhist->Fill(rise/Munits::mm);
// call something in the xpkg package expecting feet
xpkgObj.SetFeetLength(offset/Munits::feet);
```

The exception to the above is that particle energies, masses and momenta are expected to be handled in a HEP system with a base unit of GeV and where the speed of light, c, is 1.

```
// use of GeV as base unit for particles (implicit c=1)
Double_t pmass   = 938.27231 * Munits::MeV;
Double_t ptot    = 10.0 * Munits::GeV;
Double_t penergy = TMath::Sqrt(ptot*ptot+pmass*pmass); // implicit c=1
Double_t gamma   = energy/pmass;
Double_t beta    = ptot/energy;  // = TMath::Sqrt(1.-1./(gamma*gamma));
```

For those prefer to work in an all MKS system and explicitly deal with all the factors of c the correct procedure is:

```
Double_t energy_in_joules     = penergy * Munits::hep2baseEnergy;
Double_t mass_in_kg           = pmass   * Munits::hep2baseMass;
Double_t macroscopic_momentum = ptot    * Munits::hep2baseMomentum;

Double_t pc    = macroscopic_momentum * Munits::c_light;
Double_t pc_2  = pc*pc;
Double_t mc2   = mass_in_kg*Munits::c_2
Double_t mc2_2 = mc2*mc2
Double_t cross_check = TMath::Sqrt(pc_2+mc2_2);
cout << " energy in joules " << energy_in_joules << " " << cross_check << endl;
```

If the the overabundance of the `Munits::` qualifier seems onerous one can eliminate the need to explicitly qualify the namespace by use of:

```
using namespace Munits;

// now one is free to simply give the unit's name, but take care that
// simple variable names such as "m" and "cm" will need to be qualified
// if they are obsured by local variables.  No warning will be given.

Double_t cm = 12345;
cout << " cm:  local " << cm << "   Munits:: " << Munits::cm << endl;
// results in " cm:  local 12345.000000   Munits:: 0.010000"
```

In addition to the unit conversions there are a few other handy numbers, mostly physical quantities, that are available. A few, necessary for unit conversion, are in the `Munits::` namespace while the others are in the `Mphysical::` namespace. Use of `Conventions/Mphysical.h` brings `Conventions/Munits.h` along with it.

length	*meter, m* millimeter, mm, centimeter, cm, kilomenter, km micrometer, nanometer, angstrom, fermi inch, foot
length^2	*meter2, m2* millimeter2, mm2, centimeter2, cm2, kilometer2, km2 barn, millibar, microbarn, nanobarn, picobarn
length^3	*meter3, m3* millimeter3, mm3, centimeter3, cm3, kilometer3, km3
angle	*radian, rad* milliradian, mrad, degree, deg
solid angle	steradian, sr
time	*second, s* nanosecond, ns, millisecons, ms, picosecond
frequency	*hertz* kilohertz, megahertz
macroscopic energy	joule
macroscopic mass	*kilogram, kg* gram, g, milligram, mg
force	newton
pressure	hep_pascal (to avoid keyword) bar, atmosphere
electric charge	coulomb
electric current	ampere milliampere, microampere, nanoampere
electric potential	volt, kilovolt, megavolt
electric resistance	ohm
electric capacitance	farad millifarad, microfarad, nanofarad, picofarad
power	watt
magnetic flux	weber
magnetic field	tesla gauss, kilogauss
inductance	henry
temperature	kelvin
amount of substance	mole
activity	becquerel, curie
absorbed dose	gray
luminous intensity	candela
luminous flux	lumen
illuminance	lux
scales	perCent, perThousand, perMillion

Table 4.1: Units for macroscopic quantities available in Munits.h

HEP particle energy (c=1)	*gigaelectronvolt, GeV* electronvolt, eV kiloelectronvolt, keV megaelectronvolt, MeV teraelectronvolt, TeV petaelectronvolt, PeV
HEP particle momenta HEP particle mass	GeV, *etc.* GeV, *etc.*
energy conversion momentum conversion mass conversion	hep2baseEnergy hep2baseMomentum hep2baseMass

Table 4.2: HEP units (implicit $c = 1$) available in Munits.h

positron charge in Coulomb	e_SI
positron charge in base units	eplus
speed of light (c) non-HEP	c_light
c^2	c_squared, c_2
c^4	c_4

Table 4.3: physical quantities available in Munits.h

π & co.	pi, twopi, halfpi, pi2
particles/mole	Avogadro
speed of light (c), non-HEP	c_light, c_squared
Plank's constant	h_Plank, hbar_Planck, hbarc, hbarc_squared
electron charge (-eplus)	electron_charge, e_squared
simple masses, non-HEP	electron_mass_c2, proton_mass_c2, neutron_mass_c2, amu_c2, amu
free space	mu0, epsilon0
electromagnetic coupling	elm_coupling, fine_structure_const
	classic_electr_radius, electron_Compton_length, Bohr_radius
	alpha_rcl2, twopi_mc2_rcl2
Boltzmann constant	k_Boltzmann
STP	STP_Temperature, STP_Pressure, kGasThreshold
gravitational constant	G_N

Table 4.4: physical quantities available in Mphysical.h

Chapter 5

Plexus

Last significant change: 2004/03/11

plex·us (plĕk'sus) *n. pl.* **-us** or **-us·es.** *Anat.* A structure in the form of a network, esp. of nerves, blood vessels, or lymphatics: *the solar plexus.* [< Lat., p.p. of *plectere,* to plait.]

5.1 Introduction

The `Plex` package provides services for managing the logical connection between elements of the detector above the physical level. Thus it is not concerned with the placement of physical scintillator strips and steel sheets in 3-space, but instead is responsible for providing the navigation from strips to fibers to pixel spots to pixels (anodes) to photomultiplier tube and so forth. It allows reverse navigation as well (e.g. one-to-many as well as many-to-one).

A connection map representing one of the detectors is valid for a particular time and may evolve as components are added or re-configured (e.g. to correct a mis-cabling). Thus instantiating a plexus must involve constructing it for a particular detector at a particular time in its existence – a plexus will be constructed that has a time range that covers the requested time. The underlying mapping information will come from the database.

Diagram 5.1 is a schematic sketch of some of the detector components and how they relate. Figure 5.2 is an illustrative (i.e., non-authoritative) representation of the same concepts as seen from a UML[1] perspective.

[1]Unified Modelling Language, *c.f.* `http://www.rational.com/uml/index.jtmpl`

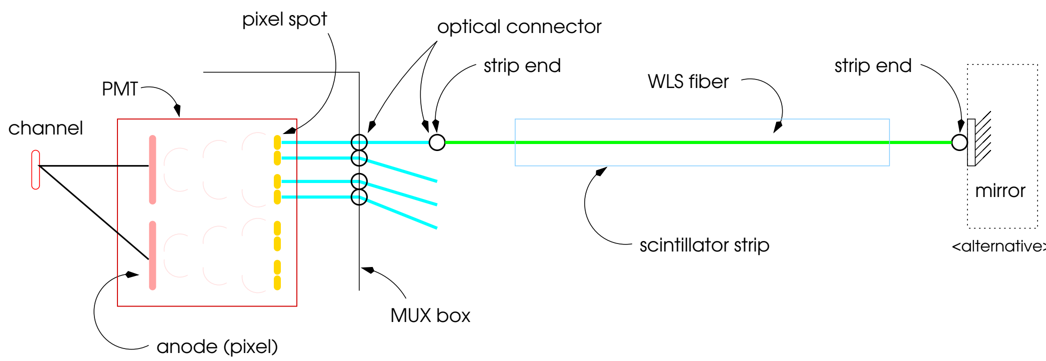

Figure 5.1:

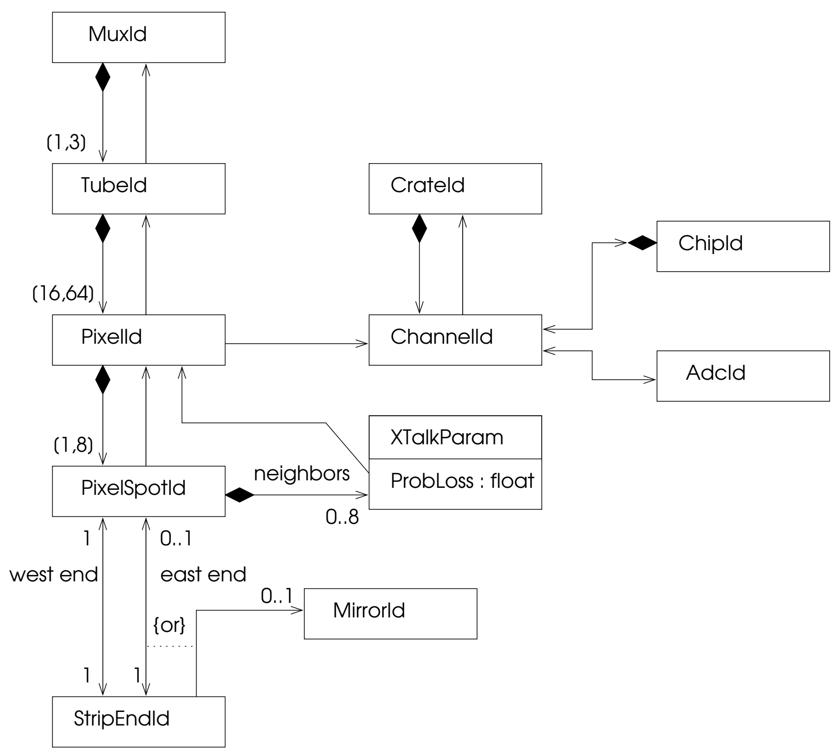

Figure 5.2:

5.2 Creating a plexus

The user does not work directly with a `Plexus` object, but instead works through a `PlexHandle`. These `PlexHandle`s are lightweight proxy objects that do little work of any substance. During the creation of a `PlexHandle` a *loan pool* of real `Plexus` objects searched for an appropriate object; if no match exists one will be created. The `PlexHandle` then forwards member function calls to the correct `Plexus`. This allows the `PlexHandle` to share the larger underlying collection of information and it lowers the overhead of the creation and deletion of `PlexHandle`s.

This approach also allows the `PlexHandles` to be created on the stack and thus avoids a common source of memory leaks. The lookup procedure is so fast that users should not be concerned with (or even attempt) to retain a handle, but rather should construct one from a `VldContext` whenever one is needed.

```
#include "RawData/RawRecord.h"
#include "Plex/PlexHandle.h"

  // somehow acquire a raw record, perhaps:
  RawRecord *rawrec = (RawRecord*) (mom->GetFragment("RawRecord"));

  // retrieve a copy of the VldContext object identifying the configuration
  VldContext vldc = rawrec->GetRawHeader()->GetVldContext();

  // create a PlexHandle (on the stack)
  PlexHandle ph(vldc)

  [....]
  // work with PlexHandle and it gets automagically destroyed
  // when the code block ends, but the next invocation is
  // likely to acquire the same underlying Plexus without
  // significant overhead.
```

5.3 Configuring the Plex

The behaviour of the plex package can be modified by manipulating the configuration of the `PlexLoanPool`. To change a parameter from within C++ code one writes code like:

```
   // PlexLoanPool is a CfgConfigurable
   PlexLoanPool* plpool  = PlexLoanPool::Instance();
```

```
plpool->Set("...");
plpool->Set("...");
plpool->Update();
```

On startup the interface checks the environmental variable `ENV_PLEX` which can contain a semi-colon separated list of configuration requests, e.g.:

```
setenv ENV_PLEX "MaxUnref=2; Cache='/tmp/myplex.cache'; CacheWrite=0"
```

One can also show the current status of the `PlexLoanPool` using:

```
PlexLoanPool::Instance()->Print();
```

which gives something like:

```
"PlexLoanPool Configuration", 3 entries. keys unlocked, values unlocked
    Cache = ''
    CacheWrite = 1
    MaxUnref = 1
 --- Shared PlexLoanPool ---
 Plexus has 1 references
    |CalDet|Data|
         2001-07-24 00:00:04.000000000Z
         2001-08-06 00:00:00.000000000Z
         from source: , From Database
 --- Private PlexLoanPool ---
End of PlexLoanPool
```

5.3.1 Managing shared plexii

As described in Section 5.2 the user normally interfaces with the package via the class `PlexHandle`. This allows the plex package to maintain a pool of `Plexus` objects to which handles can easily be attached as needed. Users might need to trade off memory requirements of holding multiple plexii vs. the time requirement of repeatedly building one for the pool. Users can configure the number of unreferenced plexii that are retained by the pool for shared use via the `MaxUnref` config value.

5.3.2 Caching a copy of the loan pool

During development when prototyping code in a rapid cycle the overhead of reconstructing a plex every time the job is run can be an annoyance. Some time can be save, of order $\sim$ 40s of $\sim$ 70s, by writing out the plex as a `root` file and reading it back rather than pulling the data from the database and building it from scratch. This approach can also be dangerous in that it bypasses the database and thus heavy reliance on it can lead to the use of outdated constants. It also has no protection against multiple jobs trying to read and write the file simultaneously.

Nevertheless, it is sometime useful. This behavour can be enabled by setting the `Cache` config variable to the name of the root file to use. One then runs the job once to build the necessary plex from the database and the job will write the file upon teardown of the `PlexLoanPool` at job end. Subsequent jobs will then read the cached file and can avoid the re-writing overhead by setting the `CacheWrite` config value to zero.

5.4 PlexStripEndId

The most frequently encountered identifier is a `PlexStripEndId`. This identifies a logical scintillator *strip-end*; it encapsulates identifiers for the physical detector, plane, strip-in-plane it refers to. It also, when meaningful, identifies which end of the strip in the case of two-sided readout. The components can be extracted via `Get` methods. It derives from `PlexPlaneId`, so some of the `Get` methods are implemented there. The most common comparisons to other `PlexStripEndId`s are available via methods in the class:

```
#include "MessageService/MsgService.h"
#include "Plex/PlexStripEndId.h"

  PlexStripEndId seid = [...]
  PlexStripEndId another = [...]

  // decompose PlexStripEndId into components
  DetectorType::Detector_t detector = seid.GetDetector();
  PlaneView::PlaneView_t   view     = seid.GetPlaneView();
  UShort_t                 plane    = seid.GetPlane();
  UShort_t                 strip    = seid.GetStrip();
  StripEnd::StripEnd_t     end      = seid.GetEnd();

  // seid.Print() might yield "[   Far|   2 Ut|123|*E]"
  // for the far detector, plane 2 (U view, Total coverage),
  // strip 123, whole strip, east end.

  // these tests should output exactly one message
```

```
if ( ! seid.IsSameStrip(another) )
   MSG("XXX",Msg::kInfo) << " not the same (detector,plane,strip)" << endl;

if ( seid.IsSameStripEnd(another) )
   MSG("XXX",Msg::kInfo) << " same (detector,plane,strip,end)" << endl;

if ( seid.IsOppositeStripEnd(another) )
   MSG("XXX",Msg::kInfo) << " same (detector,plane,strip) && different (end)" << endl;
}
```

5.5 `PlexSEIdAltL`

The `PlexStripEndId` to assign to a readout digitization is not unique because of optical (far detector) or electrical (near detector spectrometer) summing; this effect is colloquially, but mistakenly, referred to as "multiplexing". The `PlexSEIdAltL` is a list of `PlexStripEndId` alternatives. Starting with a `RawChannelId` the plexus can construct a list and hand it back to the user for refinement. To facilitate the refinement the `PlexSEIdAltL` manages the list with a weight associated with each alternative. Mechanisms for iterating over the list, setting the associated weights, removing entries and extracting the the `PlexStripEndId`s and weights are provided. The interpretation of the *weight* value is not specified – it could be $-\chi^2$, log-likelihood, etc. Any algorithm must be consistent in its usage; the only expectation is that large values are "better". There are a number of broad general categories of methods for the `PlexSEIdAltL` class: characterization, iteration, weight modification, addition, deletion.

An unexceptional list will normally have a consistent plane number and strip end enumeration and so `GetPlane()` and `GetEnd()` are generally well defined operations. It is conceivable that a miscabling could mix signals from different planes so that the mapping from `RawChannelId` to a list of `PlexStripEndId`s can't give a unique answer, but currently that isn't handled other than to note, as a printed warning, the exception.

The `PlexSEIdAltL` is a STL[2] vector of `PlexSEIdAltLItem`s each of which represent an alternative `PlexStripEndId` and associated fields. The associated data includes the `PlexPixelSpotId` (distinguishes pixel spots for the far and different pixels for the near spectrometer); the assigned weight; and

[2] C++ Standard Template Library, *c.f.* `http://www.sgi.com/tech/stl/`

when generated in conjunction with real digitization information, some calibration corrected quantities, such as photo-electrons, strip-to-strip corrected signal and times.

```
#include "Plex/PlexHandle.h"
#if !defined(__CINT__) || defined(__MAKECINT__)
#include "MessageService/MsgService.h"
CVSID("$Id: plexus.tex,v 1.7 2004/06/14 19:38:45 west Exp $");
#else
#include <iostream>
using namespace std;
#define MSG(a,b) cout
#endif

void plex_tutor()
{
   Detector::Detector_t det = Detector::kFar;
   VldContext vldc(det,SimFlag::kData,VldTimeStamp());
   PlexHandle ph(vldc);

   PlexStripEndId seid(det,1,0,StripEnd::kEast);
   RawChannelId   rcid = ph.GetRawChannelId(seid);

   PlexSEIdAltL alt = ph.GetSEIdAltL(rcid);

   // loop over alternatives assigning increasing weights
   for (unsigned int i=0; i < alt.size(); ++i) {
      alt[i].SetWeight(i);
   }

   alt.Print();

   // retrieve values for the entry with the largest weight
   // the iterator cursor is left unchanged
   PlexStripEndId best_seid = alt.GetBestSEId();
   Float_t        best_wgt  = alt.GetBestWeight();
   MSG("test",Msg::kInfo)
      << "Best " << best_seid << " weight " << best_wgt << endl;

   // Iteration using SetFirst()/Next(), SetLast()/Previous()
   // is now deprecated along with all the methods involving the
   // word "Current". PlexSEIdAltL is a STL vector and CINT has
   // full access to its STL methods.

   // iterate forward, retrieve each value in turn
```

```
   alt.SetFirst();
   while ( alt.IsValid() ) {  // while the cursor is still in range
      PlexStripEndId seid = alt.GetCurrentSEId();
      Float_t        wgt  = alt.GetCurrentWeight();
      MSG("test",Msg::kInfo) << seid << " had weight " << wgt << std::endl;
      alt.Next();
   }

   // iterate backwards
   alt.SetLast();
   while ( alt.IsValid() ) {
      alt.SetCurrentWeight(alt.size()-alt.GetCurrentWeight());
      alt.Previous();
   }

}
```

The ordering of entries initially follows the order in which they were entered into the list. The list can be sorted by weight value but this must be via an explicit method call so as not to confuse iteration.

One can set all the weights to zero using `ClearWeights()`. The individual weight for the "current" position of the iterator cursor can be adjusted via `SetCurrentWeight()` or `AddToCurrentWeight()`.

There is one method, `AddStripEndId()`, for adding to the list of strip-end, weight pairs but this is the purview of the plexus itself and isn't a normal operation for the user. The user of the list though may wish to delete selected entries and a variety of such facilities are provided:

- `DropZeroWeights()` removes all entries with a weight exactly equal to zero.

- `KeepTopWeights(n,keeporder)` keeps at least the top `n` entries with the largest weights. The resulting list may have more than `n` entries if it would otherwise have to decide between two entries with identical weights. If the boolean flag `keeporder` is true than the final list retains the initial ordering, otherwise it is sorted in descending order so that the first is the entry with the largest weight.

- `DropCurrent()` removes the strip-end, weight pair pointed to by the iterator cursor. The cursor index remains unchanged but the gap is eliminated, effectively moving the cursor on to the next entry. Thus removal via this method should be done in a reverse loop:

```
    // iterate *backwards*, remove various values
    // testing each and skipping none
    alt.SetLast();
    while ( alt.IsValid() ) {
       [...]
       if (bad_choice) alt.DropCurrent();
       alt.Previous();
    }
```

5.6 Special Channels

In a number of cases (especially at CalDet) some electronics channels are used for readouts other than PMTs attached to scintillator strips. The mundane alternatives are pin diode and VA common mode readout; more specialized outputs include a PMT attached to the Flasher system; an "or" of the triggers; Cerenkov; Time-of-flight and mixed electronics timing fiducial markers.

The Plex can be queried by `RawChannelId` as to which `ReadoutType` (see Section ??) is generating data on that channel. For readout types other than `kScintStrip`, `kPinDiode` and `kVACommonMode` this information is given to the Plex via the `PLEXRAWCHANNELREADOUTTYPE` table in the database, which should be kept up-to-date whenever special channels are rewired. The `READOUTNAME` column should start with the appropriate readout type name, but can have appended to it any useful descriptive text.

Since, for some of these special readouts, there might be multiple channels of this special type one can use the extra information in the `READOUTNAME` to select specific instances. Thus given:

```
RawChannelId       READOUTNAME     ReadoutType
C:V---:0x00:1524  'FlashTrigPMT'  kFlashTrigPMT
C:V---:0x00:05b0  'CerenkovAdc1'  kCerenkov
C:V---:0x00:0590  'CerenkovAdc2'  kCerenkov
C:V---:0x00:05d0  'CerenkovAdc3'  kCerenkov
C:V---:0x00:1602  'TOFAdc1'       kTOF
C:V---:0x00:1590  'TOFAdc2'       kTOF
C:V---:0x00:15b0  'TOFAdc3'       kTOF
```

one can look up a specific special channel via:

```
RawChannelId rcid = ... // special channel...or not

// get the readout type
```

```
ReadoutType::Readout_t rtype = ph.GetReadoutType(rcid);
if ( ReadoutType::ScintStrip   != rtype &&
     ReadoutType::PinDiode     != rtype &&
     ReadoutType::VACommonMode != rtype    ) {
   // not mundane, let's ask for more info
   std::string descript = ph.GetSpecialDescript(rcid);
}

// look up the specific CerenkovAdc1 channel, if one is so labelled
RawChannelId ckv1 = ph.GetSpecialChannelContains("CerenkovAdc1");
```

5.7 Far Detector Veto Shield

The construction of the far detector veto shield is an *ad hoc* afterthought. As a consequence it suffers from a lack of regularity and consistency. The veto shield only uses C and E modules; but which is used where and in what orientation forms no regular pattern. And while all normal planes, once installed, never change their orientation this is not true for the veto shield. Additional confusion arises because a numbering scheme was used in entering the strip-to-pixel mapping that lumps together modules in different orientations, in effect having non-planar "planes". Since a `PlexPlaneId` (and a `PlexStripEndId` which derives from it) are supposed to support the concept of asking for a `PlaneView::PlaneView_t`, this leads to ambiguity.

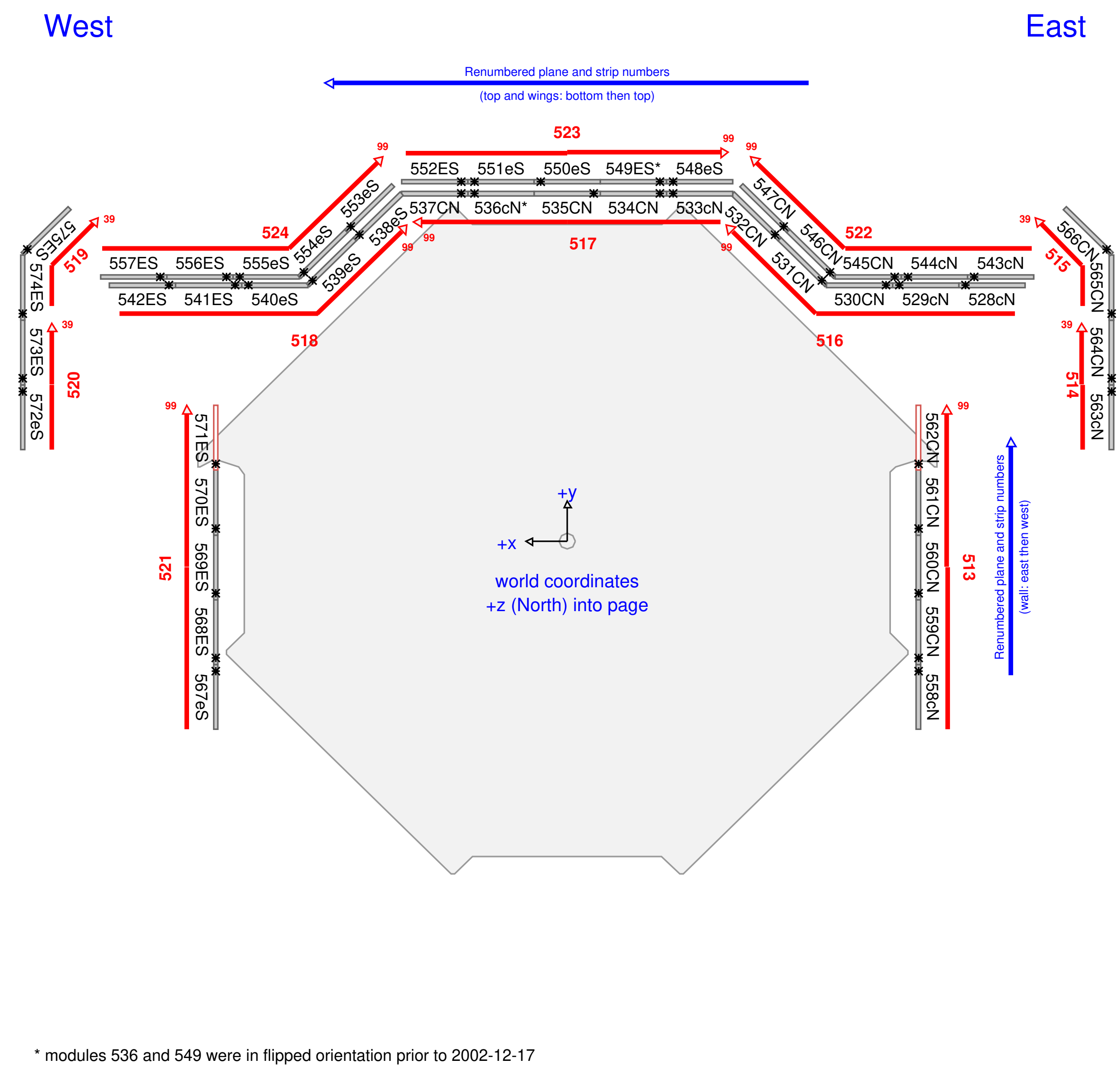

Figure 5.3: Definition of plane and strip numbering in the first section of the prototype veto shield. Red values are "as entered" into the Plex; Blue (and black) values are after renumbering so that individual modules are interpreted as independent planes. For individual modules the trailing letters indicate the module's type and orientation. The type can be one of: `c`, `C`, `e`, `E`; if the type is lowercase then the logical strip numbers count in the opposite direction as the construction/mapper numbering. For example, plane 528cN would have logical strips 0 to 19 counting east to west, but is placed so that *mapper "strip 1"* (indicated by the asterisk) is on the west side. The `N` and `S` indicate whether the *F1* connector is on the North or South side.

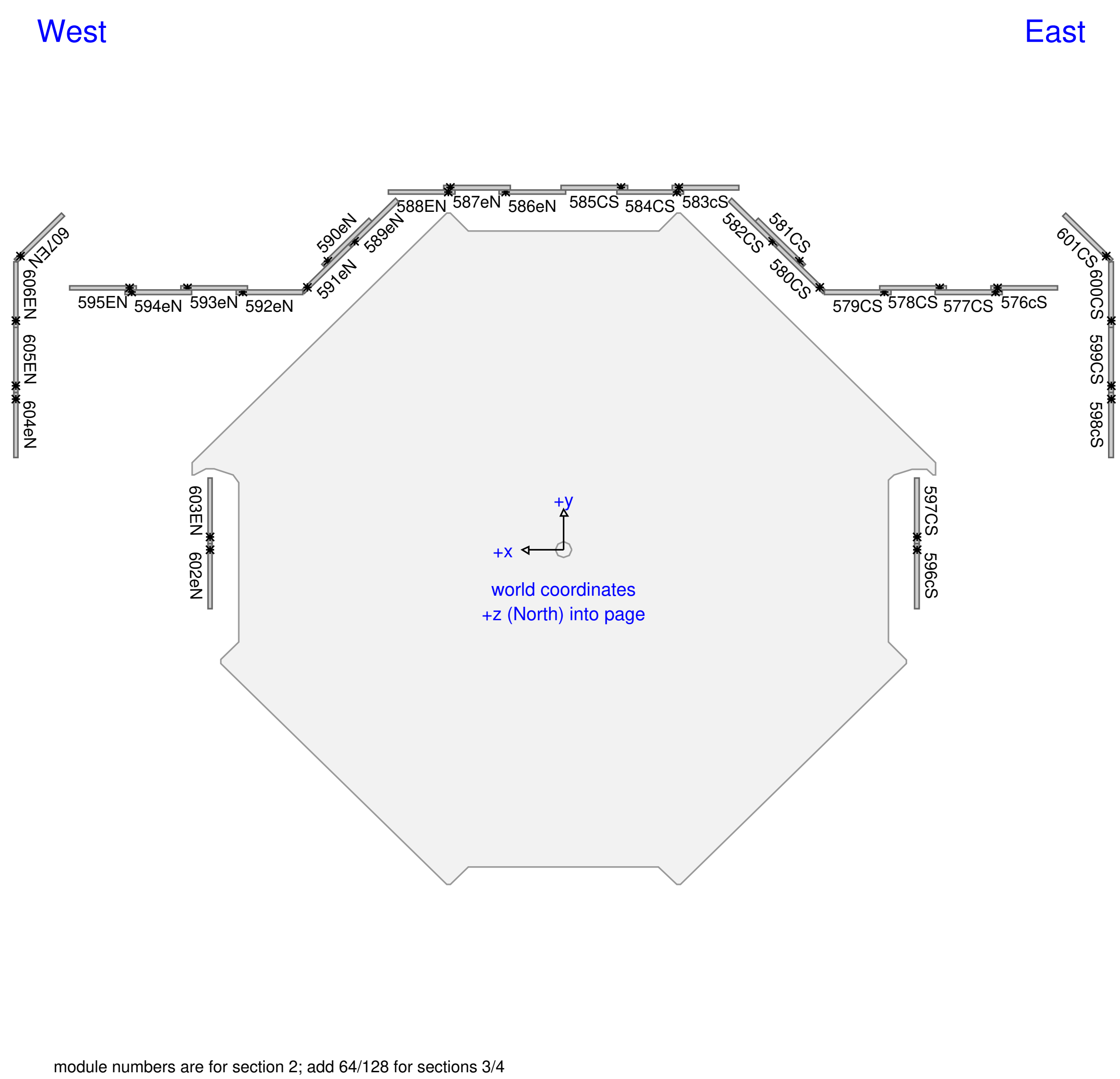

Figure 5.4: Definition of plane and strip numbering in the second (3rd & 4th) section of the prototype veto shield. Each module is entered as a separate plane.

Chapter 6

UgliGeometry

Last significant change: 2004/03/11

6.1 Introduction

The `UgliGeometry` package provides services for managing the physical placement of detector elements in 3-space.

6.2 Creating a geometry

The user does not work directly with a `UgliGeometry` object, but instead works through a `UgliGeomHandle`. These `UgliGeomHandle`s are lightweight proxy objects that do little work of any substance. During the creation of a `UgliGeomHandle` a *loan pool* of real `UgliGeometry` objects searched for an appropriate object; if no match exists one will be created. The `UgliGeomHandle` then forwards member function calls to the correct `UgliGeometry`. This allows the `UgliGeomHandle` to share the larger underlying collection of information and it lowers the overhead of the creation and deletion of `UgliGeomHandle`s.

This approach also allows the `UgliGeomHandles` to be created on the stack and thus avoids a common source of memory leaks. The lookup procedure is so fast that users should not be concerned with (or even attempt) to retain a handle, but rather should construct one from a `VldContext` whenever one is needed.

```
#include "UgliGeometry/UgliGeomHandle.h"
```

```
// retrieve a copy of the VldContext object identifying
// the configuration of the current record
VldContext vldc = [..]

// create a UgliGeomxHandle (on the stack)
UgliGeomHandle ugh(vldc)

[....]
// work with UgliGeomHandle and it gets automagically destroyed
// when the code block ends, but the next invocation is
// likely to acquire the same underlying UgliGeometry without
// significant overhead.
```

6.3 Configuring how geometries are constructed

The behaviour of the ugli package can be modified by manipulating the configuration of the `UgliLoanPool`. To change a parameter from within C++ code one writes code like:

```
// UgliLoanPool is a CfgConfigurable
UgliLoanPool* ulpool  = UgliLoanPool::Instance();
ulpool->Set("...");
ulpool->Set("...");
ulpool->Update();
```

On startup the interface checks the environmental variable `ENV_UGLI` which can contain a semi-colon separated list of configuration requests, e.g.:

```
setenv ENV_UGLI "MaxUnref=2; AlwaysUseDbi=0"
```

One can also show the current status of the `UgliLoanPool` using:

```
UgliLoanPool::Instance()->Print();
```

which gives something like:

```
"UgliLoanPool Configuration", 13 entries. keys unlocked, values unlocked
    AlgorithmicCalDet = 0
    AlgorithmicFar = 0
    AlgorithmicNear = 0
    AlwaysUseDbi = 1
    Cache = ''
    CacheWrite = 1
    CutAppliesToMC = 0
    CutAppliesToVetoShield = 0
```

```
    CutMsgLevel = 'Synopsis'
    CutOnPlnInstallCalDet = 0
    CutOnPlnInstallFar = 1
    CutOnPlnInstallNear = 0
    MaxUnref = 1
--- Frozen UgliLoanPool ---
 UgliGeometry has 1 references
    |CalDet|Data|
         2001-07-24 00:00:04.000000000Z
         2001-08-13 00:00:04.000000000Z
         from source: DBI, From Database
    fRootGeom "Frozen|CalDet|Data|2001-07-24 00:00:04"
--- Modifiable UgliLoanPool ---
--- End of UgliLoanPool ---
```

6.3.1 Managing shared geometries

As described in Section 6.2 the user normally interfaces with the package via the class `UgliGeomHandle`. This allows the ugli package to maintain a pool of `UgliGeometry` objects to which handles can easily be attached as needed. Users might need to trade off memory requirements of holding multiple geometries vs. the time requirement of repeatedly building one for the pool. Users can configure the number of unreferenced geometries that are retained by the pool for shared use via the `MaxUnref` config value.

6.3.2 Caching a copy of the loan pool

During development when prototyping code in a rapid cycle the overhead of reconstructing a geometry every time the job is run can be an annoyance. Some time can be save, of order ~ 40s of ~ 70s, by writing out the geometry as a `root` file and reading it back rather than pulling the data from the database and building it from scratch. This approach can also be dangerous in that it bypasses the database and thus heavy reliance on it can lead to the use of outdated constants. It also has no protection against multiple jobs trying to read and write the file simultaneously.

Nevertheless, it is sometime useful. This behavour can be enabled by setting the `Cache` config variable to the name of the root file to use. One then runs the job once to build the necessary geometry from the database and the job will write the file upon teardown of the `UgliLoanPool` at job end.

Subsequent jobs will then read the cached file and can avoid the re-writing overhead by setting the `CacheWrite` config value to zero.

6.4 `UgliGeomHandle`

6.5 `UgliPlaneHandle`

6.6 `UgliStripHandle`

6.7 Building, changing and sharing a modified geometry

Normally one builds a geometry that is *frozen*, i.e. unmutable, and exactly reflects the constants stored in the database. Sometimes it is useful to adjust various parameters, for instance in attempting to do alignment. The UgliLoanPool in fact manages two distinct pools of geometries. The normal `UgliGeomHandle` constructor actually has a second argument beyond the `VldContext`. The second argument specifies from which pool to draw.

```
UgliGeomHandle cold_geom(vldc,Ugli::kFrozen);
UgliGeomHandle playdough_geom(vldc,Ugli::kModifiable);

UgliGeomHandle oneofthetwo_geom(vldc);
```

The `cold_geom` and `playdough_geom` are distinct geometries. The third geometry, since it has a compatible `VldContext`, is guarenteed to be one of the two previously created. But exactly which depends on the state of the *DefaultModifyMode*. This allows one to set the default so that an code written using the single argument for the constructor can be toggled:

```
Ugli::SetDefaultModifyMode(Ugli::kModifiable);
// run a job path to do some tracking/fitting
// it will run using the modified geometry
// even if it wasn't explicitly written to do so.
[...]
// run a chi2 check against the modified geometry
[...]
Ugli::SetDefaultModifyMode(Ugli::kFrozen);
// run a chi2 check against the frozen (Database) geometry
[...]
```

Example code demonstrating some of the adjustment capabilities is show below:

```
UgliStripHandle    ush  = playdough_geom.GetStripHandle(stripid);
UgliScintPlnHandle usph = ush.GetScintPlnHandle();
UgliScintMdlHandle usmh = ush.GetScintMdlHandle();

// shift,rotate the strip relative to the module by
// some delta from current position
// L=longitudinal position, T=transverse position
//ush.ShiftLTPosRelMdlBy(0*Munits::cm,1.0*Munits::cm);
//ush.RotateRelMdlBy(2.0*Munits::mrad);

// shift,rotate module relative to scint plane by absolute amounts
usmh.SetLTPosRelPln(1.5*Munits::cm,200.0*Munits::cm);
usmh.SetZRotRelPlnRad(2.0*Munits::mrad);

// see Ugli*Handle for more details about available methods
// currently UgliSteelPlnHandle's can not be adjusted.
```

Chapter 7

Database Interface

Last significant change: 2004/09/04

This chapter is organised into the following sections.

Concepts is an introduction to the basic concepts behind the DatabaseInterface. You can skip this section if you are in a hurry, but reading it will help you understand the package.

Installing and Running provides a few tips on building running programs that use the DatabaseInterface.

Accessing Existing Tables tells you how you write code to retrieve data from existing tables.

Creating New Tables describes how new tables are added to the database and the corresponding classes, that serve the data, are designed.

Filling Tables explains how new data is added to existing tables in the database.

MySQL Crib gives the bare minimum necessary to use MySQL to manage a database. The DatabaseInterface runs directly on top RDBC under which MySql, ORACLE and flat ASCII files will used to implement a hierarchical database. Currently only MySQL is supported

7.1 Concepts

7.1.1 Types of Data

Besides the data from the detector itself, off-line software requires two additional types of data:-

Detector Description i.e. data that describes the construction of the detector and how it responds to the passage of particles through it. The geometry, the cabling plea and calibration constants are all examples of this type of data.

Algorithm Configuration i.e. data used to control the way an algorithm operates. This includes such things as software switches to control flow and tunable cuts.

It is the purpose of the DatabaseInterface to provide simple and efficient access to such data and to provide a framework in which new types of data can be added with minimal effort.

7.1.2 Simple, Compound and Aggregated

Within the database, data is organised into tables. When the user requests data from a table, the DatabaseInterface collect rows of data from the appropriate table. From the perspective of the interface, there are 3 types of organisation:-

Simple A single row is retrieved. Algorithm Configuration data is always simple; even if multiple configurations are possible, only one can be selected at a time. Detector Description, on the other hand, is almost never Simple.

Compound Multiple rows are retrieved. Each row represents a single sub-system and the request retrieves data for a complete set of sub-systems. For example a request for plane positions will produce a set of rows, one for each plane.

Aggregated A special form of Compound depending on the way new data is added to the database:-

- If data for the entire detector is written as a single logical block, then it is Compound. A table that describes the way PMT pixels map to electronics channels might be compound: a complete description is written as a single unit
- If it is written in smaller chunks (called aggregates) then it is Aggregated. For example light injection constants come from pulser data and it is quite possible that a calibration run will only pulse some LEDs and so only part of a full detector set of constants gets written to the database. One of the jobs of the interface is to reassemble these aggregates so that the user only ever sees a complete set.

7.1.3 Tables of Data

The DatabaseInterface provides a simple, uniform concept regardless of the data being accessed. Each request for data produces a pointer giving read access to a results table, which is effectively a slice of the underlying database table. Each row of the results table is an object, the type of which is table-specific. These table row objects give access to the data from one row but can hide the way the database table is organised. So changes to the physical layout of a database table should only effect its table row object, not the end users of the data. Note that a single request only ever accesses a single table; the interface does not support retrieval of data from multiple database tables simultaneously.

If the request for data fails for some reason, then the resulting table will be empty, otherwise it will have a single row for Simple organisation and more than one row for Compound and Aggregated. The user can ask how many rows the table has and can directly access any of them. The physical ordering of the rows in the table reflects the way the data was originally written, so for Aggregated data, the ordering is not optimised for retrieval. To deal with this, each table row object can declare a natural index, independent of its physical position, and this natural index can be used to retrieve data.

7.1.4 A Cascade of Databases

The DatabaseInterface can access data for more than one database. During initialisation it is given a list of database URLs. The list order reflects priority; the interface first looks for data in the first database in the list, but if that fails, tries the others in turn until all have been tried or data is found. This scheme allows a user to override parts of the official database by creating a mini-database with their own data and then placing it in the list ahead of the official database. The concept of a set of overlaying databases is called a cascade.

7.1.5 Context Sensitive

In principle, any of the data retrieved by the interface could depend on the the current event being processed. Clearly Detector Descriptions, such as calibration constants, will change with time and the interface has to retrieve the right ones for the current event. Although less likely, its possible that Algorithm Configurations too could change to reflect changes in the detector. For this reason, all requests for data through the interface must supply information about the:-

- The type of data: real or Monte Carlo.
- The type of detector: Near, Far or Calibration.
- The date and times of the event.

Collectively this information is called the *Context* and is represented by the `VldContext` class of the Validity package.

In the database all data is tagged by a *Validity Range* which identifies the types of data and detector and the ranges of date times for which it is valid. This is represented by the *VldRange* class of the Validity package. Some data, typically Algorithm Configuration, is universal; the same database data can be used for any event. Others may be very specific to a single type of data and detector and a limited date time range. Physically the way this is done is to have a pair of tables:-

Validity Range Table This table consists of rows of VldRange objects, each with a unique sequence number which is used as a key into the Main Data Table.

Main Data Table Each row has a sequence number corresponding to an entry in the Validity Range Table.

The interface first finds a match in the Validity Range Table for the current context and then retrieves all rows in the Main Data Table that match its sequence number. The reasons for this two step approach are:-

- To simplify the task of Validity Management.
- To avoid repeated data. For Compound and Aggregated data, many rows can share a single Validity Range. So this range only appears once and only a simple sequence number has to be repeated in the main table.

7.1.6 Extended Context

The primary function of DatabaseInterface is to provide the best information for a specific context, but it can also retrieve information for much more general queries. The query is still broken into two parts: the "context" which is matched to the Validity Range Table and then the data from the main table is taken for the selected sequence number(s). However the user can supply a context such as "All ranges that start between this time and that time" hence the term "Extended Context". Further, during the retrieval of data from the main table addition restrictions can be imposed. The result of an Extended Context query is a collection of rows that will not normally represent the state of the detector at a single moment in time and it is up to the user to interpret the results meaningfully.

7.1.7 SimFlag Association

As explained in the preceding section, the interface finds the database data that best matches the context of the data. There are occasions when this matching needs to be changed, for example there can be times when Monte Carlo data needs to be treated exactly as if it were event data and this includes the way it retrieves from the database. To support this the user can specify, for any type of data, an associated list of data types. If this is done then, instead of using the current type, each of the alternative types are tried until a match is found. This matching takes precedence over the cascade i.e. all associated types are tried on the first database in the cascade before moving on to the second and subsequent cascade members. This ensures that higher members, which might even refer back to the ORACLE database at FNAL, are only tried as a last resort.

7.1.8 Authorising Databases and Global Sequence Numbers

As explained in the previous section, sequence numbers in the Validity Range Table are unique. However this can present a problem if the same type of data is being entered into several different databases. For example calibration constants will be created in the Near, Far and Calibration detectors. Eventually the tables will be merged but it is essential that there is no conflict in the sequence numbers. To solve this problem, certain databases are special: they are able to produce globally unique sequences numbers. They do this as each is allocated a unique block of 10,000,000 sequence numbers (which is enough to allow a new entry to be made every minute for 20 years!). These blocks are recorded in a special table: `GlobalSeqNo` that holds the last used sequence number for each table. The block 1..9,999,999 is used for local sequence numbers i.e. ones that are only guaranteed unique within the current database table.

So long as the cascade includes at least one authorising database then new data will be written with global sequence numbers. For temporary data, or if no authorising database, then a local sequence number are used.

Important:-

- *Merging database tables that have local sequence numbers will require a special procedure to avoid conflicts.*
- `GlobalSeqNo` *tables must never be propagated between databases.*

7.1.9 Validity Management

For constants that change with time (if that is not a contradiction in terms!) it makes sense to have overlapping Validity Ranges. For example, suppose we know that a certain sort of calibration constants drifts with time and that, once determined, is only satisfactory for the next week's worth of data. A sensible procedure would be to limit its validity to a week when writing to the database but to determine new constants every few days to ensure that the constants are always "fresh" and that there is no danger that there will be a gap. However, this means that the interface has to perform two types of Validity Management:-

Ambiguity Resolution When faced with two or more sets of data the interface has to pick the best. It does this simply by picking the one with the latest creation date time.

Validity Range Trimming Having found the best set, the interface wants to know how long it will remain the best. Any set whose creation date is later will be better according to the above rule and so the retrieved data has its validity trimmed so as not to overlap it. This reduced Validity Range is called the *Effective Validity Range.* This only happens in memory; the database itself is not modified, but it does mean that the interface does not need to check the database again for this set of data until the Effective Validity Ranges has expired. This trimming also applies between databases in a cascade, with sets in higher priority databases trimming those in lower ones.

Overlay Creation Dates As explained above, creation dates play a crucial role in resolving which set of data to use; later creation dates take priority over earlier ones. This scheme assumes that constants from earlier runs are created before constants from later runs, but this isn't always true. When improving e.g. calibration constants, it's quite normal to recalibrate recent runs before going back and fixing earlier ones and then, simply to use the date when the constants were created would mean that the constants from earlier runs would take priority over any later runs they overlapped. To allow constants to be created in any order the interface provides a system for deducing the best creation dates for any constants as follows:-

- A query is made using as the context, the start of the validity for the new constants.
- If the query finds no data, the creation date of the new constants is set to its validity start date.
- If the query finds data, the creation date of the new data is set to be 1 minute greater than the creation date of the found data i.e. just late enough to replace it.

The scheme means that creation dates always follow that dates of the runs that they correspond to rather than the dates when their constants were created. When using the scheme its probably better to consider the "dates" to be version numbers.

7.1.10 Rollback

The database changes almost constantly to reflect the state of the detector, particularly with regard to the calibration constants. However this can mean that running the same job twice can produce different results if database updates that have occurred between the two runs. For certain tasks, e.g. validation, its necessary to decouple jobs from recent updates and this requires database rollback i.e. restoring the database to a previous state. Rollback works by exploiting the fact that data is not, in general, ever deleted from the database. Instead new data is added and, by the rules of Ambiguity Resolution (see the previous section) supersede the old data. All data is tagged by the date it was inserted into the local database, so rollback is implemented by imposing an upper limit on the insertion date, effectively masking out all updates made after this limit.

7.1.11 Lightweight Pointers to Heavyweight Data

One of the interface's responsibilities is to minimise I/O. Some requests, particularly for Detector Configuration, can pull in large amounts of data but users must not load it once at the start of the job and then use it repeatedly; it may not be valid for all the data they process. Also multiple users may want access to the same data and it would be wasteful for each to have their own copy.

To deal with both of the above, the interface reuses the concept of a handle, or proxy, that appears in other packages such as Candidate. The system works as follows:-

1. When the user wants to access a particular table they construct a table-specific pointer object. This object is very small and is suitable to be stack based and passed by value, thus reducing the risk of a memory leak.

2. During construction of the pointer, a request for data is passed down through the interface and the results table, which could be large, is created on the heap. The interface places the table in its cache and the user's pointer is attached to the table, but the table is owned by the interface, not the user.

3. Each request for data is first sent to the cache and if already present then the table is reused.

4. Each table knows how many user pointers are connected to it. As each pointer is discarded by its owner, it disconnects itself from the table it points to.

5. Once a table has no pointers left it is a candidate for being dropped by its cache. However this is not done at once as, between events, there are likely to be no user pointers, so just because a table is not currently being pointed to, it doesn't mean that it won't be needed again.

7.1.12 Natural Table Index

For Detector Description data, tables can be large and the user will require direct access to every row. However, the way the table is arranged in memory reflects the way the data was originally written to the database. For Simple and Compound data the table designer can control this organisation as complete sets are written as a single unit. For Aggregated data, the layout reflects the way aggregates are written, for example, for pulser data, all the strip ends illuminated by an LED will appear together in the table. This allows the interface to replace individual aggregates as their validity expires. However this means that the physical layout may not be convenient for access. To deal with this table row objects, which all inherit from `DbiTableRow` are obliged to return a Natural Table Index, if the physical ordering is not a natural one for access. Tables can then be accessed by this index.

7.1.13 Task

Task will provide a way to further select the type of data retrieved. For example:-

- A tracking algorithm might have two Algorithm Configurations, one for stopping tracks and another for high energy ones.

- Detector Configuration data could have two tasks, one for raw calibration and another for refined calibration.

The aim is that Task will allow a particular database table to be subdivided according to the mode of use. Currently Task is a data type defined in Dbi i.e. `Dbi::Task` and is implemented as an integer. The default value is zero.

7.1.14 Level 2 (disk) Cache

Loading a large table from the database is a lot of work:-

1. The query has to be applied and the raw data loaded.
2. The row objects have to be individually allocated on the heap.
3. Each data word of each row object has to be individually converted through several layers of the support database software from the raw data.

Now as the detector configuration changes slowly with time identically the same process outlined above is repeated many times, in many jobs that process the data, so the obvious solution is to cache the results to disk in some way that can be reloaded rapidly when required. The technique essentially involves making an image copy of the table to disk. It can only be applied to some tables, but these include the Calibration tables which represent the largest database I/O load, and for these tables loading times can be reduced by an order of magnitude.

7.2 Installing and Running

7.2.1 Installing

The DatabaseInterface relies on RDBC, another package in the framework, which in turn relies on external libraries:-

- unixODBC. This provides back end drivers for the real database such as MySQL and ORACLE. It is used by libodbc++.
- MyODBC. This provides an ODBC interface to MySQL, the back-end we currently use at sites throughout the collaboration.

If these libraries are not installed than you will need to do this.

7.2.2 Setting up the Environment

The interface needs a list of Database URLs, a user name and a password. These can be communicated via a set environmental variables of the form ENV_TSQL*. Here is the set with some example setting:-

```
setenv ENV_TSQL_URL "mysql:odbc://pplx2.physics.ox.ac.uk/offline"
setenv ENV_TSQL_USER "west"
setenv ENV_TSQL_PSWD ""
```

The meanings are as follows.

ENV_TSQL_URL This is a semi-colon separated list of URLs. Each URL takes the form:-

```
protocol:[subprotocol:][driver:]//host[:port]/[database][?options]
where:
   protocol - DBMS type , e.g. mysql,oracle, etc.
   subprotocol - driver type , currently: "odbc"
   driver - driver name it could be
                 alias name defined in /etc/odbcinst )
              or path to the driver lib,
                 e.g.myodbc - /usr/local/lib/libmyodbc.so
             If absolute pathname is not specified, driver manager
             will try to find "lib[driver].so" in the
             following directories
                /usr/lib,
                /usr/local/lib,
                $ROOTSYS/lib, %$
   host - host name or IP address of database server
   port - port number
   database - name of database
   options - string key=value's separated by ';' or '&'
 Example:
 "mysql:odbc:myodbc://myhost:3306/test?Trace=Yes;TraceFile=qq.log"
```

ENV_TSQL_USER Your database user name. If you require different names for different databases in the cascade then this can be a semi-colon separated list in the same order as ENV_TSQL_URL. If the list is shorter than that list, then the first entry is used for the missing entries.

ENV_TSQL_PSWD You could put your password here, but it is a security risk to leave that set up in the environment. Leaving it null will mean that it will be prompted for when the interface initialises. As with ENV_TSQL_USER it can be a semi-colon separated list with the first entry providing the default if the list is shorter than ENV_TSQL_URL.

These variable should be set for the standard read-only configuration. Any job that uses the JobControl package can override any or all of these variables using the switches:-

- -d: Supply database URL (replaces ENV_TSQL_URL)
- -u: Supply database user (replaces ENV_TSQL_USER)
- -p: Supply database password (replaces ENV_TSQL_PSWD)

for example:-

```
demojob -d mysql://pplx2.physics.ox.ac.uk/offline -u reader ...
```

replaces ENV_TSQL_URL and ENV_TSQL_USER but takes the password from ENV_TSQL_PSWD. Use of these switches is the recommended way to configure a framework job when write-access to the database is required.

7.2.3 Configuring

Once the job is running you can configure the DatabaseInterface. To change any configurable aspect of the interface get a reference to its `CfgConfigurable` interface, set parameters as required and update:-

```
// DbiTableProxyRegistry is a CfgConfigurable
DbiTableProxyRegistry& dbiCfg = DbiTableProxyRegistry::Instance();
dbiCfg.Set("...");
dbiCfg.Set("...");
dbiCfg.Set("...");
dbiCfg.Update();
```

On startup the interface checks the environmental variable `ENV_DBI` which can contain a semi-colon separated list of configuration requests. For example:-

```
setenv ENV_DBI "Level2Cache = '/home/minos/temp/dbi_chache';Rollback:* = '2000-01-01'"
```

see below for an explanation of what these requests mean.

In the above example the environment string contained a *. To prevent csh and derivative shells (e.g.tcsh) for attempting to expand the * as a set of file names, temporarily define the shell variable nogob:-

```
set noglob
setenv ENV_DBI " ...
unset noglob
```

Rollback

To impose a global rollback date to say September 27th 2002:-

```
dbiCfg.Set("Rollback:* = '2002-09-27 00:00:00'");
```

This will ensure that the interface ignores data inserted after this date for all future queries. The hours, minutes and seconds can be omitted and default to 00:00:00.

Rollback can be more selective, specifying either a single table or a group of tables with a common prefix. For example:-

```
dbiCfg.Set("Rollback:*            =  '2002-09-01'");
dbiCfg.Set("Rollback:CAL*         =  '2002-08-01'");
dbiCfg.Set("Rollback:CALPINTOADC  =  '2002-07-01'");
```

Now the table CALPINTOADC is frozen at July 2002, other CAL tables at August and all other tables at September. The ordering of the commands is not important; the interface always picks the most specific one to apply to each table.

> *Rollback only applies to future queries, it does not invalidate any existing query result in the cache which are still available to satisfy future requests. So impose rollback conditions at the start of the program to ensure they apply consistently.*

SimFlag Association

By default the DatabaseInterface matches data using the context of the current event, and this includes the type of data e.g. Monte Carlo or real. As explained in the Concepts section 7.1.7 this can be overridden. For example:-

```
dbiCfg.Set("SimFlagAssociation:MC = 'Reroot,MC,Data'");
```

would mean that for MC data, the interface would try in order: Reroot, then MC and finally Data for the first database cascade member and then the second and so on until a match was found.

Multiple associations can be set up, here is a really silly example:-

```
dbiCfg.Set("SimFlagAssociation:MC = 'Data'");
dbiCfg.Set("SimFlagAssociation:Data = 'MC'");
```

Shutdown

By default the DatabaseInterface does not clean up its cache at the end of the program; for to do so would waste time and gain nothing. However, if looking for leaks this then appears as a massive one, so a full shutdown with cache clean up can be enabled by:-

```
dbiCfg.Set("Shutdown = 1");
```

Level 2 Cache

Enabling the Level 2 Cache allows certain large tables query results to be written to disk from which they can be reloaded by subsequent jobs saving as much as an order of magnitude in load time. Data in the cache will not prevent changes in the database from taking affect for the DatabaseInterface does an initial (lightweight) query of the database to confirm that the data in the cache is not stale. To enable the cache, the user specifies a directory to which they have read/write access. For example, to make the current working directory the cache:-

```
dbiCfg.Set("Level2Cache = './'");
```

Cache files all have the extension .dbi_cache. Not all tables are suitable for Level 2 caching; the DatabaseInterface will only cache the ones that are.

Cache files can be shared between users at a site to maximise the benefit. In this case the local Database Manager must set up a directory to which the group has read/write access. Management is trivial, should the cache become too large, it can simply be erased and then the next few jobs that run will re-populate it with the currently hot queries.

Note that Cache performance is achieved by doing raw binary I/O so the cache files are platform specific, so if running in a heterogeneous cluster the Database Manager should designate a platform specific directory. To simplify this, the name of the directory used by the cache can include environmental variables e.g.:-

```
dbiCfg.Set("Level2Cache = '$DBI_L2CACHE/$SRT_ARCH'");
```

7.3 Accessing Existing Tables

7.3.1 Introduction

The DatabaseInterface gives you access to two types of data:-

Detector Description i.e. data that describes the construction of the detector and how it responds to the passage of particles through it. The geometry, the cabling plex and calibration constants are all examples of this type of data.

Algorithm Configuration i.e. data used to control the way an algorithm operates. This includes such things as software switches to control flow and tunable cuts.

In all cases the user specifies the database table to be accessed and supplies a "context" for the query. The context describes the type and date time of the current event. This is stored in a Validity package `VlsContext` object. Candidate and RawData have:-

```
VldContext   GetVldContext() const
```

methods to get their context. The DatabaseInterface uses the context to extract all the rows from the database table that are valid for this event. It forms the result into a table in memory and returns a object that acts like a pointer to it.

> *You are NOT responsible for deleting the table; the Database Interface will do that when the table is no longer needed*

You have random access to any row of the results table. Each row is an object which is specific to that table. The key to understanding how to get data from a database table is study the class that represent a row of it results table.

7.3.2 Accessing Detector Descriptions

Making the Query

As explained above, the key to getting data is to locate the class that represents one row in a database table. At the time of writing this chapter, there are no real-life database tables so you are going to have to settle for one of the sample tables included in the test directory of the DatabaseInterface package and imaginatively called `DbiDemoData1, DbiDemoData2` and `DbiDemodata3`. For purposes of illustration we will pick the first of these. Its header can be found in:-

```
DatabaseInterface/test/DbiDemoData1.h
```

To make a query you create a `DbiResultPtr` object. Its header can be found in:-

```
DatabaseInterface/DbiResultPtr.h
```

This is a class that is templated on the table row class, so in this case the instantiated class is:-

```
DbiResultPtr<DbiDemoData1>
```

and to instantiate an object of this class you just need a `VldContext` object. Suppose vc is such an object, then this creates the pointer:-

```
DbiResultPtr<DbiDemoData1> myResPtr(vc);
```

This statement creates a DbiResultPtr for DbiDemoData1 class. First it searches through the database for all DbiDemoData1 objects that are valid for vc, then it assembles them into a table and finally passes back a pointer to it. Not bad for one statement! The constructor can take a second argument:-

```
DbiResultPtr(VldContext vc,Dbi::Task task=0);
```

Dbi::Task offers a way to sub-divided a table according to the mode of operation. For example:-

- A tracking algorithm might have two Algorithm Configurations, one for stopping tracks and another for high energy ones.
- Detector Configuration data could have two modes, one for raw calibration and another for refined calibration.

If the concept of a task is not relevant for a particular database table, then it should be left at its default value of 0. Otherwise data for each task should be allocated a unique positive number and then selection will only pick rows with the required value of task.

Accessing the Results Table

Having got a pointer to the table the first thing you will want to know is how many rows it has. Do this using the method:-

```
UInt_t GetNumRows() const;
```

If the query failed then the number of rows returned will be zero. This could either be the result of some catastrophic failure, for example the database could not be opened, or simply that no appropriate data exists for the current event. If you want to know which of these it is you can use the:-

```
const DbiValidityRec* GetValidityRec() const;
```

If this returns a null pointer, then the failure was a major one. If not then the `DbiValidityRec` tells you about the validity of the gap. Its method:-

```
const VldRange& GetVldRange() const;
```

returns a Validity package `VldRange` object that can yield the start and end times of the gap. Due to the way the DatabaseInterface forms the query, this may be an underestimate, but never an overestimate.

If the table has rows then the GetVldRange() will give you an object that tells you the validity range of the data. Again, the range may be an underestimate. To get to the data itself, use the method:-

```
const T* GetRow(UInt_t i) const;
```

where T = DbiDemoData1 in this case. This gives you a const pointer to the i^{th} row where i is in the range $0 < i <=$ GetNumRows().

Putting this all together, suppose you have a `CandDigitListHandle` object cdlh, and you want to loop over all `DbiDemoData1` objects that are valid for it, the code is:-

```
DatabaseInterface/test/DbiDemoData1.h
DatabaseInterface/DbiResultPtr.h

...

DbiResultPtr<DbiDemoData1> myResPtr(cdlh.GetVldContext());

for ( UInt_t irow = 0; irow < myResPtr.GetNumRows(); ++ires) {
  const DbiDemoData1* ddd1 = myResPtr.GetRow(irow);

// Process row.

}
```

GetRow is guaranteed to return a non-zero pointer if the row number is within range, otherwise it returns zero. The ordering of rows reflects the way the data was written to the database. For some types of data this layout is not well suited for access. For example, for pulser data, all the strip ends illuminated by an LED will appear together in the table. To deal with this table row object are obliged to return a Natural Table Index, if the physical ordering is not a natural one for access. You get rows from a table according to their index using the method:-

```
const T* GetRowByIndex(UInt_t index) const;
```

You should always check the return to ensure that its non-zero when using this method unless you are absolutely certain that the entry must be present.

Getting Data from a Row

Having got to the table row you want, the last job is to get its data. Its up to the table row objects themselves to determine how they will present the database table row they represent. In our example, the `DbiDemoData1` is particularly dumb. Its internal state is:-

```
 Int_t    fSubSystem;
 Float_t  fPedestal;
 Float_t  fGain1;
 Float_t  fGain2;
```

which it is content to expose fully:-

```
   Int_t GetSubSystem() const { return fSubSystem; }
 Float_t GetPedestal() const { return fPedestal; }
 Float_t GetGain1() const { return  fGain1; }
 Float_t GetGain2() const { return fGain2; }
```

Its worth pointing out though that it is the job of the table row object to hide the physical layout of the database table and so shield its clients from changes to the underlying database. Its just another example of data encapsulation.

Making Further Queries

Even though a `DbiResultPtr` is lightweight it is also reusable; you can make a fresh query using the `NewQuery` method:-

```
UInt_t NewQuery(VldContext vc, Dbi::Task task=0);
```

which returns the number of rows found in the new query. For example:-

```
DbiResultPtr<DbiDemoData1> myResPtr(vc);
...
VldContext newVc;
...
myResPtr.NewQuery(newVc);
...
```

Having made a query you can also step forwards or backwards to the adjacent validity range using the method:-

```
UInt_t NextQuery(Bool_t forwards = kTRUE);
```

supply a false value to step backwards. This method can be used to "scan" through a database table, for example to study calibration constants changes as a function of time. **Caution:** The interface can create artificial validity range boundaries to improve performance so stepping to a neighbouring range may give access to the same data and further steps may be required to get to the next set. For aggregated data stepping to a neighbouring range will almost certainly contain some rows in common unless all component aggregates have validity ranges that end on the boundary you are crossing.

Simple Optimisation

The first, and most important, level of optimisation is done within the DatabaseInterface itself. Each time it retrieves data from the database it places in an internal cache. This is then checked during subsequent queries and reused as appropriate. So the first request for a large table of calibration constants may require a lot of I/O. However the constants may remain valid for an entire job and in which case there is no further I/O for this table.

Although satisfying repeat requests for the same data is quick it still requires the location of the appropriate cache and then a search through it looking for a result that it is suitable for the current event. There are situations when even this overhead can be a burden: when processing many rows in a single event. Take for example the procedure of applying calibration. Here every digitization needs to be calibrated using its corresponding row in the database. The naive way to do this would be to loop over the digits, instantiating a `DbiResultPtr` for each, extracting the appropriate row and applying the calibration. However it would be far more efficient to create a little calibration object something like this:-

```
class Calibrator {
public:
  Calibrator(const VldContext vc): fResPtr(vc) {}
  Float_t Calibrate(const CandDigitHandle cdh) {
     /*  Use fResPtr to calibrate cdh  */
}

private
  DbiResultPtr<DbiDemoData1> fResPtr;
};
```

`Calibrator` is a lightweight object holding only a pointer to a results table. It is created with a `VldContex` object which it uses to prime its pointer. After that it can be passed `CandDigitHandle` objects for which it returns calibrated results using its Calibrate method. Now the loop over all digitizations can use this object without any calls to the DatabaseInterface at all. Being lightweight `Calibrator` is fine as a stack object, staying in scope just long enough to do its job.

7.3.3 Accessing Algorithm Configuration

In all probability you will not need to know much about the DatabaseInterface when it comes to Algorithm Configuration; all such data ends up inside `AlgConfig` objects. A helper class `DbiConfigStream` provides a simple way to fill and store `Registry` objects, and as `AlgConfig` objects inherit from `Registry` it can fill and store them too. All configuration data is tagged with a pair of names:-

Software Name This is the name of the software (normally an Algorithm) that is being configured.

Configuration Name Each configurable software system may have multiple configuration sets each with it's own name. If there is only a single set then the name will be "default".

To create a `Registry` object and fill it from the set "MyConfig" for the software "MyAlgorithm":-

```
Registry reg;
DbiConfigstream stream("MyAlgorithm","MyConfig");
stream >> reg;
```

`DbiConfigStream` is a lightweight object that can be discarded as soon as the `Registry` object has been filled.

You can stop reading this section now unless you are curious as to what is happening inside `DbiConfigStream`. For Detector Description data, every database table has its own table row class. With Algorithm Configuration, the reverse is true. All data is loaded from a single table (SOFTWARE_CONFIGURATION) using the same table row class: `DbiConfigSet`, and a `DbiConfigStream` object provides a simple way to read and write these objects. Also, in contrast to Detector Configuration data, Algorithm Configuration queries yield a simple (i.e. single row) result. The single `DbiConfigSet` object give access to all of its parameters i.e. columns:-

```
      UInt_t GetNumParams() const;
      string GetParamName(UInt_t parNo) const;
DbiFieldType GetParamType(UInt_t parNo) const;
      string GetParamValue(UInt_t parNo) const;
```

Each parameter ends up as an item in the `Registry`.

By default `DbiConfigSet` is loaded from the SOFTWARE_CONFIGURATION table which has the following columns:-

SEQNO The sequence number.

SOFTW_NAME The software name.

CONFIG_NAME The configuration set name.

CONFIG_DATA The configuration data as a single string as produced by Registry::PrintStream(). For example:-

```
['' 'mydouble'=(double)1.234e+100 'myint'=(int)12345 'mystring'=(string)'This is a string']
```

In the above case the Registry would be filled with 3 values:-

```
mydouble = 1.234e+100
myint = 12345
mystring = This is a string
```

Normally Algorithm Configuration data will be universally valid and only use the default task so the constructor takes a series of default arguments, but the full set is:-

```
DbiConfigStream(const std::string& SoftwName,
                const std::string& ConfigName="default",
                VldContext vc = Dbi::fgDefaultContext,
                Dbi::Task task=0,
                const std::string& tableName="SOFTWARE_CONFIGURATION");
```

So it is possible to specify a `VldContext` and a task and even an alternative table name.

7.3.4 Writing Algorithm Configuration

Although database writing is dealt with later in this chapter, this is a convenient place to deal with configuration output. To create new configuration data, or overlay existing ones:-

1. First create a `DbiConfigStream` object as described above e.g.:-

   ```
   DbiConfigstream stream("MyAlgorithm","MyNewConfig");
   ```

2. Next fill it from a `Registry`:-

   ```
   Registry reg;
   // Fill reg from somewhere...
   stream << reg;
   ```

   At this stage only the `DbiConfigStream` has been updated; no database output has been performed.

3. By default the data will be universally valid but you can adjust that by:-

   ```
   DbiValidityRec& vrec = stream.GetValidityRec();
   ```

   and modifying/replacing as required.

4. Finally write out the new data using:-

   ```
   Bool_t DbiConfigStream::Write(UInt_t dbNo = 0,
                const std::string& logComment = "");
   ```

   specifying the cascade entry number and the comment used to create a new database log record.

7.3.5 Extended Context Queries

Making the Query

The constructor of a `DbiResultPtr` for an Extended Context Query is:-

```
DbiResultPtr(const string& tableName,
             const DbiSqlContext& context,
             const Dbi::Task& task = Dbi::kAnyTask,
             const string& data = "",
             const string& fillOpts = "",
```

Dealing with each of these arguments in turn:-

const string& tableName The name of the table that is to be accessed. This allows any type of `DbiTableRow` to be loaded from any type of table, but see section 7.5 on filling if you are going to play tricks!

const DbiSqlContext& context This argument provides the extended context through the utility class `DbiSqlContext`. Consider the following code:-

```
// Construct the extended context: FarDet data that starts on Sept 1 2003.
VldTimeStamp tsStart(2003,9,1,0,0,0);
VldTimeStamp   tsEnd(2003,9,1,23,59,59);
DbiSqlContext context(DbiSqlContext::kStarts,tsStart,tsEnd,DetectorType::kFar,SimFlag::kData);
```

You supply the type of context (in this case `DbiSqlContext::kStarts`), the date range and the detector type and sim flag. Other types of context are kEnds and kThroughout. See

```
DatabaseInterface/DbiSqlContext.h
```

for the complete list.

You are not limited to the contexts that `DbiSqlContext` provides. If you know the SQL string you want to apply then you can create a DbiSqlContext with the WHERE clause you require e.g.:-

```
DbiSqlContext myContext("DETECTORMASK & 4")
```

which would access every row that is suitable for the CalDet detector.

const Dbi::Task& task The task is as for other queries but with the default value of:-

```
Dbi::kAnyTask
```

which results in the task being omitted from the context query.

const string& data This is an SQL fragment, that if not empty (the default value) is used to extend the WHERE clause that is applied when querying the main table. For example consider:-

```
DbiSqlContext context(DbiSqlContext::kStarts,tsStart,tsEnd,DetectorType::kFar,SimFlag::kData);
DbiResultPtr<DbuSubRunSummary> runs("DBUSUBRUNSUMMARY",context,Dbi::kAnyTask,"RUNTYPENAME = 'NormalData'");
```

This query reads the DBUSUBRUNSUMMARY table, and besides imposing the context query also demands that the data rows satisfies a constraint on RUNTYPENAME.

const string& fillOpts This is a string that can be retrieved from `DbiResultSet` when filling each row so could be used to program the way an object fills itself e.g. by only filling certain columns. The DatabaseInterface plays no part here; it merely provides this way to communicate between the query maker and the the author of the class that is being filled.

Accessing the Results Table

Accessing the results of an Extended Context query are essentially the same as for a standard query but with following caveats:-

- If the method:-

  ```
  const DbiValidityRec* GetValidityRec(const DbiTableRow* row=0) const;
  ```

  is used with the default argument then the "global validity" of the set i.e. the overlap of all the rows is returned. Given the nature of Extended Queries there may be no overlap at all. In general it is far better to call this method and pass a pointer to a specific row for in this case you will get that validity of that particular row.

- The method:-

  ```
  const T* GetRowByIndex(UInt_t index) const;
  ```

  will not be able to access all the data in the table if two or more rows have the same Natural Index. This is prohibited in a standard query but extended ones break all the rules and have to pay a price!

7.3.6 Error Handling

All `DbiResultPtr` constructors, except the default constructor, have a last optional argument:-

```
Dbi::AbortTest abortTest = Dbi::kTableMissing
```

Left at its default value any query that attempts to access a non-existent table will abort the job. The other values that can be supplied are:-

kDisabled Never abort. This value is used for the default constructor.

kDataMissing Abort if the query returns no data. Use this option with care and only if further processing is impossible.

Currently aborting means just that; there is no graceful shut down and saving of existing results. You have been warned!

7.4 Creating New Tables

7.4.1 Choosing Table Names

The general rule is that a table name should match the `DbiTableRow` subclass object that it is used to fill. For example the table PlexRawChannelToPinDiode corresponds to the class `PLEXRAWCHANNELTOPINDIODE`. Beyond this the following are recommended as we need to remain compatible with ORACLE:-

- Use upper case letters.
- A limit of 30 characters, including VLD.
- Avoid common names. VIEW and MODE are used by ORACLE. A good list of names to avoid can be found at:-

  http://home.fnal.gov/%7Edbox/SQL_API_Portability.html

These restrictions also apply to column names.

7.4.2 Creating Detector Descriptions

A Simple Example

Creating new Detector Descriptions involves the creation of a database table and the corresponding table row Class. The main features can be illustrated using the example we have already studied: `DbiDemoData1`. Recall that its state data is:-

```
     Int_t   fSubSystem;
    Float_t  fPedestal;
    Float_t  fGain1;
    Float_t  fGain2;
```

Its database table, which bears the same name, is defined, in MySQL, as:-

```
create table DbiDemoData1(
    SeqNo      int,
    SubSystem  int,
    Pedestal   float,
    Gain1      float,
    Gain2      float
 );
```

as you can see there is a simple 1:1 correspondence between them except that the database table has an additional leading entry:-

```
    SeqNo      int,
```

All database tables supported by the DatabaseInterface have an auxiliary Validity Range Tables that defines validity ranges for them. Each validity range is given a unique sequence number that acts as a key and corresponds to `SeqNo`. So, when the DatabaseInterface looks for data that is acceptable for a give validity it:-

1. Matches the validity to an entry in the appropriate Validity Range Table and gets its SeqNo.

2. Uses SeqNo as a key into the main table to get all the rows that match that key.

So, as a designer, you need to be aware of the sequence number, it *must* be the first column in the database table, but is not reflected in the table row class.

Filling a table row object from the database is done using the class's Fill method. For our example:-

```
void DbiDemoData1::Fill(DbiResultSet& rs,
                        const DbiValidityRec* vrec) {

  rs >> fSubSystem >> fPedestal >> fGain1 >> fGain2;

}
```

the table row object is passed a `DbiResultSet` which acts rather like an input stream. The sequence number has already been stripped off; the class just has to fill its own data memeber. The DatabaseInterface does type checking (see the next section) but does not fail if there is a conflict; it just produces a warning message and puts default data into the variable to be filled.

The second argument is a `DbiValidityRec` which can, if required, be interrogated to find out the validity of the row. For example:-

```
  const VldRange& range = vrec->GetVldRange();
```

vrec may be zero, but only when filling `DbiValidityRec` objects themselves. On all other occasions vrec should be set.

Creating a Database Table

The previous section gave a simple MySQL example of how a database table is defined. There is a bit more about MySql in section 7.7. The table name normally must match the name of the table row class that it corresponds to. There is a strict mapping between MySQL data types and table row data members, although a few cases where one MySQL type can be used to load more than one type of data member. The mapping is given in table 7.1

Unsigned data types are not supported in the underlying RDBC library, but the interface does with the following caveats:-

1. Int_t must be declared in the database as BIGINT if the top bit could be set i.e. the integer exceeds 2**31-1 = 2147483647.

MySQL Type	Table Row Type	Comments
CHAR or CHAR(n) n<4	Bool_t	
CHAR or CHAR(n) n<4	Char_t	
CHAR(n) e.g. CHAR(7)	string	
TINYTEXT	string	
TINYINT	Char_t	
SMALLINT	Short_t	
INT or INTEGER	Int_t	
BIGINT	Int_t	See below
FLOAT	Float_t	
DOUBLE	Double_t	
TEXT	string	
DATETIME	VldTimeStamp	

Table 7.1: Valid MySQL and table row type combinations

2. unsigned integers are allowed as both MySQL types (add UNSIGNED after the type) and Table Row Types and are considered to be compatible to their signed. equivalents.

Not all the obvious ROOT types are supported, e.g. TString, but adding new mappings is straight forward and will be introduced as required. The long integer forms are not supported as on Intel processors they are only 4 bytes long.

Designing a Table Row Class

Here is a list of the requirements for a table row class.

Must inherit from DbiTableRow All table row objects must publicly inherit from the abstract class`DbiTableRow`. `DbiTableRow` does provide some default methods even though it is abstract.

Must provide a public default constructor e.g.:-

```
DbiDemoData1::DbiDemoData1() { }
```

The DatabaseInterface needs to keep a object of every type of table row class.

Must implement CreateTableRow method e.g.:-

```
virtual DbiTableRow* CreateTableRow() const {
                             return new DbiDemoData1; }
```

The DatabaseInterface uses this method to populate results tables.

May overload the GetIndex method As explained in section 7.3.2 the ordering of rows in a table is determined by the way data is written to the database. Where that does not form a natural way to access it, table row objects can declare their own index using:-

```
UInt_t GetIndex(UInt_t defIndex) const
```

`DbiDemoData2` provides a rather artificial example:-

```
UInt_t GetIndex(UInt_t defIndex) const { return fSubSystem/10; }
```

and is just meant to demonstrate how a unique index could be extracted from some packed identification word.

The following is required of an index:-

- The number must be unique within the set.
- It must fit within 4 bytes.

GetIndex returns an unsigned integer as the sign bit has no special significance, but its O.K. to derive the index from a signed value, for example:-

```
Int_t PlexStripEndId::GetEncoded() const
```

would be a suitable index for tables indexed by strip end.

Must implement Fill method This is the way table row objects get filled from a `DbiResultSet` that acts like an input stream. We have seen a simple example in `DbiDemoData1`:-

```
void DbiDemoData1::Fill(DbiResultSet& rs,
                        const DbiValidityRec* vrec) {

  rs >> fSubSystem >> fPedestal >> fGain1 >> fGain2;

}
```

However, filling can be more sophisticated. `DbiResultSet` provides the following services:-

```
      string DbiResultSet::CurColName() const;
      UInt_t DbiResultSet::CurColNum() const;
      UInt_t DbiResultSet::NumCols() const;
DbiFieldType DbiResultSet::CurColFieldType() const;
```

The first 3 give you the name of the current column, its number (numbering starts at one), and the total number of columns in the row. `DbiFieldType` can give you information about the type, concept and size of the data in this column. In particular you can see if two are compatible i.e. of the same type:-

```
Bool_t DbiFieldType::IsCompatible(DbiFieldType& other) const;
```

and if they are of the same capacity i.e. size:-

```
Bool_t DbiFieldType::IsSmaller(DbiFieldType& other) const;
```

You can create `DbiFieldType` objects e.g:-

```
DbiFieldType myFldType{Dbi::kInt)
```

see enum Dbi::DataTypes for a full list, to compare with the one obtained from the current row.

In this way filling can be controlled by the names, numbers and types of the columns. The Fill method of `DbiDemoData1` contains both a "dumb" (take the data as it comes) and a "smart" (look at the column name) code. Here is the latter:-

```
Int_t numCol = rs.NumCols();

//  The first column (SeqNo) has already been processed.
for (Int_t curCol = 2; curCol <= numCol; ++curCol) {
  string colName = rs.CurColName();
  if (      colName == "SubSystem" ) rs >> fSubSystem;
  else if ( colName == "Pedestal" )  rs >> fPedestal;
  else if ( colName == "Gain1" )     rs >> fGain1;
  else if ( colName == "Gain2" )     rs >> fGain2;
  else {
    MSG("Dbi",Msg::kDebug) << "Ignoring column " << curCol
                           << "(" << colName << ")"
             << "; not part of DbiDemoData1" << endl;
    rs.IncrementCurCol();
  }
}
```

> *Being "smart" comes at a price; if your table has many rows valid at at time, defensive programming like this can cost performance!*

In such cases, and if the table only exists is a few variants, its better to determine the variant and then branch to code that hardwires that form

Other services that `DbiResultSet` offers are:-

```
UInt_t DbiResultSet::CurRowNum() const;
Bool_t DbiResultSet::IsExhausted() const;
string DbiResultSet::TableName();
```

These tell you the current row number, whether there is no data left and the name of the table.

Also note that it is not a rule that database columns and class data members have to be in a 1:1 correspondence. So long as the table row can satisfy its clients (see below) it can store information derived from the database table rather than the data itself.

May overload the CanL2Cache method As explained in section 7.1 the Level 2 cache allows table loading to be speeded up by caching the query results as disk files. Only certain tables support this option which by default is disabled. To enable it the table row object overrides this method as follows:-

```
Bool_t CanL2Cache() const { return kTRUE; }
```

Only table row classes who data members are built-in data types (ints, floats and chars) should do this. Table rows having objects or dynamic data e.g. string or pointers must not claim to support L2 caching. Note the table row doesn't need code to save/restore to the cache, this is handled by the `DbiTableProxy`

Must Provide Services to its Clients There would not be much point in its existence otherwise would there? However its not necessarily the case that all its does is to provide direct access to all the data that came from the table. This subject is explored in the next section.

The LinkDef.h file

For each Table Row class you must add an entry to the package's `LinkDef.h` file so that rootcint can supply all the additional methods such as Streamer to integrate the class into the ROOT framework. For example DatabaseInterface/test/LinkDef.h are the lines:-

```
#pragma link C++ class DbiResultPtr<DbiDemoData1>;
#pragma link C++ class DbiResultPtr<DbiDemoData2>;
#pragma link C++ class DbiResultPtr<DbiDemoData3>;
```

Caution: if you have class Yyy that has a `DbiResultptr<Xxx>` *data member. and Yxx and Xxx are in the same package the line:-*

```
#pragma link C++ class DbiResultPtr<Xxx>;
```

must precede:-

```
#pragma link C++ class Yyy;
```

The GNUmakefile file

The additions to the LinkDef.h file tells rootcint what new classes you have introduced to access the database, but you also have to ensure that the make system supplies rootcint with the DatabaseInterface template class headers. This is actually a generic problem that occurs whenever one package uses template headers from another. The solution is to add the required header to the CINTLIST and to use the make vpath command to tell make where to find it. For example, to add DbiResultPtr.h:-

```
ROOTCINT := YES
vpath DatabaseInterface/DbiResultPtr.h $(SRT_PRIVATE_CONTEXT)/include/ \
                                       $(SRT_PUBLIC_CONTEXT)/include/
CINTLIST += DatabaseInterface/DbiResultPtr.h
```

The first line is as normal and switches on rootcint, supplying it with all the packages headers, which are stored in the variable CINTLIST. The next line tells make to look for DatabaseInterface/DbiResultPtr.h first the SRT private and then public contexts. That ensures that searching is in the correct order; private supersedes public. The last line adds the header to the list of headers passed to rootcint.

Data Encapsulation

A table row object is the gateway between a database table and the end users who want to use the data it contains. Like any good OO design, the aim should be to hide implementation and only expose the abstraction. There is nothing wrong in effectively giving a 1:1 mapping between the columns of the database table and the getters in the table row object if that is appropriate. For example, a table that gives the position of each plane in a detector is going to have an X, Y and Z both in the database and in the getter. However at the other extreme there is calibration. Its going to be well into detector operation before the best form of calibration has been found, but it would be bad design to constantly change the table row getters. Its far better to keep the data in the database table very generic, for example:-

```
    SeqNo      int,
    SubSystem  int,
    CalibForm  int,
    parm0      float,
    parm1      float,
    parm2      float,
    ...
```

The significance of parm0,... depends on CalibForm. The table row object could then provide a calibration service:-

```
Float_t Calibrate(Float_t rawValue) const;
```

rather than expose parm0,.. Calibrate() would have code that tests the value of CalibForm and then uses the appropriate formula involving parm0... Of course some validation code will want to look at the quality of the calibration by looking at the calibration constants themselves, but this too could be abstracted into a set of values that hide the details of the form of the calibration.

7.5 Filling Tables

7.5.1 Overview

DatabaseInterface can be used to write back into any table from which it can read. To do this you need the services of a `DbiWriter`. In the case of Algorithm Configuration there is a convenient interface using the `DbiConfigStream` object - see section 7.3.4.

For all other cases you will need to explicitly use a `DbiWriter` which is a templated class like `DbiResultPtr`, so for example, to write `DbiDemoData1` rows you need an object of the class:-

```
DbiWriter<DbiDemoData1>
```

In outline the procedure is as follows:-

1. Decide the validity range of the data to be written and store it in a `VldRange` object.
2. Instantiate a `DbiWriter` object using this `VldRange` object together with an aggregate number and task. Aggregate numbers are discussed below.
3. Pass filled `DbiTableRow` sub-class objects (e.g. `DbiDemoData1`) to the `DbiWriter`. It in turn will send these objects their Store message that performs the inverse of the Fill message. DbiWriter caches the data but performs no database I/O at this stage.
4. Finally send the `DbiWriter` its Close message which triggers the output of the data to the database.

The fact that I/O does not occur until all data has been collected has a couple of consequences:-

- It minimises the chances of writing bad data. If you discover a problem with the data while `DbiWriter` is assembling it you use `DbiWriter`'s Abort method to cancel the I/O. Likewise if `DbiWriter` detects an error it will not perform output when Close is invoked. Destroying a `DbiWriter` before using Close also aborts the output.

- Although `DbiWriter` starts life as very lightweight, it grows as the table rows are cached.

  > *Be very sure that you delete the* `DbiWriter` *once you have finished with it or you will have a serious memory leak!*

  To cut down the risk of a memory leak, you cannot copy construct or assign to `DbiWriter` objects.

7.5.2 Aggregate Numbers

As explained in Concepts (see section 7.1) some types of data are written for the entire detector as a single logical block. For example the way PMT pixels map to electronics channels might be written this way. On the other hand if it is written in smaller, sub-detector, chunks then it is Aggregated. For example light injection constants come from pulser data and it is quite possible that a calibration run will only pulse some LEDs and so only part of a full detector set of constants gets written to the database for the run. Each chunk is called an aggregate and given an aggregate number which defines the sub-section of the detector it represents. For pulser data, the aggregate number will probably be the logical (positional) LED number A single `DbiWriter` can only write a single aggregate at a time, for every aggregate can in principle have a different validity range. For unaggregated data, the aggregate number is -1, for aggregated data numbers start at 0,1,2...

The way that the DatabaseInterface assembles all valid data for a given context is as follows:-

- First if finds all aggregate records that are currently valid.

- For each aggregate number it finds the best (most recently created) record and loads all data associated with it.

This has two consequences:-

- For a given table, the regime whereby the data is organised into aggregates should remain constant throughout all records in the table. If absolutely necessary the regime can be changed, but no records must have validities that span the boundary between one regime and another. Were that to be the case the same entry could appear in two valid records with different aggregates numbers and end up appearing in the table multiple times. The system checks to see that this does not happen.

- For any given context it is not necessary for all detector elements to be present; just the ones that are really in the detector at that time. For example, the Far detector will grow steadily over more than a year and this will be reflected in some database tables with the number of valid aggregates similarly growing with time. What aggregates are present can appear in any order in the database tables, the interface will assemble them into the proper order as it loads them.

Its perfectly possible that a calibration procedure might produce database data for multiple aggregates at a single pass. If you are faced with this situation and want to write all aggregates in parallel, then simply have a vector of `DbiWriter`'s indexed by aggregate number and pass rows to the appropriate one. See `DbiValidate::Test_6()` for an example of this type of parallel processing.

7.5.3 Simple Example

We will use the class `DbiDemoData1` to illustrate each of the above steps.

1. Set up `VldRange` object.
   Typically the `VldRange` will be based on the `VldContext` for the event data that was used to generate the database data that is to be stored. For our example we will assume that `DbiDemoData1` represents calibration data derived from event data. It will be valid for 1 week from the date of the current event and be suitable for the same type of data.

```
VldContext now;  // Event context e.g. CandHandle::GetVldContext()
VldTimeStamp start = now.GetTimeStamp();
// Add 7 days (in secs) to get end date.
```

```
time_t vcSec = start.GetSec() + 7*24*60*60;
VldTimeStamp    end(vcSec,0);
// Construct the VldRange.
VldRange      range(now.GetDetector(),
                    now.GetSimFlag(),
                    start,
                    end,
                    "Demo");
```

2. Instantiate a `DbiWriter`.
   Create a `DbiDemoData1` writer for unaggregated data task 0.

```
UInt_t aggNo = -1;
Dbi::Task task = 0;
// Decide a creation date (default value is now)
VldTimeStamp create;
DbiWriter<DbiDemoData1> writer(range,aggNo,task,create);
```

3. Pass filled `DbiDemoData1` objects.

```
// Create some silly data.
DbiDemoData1 row0(0,10.,20.,30.);
DbiDemoData1 row1(0,11.,21.,31.);
DbiDemoData1 row2(0,12.,22.,32.);

// Store the silly data.
writer << row0;
writer << row1;
writer << row2;
```

   The `DbiWriter` will call `DbiDemoData1`'s Store method which is rather similar to its Fill method:-

```
void DbiDemoData1::Store(DbiOutRowStream& ors,
                         const DbiValidityRec* vrec) const {
  ors << fSubSystem << fPedestal << fGain1 << fGain2;
}
```

   Again notice that the SeqNo, which is part of the table row, but not part of the class data, is silently handled by the system.

4. Send the `DbiWriter` its Close message.

```
writer.Close();
```

7.5.4 Using DbiWriter

- The `DbiWriter`'s constructor is:-

```
DbiWriter(const VldRange& vr,
          Int_t aggNo,
          Dbi::Task task = 0,
          VldTimeStamp creationDate = VldTimeStamp(0,0),
          UInt_t dbNo = 0,
          const std::string& LogComment = ""
          );
```

  - The first argument determines the validity range of the data to be written, i.e. what set of `VldContexts` it is suitable for. You can control the date range as well as the type(s) of data and detector.
  - The second argument is the aggregate number. For unaggregated data it is -1, for aggregated data its a number in the range 0..n-1 where n is the number of aggregates.
  - The third argument is the Task of the data. It has a default of 0.
  - The fourth argument supplies the data's creation date. The default is a special date and time which signifies that `DbiWriter` is to use Overlay Creation Dates (see Concepts section 7.1.9.) Alternatively, at any time before writing data, use the method:-

    ```
    void SetOverlayCreationDate();
    ```

    to ensure that `DbiWriter` uses Overlay Creation Dates.
  - The fifth argument defines which entry in the database cascade the data is destined for. By default it is entry 0 i.e. the highest priority one.

    Caution: Supplying the entry number assumes that at execution time the cascade is defined in a way that is consistent with the code that is using the `DbiWriter`. As an alternative, you can supply the database name (e.g. offline) if you know it and are certain it will appear in the cascade.
  - The last argument supplies a comment for the update. Alternatively, at any time before writing data, use the method:-

    ```
    void  SetLogComment(const std::string& LogComment)
    ```

Update comments are ignored unless writing to a Master database (i.e. one used as a source database e,g. the database at FNAL), and in this case a non-blank comment is mandatory unless the table is exempt. Currently only DBI, DCS and PULSER tables are exempt.

If the first character on the string is the '@' character then the rest of the string will be treated as the name of a file that contains the comment. If using DbiWriter to write multiple records to the same table as part of a single update then only create a single DbiWriter and use the Open method to initialise for the second and subsequent records. That way a single database log entry will be written to cover all updates.

- Having instantiated a `DbiWriter`, filled table row objects must be passed using the operator:-

```
DbiWriter<T>& operator<<(const T& row);
```

  for example:-

```
writer << row0;
writer << row1;
writer << row2;
```

  `DbiWriter` calls the table row's Store method, see the next section. It also performs some basic sanity checks:-

  - The row's aggregate number matches its own.
  - The type of the data written is compatible with database table.

  If either check fails then an error message is output and the data marked as bad and the subsequent Close method will not produce any output.

- Once all rows for the current aggregate have been passed to `DbiWriter` the data can be output using:-

```
Bool_t Close();
```

which returns true if the data is successfully output.

- At any time before issuing the Close command you can cancel the I/O by either:-
  - Destroying the `DbiWriter`.
  - Using the method:-

    ```
    void Abort();
    ```

- If you want to, you can reuse a `DbiWriter` by using:-

  ```
  Bool_t Open(const VldRange& vr,
              Int_t aggNo,
              Dbi::Task task = 0,
              VldTimeStamp creationDate = VldTimeStamp(),
              UInt_t dbNo = 0);
  ```

  The arguments have the same meaning as for the constructor. An alternative form of the Open statement allows the database name to be supplied instead of its number. If the `DbiWriter` is already assembling data then the Close method is called internally to complete the I/O. The method returns true if successful. As explained above, the Open method must be used if writing multiple records to the same table as part of a single update for then a single database log entry will be written to cover all updates.

7.5.5 Table Row Responsibilities

All `DbiTableRow` sub-class objects must support the input interface accessed through `DbiResultPtr`. The responsibilities that this implies are itemised in section 7.4.2. The output interface is optional; the responsibilities listed here apply only if you want to write data to the database using this interface.

Must override GetAggregateNo method if aggregated `DbiTableRow` supplies a default that returns -1. The GetAggregateNo method is used to check that table row objects passed to a particular `DbiWriter` have the right aggregate number.

Must override Store Method The Store method is the inverse to Fill although it is passed a `DbiOutRowStream` reference:-

```
void Store(DbiOutRowStream& ors) const;
```

rather than a `DbiResultSet` reference. Both these classes inherit from `DbiRowStream` so the same set of methods:-

```
      string DbiResultSet::CurColName() const;
      UInt_t DbiResultSet::CurColNum() const;
      UInt_t DbiResultSet::NumCols() const;
DbiFieldType DbiResultSet::CurColFieldType() const;
      UInt_t DbiResultSet::CurRowNum() const;
      string DbiResultSet::TableName();
```

are available. So, as with the Fill method, there is scope for Store to be "smart". The quotes are there because it often does not pay to be too clever! Also like the Fill method its passed a `DbiValidityRec` pointer (which is only zero when filling `DbiValidityRec` objects) so that the validity of the row can be accessed if required.

7.5.6 Creating and Writing Temporary Tables

It is possible to create and write temporary tables during execution. Temporary tables have the following properties:-

- For the remainder of the job they look like any other database table, but they are deleted when the job ends.
- They completely obscure all data from any permanent table with the same name in the same database. Contrast this with the cascade, which only obscures data with the same validity.
- They are local to the process that creates them. Even the same user running another job using the same executable will not see these tables.

Temporary tables are a good way to try out new types of table, or different types of data for an existing table, without modifying the database. Writing data is as normal, by means of a `DbiWriter`, however before you write data you must locate a database in the cascade that will accept temporary tables and pass it a description of the table. This is done using the `DbiCascader` method `CreateTemporaryTable`. You can access the cascader by first locating the singleton `DbiTableProxyRegister` which is in overall charge of the DatabaseInterface. The following code fragment shows how you can define a new table for DbiDemoData1:-

```
#include "DatabaseInterface/DbiCascader.h"
#include "DatabaseInterface/DbiTableProxyRegistry.h"

...

//  Ask the singleton DbiTableProxyRegistry for the DbiCascader.
  const DbiCascader& cascader
              = DbiTableProxyRegistry::Instance().GetCascader();

//  Define the table.
  string tableDescr = "(SeqNo int,   SubSystem  int, Pedestal float,"
                     " Gain1 float, Gain2 float )";
//  Ask the cascader to find a database that will accept it.
  Int_t dbNoTemp = cascader.CreateTemporaryTable("DbiDemoData1",
                                                 tableDescr);

  if ( dbNoTemp < 0 ) {
     cout << "No  database to will accept temporary tables. " << endl;
  }
```

You pass `CreateTemporaryTable` the name of the table and its description. The description is a parenthesised comma separated list. It follows the syntax of the MYSQL CREATE TABLE command, see section 7.7.

In principle not every database in the cascade will accept temporary tables so the cascader starts with the highest priority one and works done until it finds one, returning its number in the cascade. It returns -1 if it fails. For this to work properly the first entry in the cascade must accept it so that it will be taken in preference to the true database. It is recommended that the first entry be the temp database, for everyone has write-access to that and write-access is needed to create even temporary tables. So a suitable cascade might be:-

```
setenv ENV_TSQL_URL "mysql:odbc://pplx2.physics.ox.ac.uk/temp;\
                     mysql:odbc://pplx2.physics.ox.ac.uk/offline"
```

Having found a database and defined the new or replacement table, you can now create a `DbiWriter` and start writing data as describe in section 7.5. You have to make sure that the DbiWriter will output to the correct database which you can either do by specifying it using the 5th arg of its constructor:-

```
DbiWriter(const VldRange& vr,
          Int_t aggNo,
          Dbi::Task task = 0,
          VldTimeStamp creationDate = VldTimeStamp(),
          UInt_t dbNo = 0);
```

or alternatively you can set it after construction:-

```
DbiWriter<DbiDemoData1> writer(range,aggNo);
writer.SetDbNo(dbNoTemp);
```

As soon as the table has been defined it will, as explained above, completely replace any permanent table in the same database with the same name. However, if there is already data in the cache for the permanent table then it may satisfy further requests for data. To prevent this from happening you can clear the cache as described in the next section.

> *Do NOT write permanent data to any temporary database for it could end up being used by anyone who includes the database for temporary tables. Database managers may delete any permanent tables in temporary databases without warning in order to prevent such problems.*

7.5.7 Clearing the Cache

Normally you would not want to clear the cache, after all its there to improve performance. However if you have just created a temporary table as described above, and it replaces an existing table, then clearing the cache is necessary to ensure that future requests for data are not satisfied from the now out of date cache. Another reason why you may want to clear the cache is to study database I/O performance.

Although this section is entitled Clearing the Cache, you cannot actually do that as the data in the cache may already be in use and must not be erased until its clients have gone away. Instead the data is marked as stale, which is to say that it will ignored for all future requests. Further, you don't clear the entire cache, just the cache associated with the table that you want to refresh. Each table is managed by a `DbiTableProxy` that owns a `DbiCache`. Both `DbiWriter` and `DbiResultPtr` have a `TableProxy` method to access the associated `DbiTableProxy`. The following code fragment shows how to set up a writer and mark its associated cache as stale:-

```
DbiWriter<DbiDemoData1> writer(range,aggNo);
writer.SetDbNo(dbNoTemp);
writer.TableProxy().GetCache()->SetStale();
```

7.6 ASCII Flat Files and Catalogues

7.6.1 Overview

ASCII flat files and catalogues provide a convenient way to temporarily augment a database with additional tables under your control. A flat file is a file that contains, in human readable form, the definition of a table and its data. It can be made an entry in a cascade and, by placing before other entries allows you to effectively modify the database just for the duration of a single job. As has already been explained, for each Main Data Table there is also an auxiliary Validity Range Table, so you need 2 entries in the cascade for each table you want to introduce. If introducing a number of tables the cascade could get rather large and to avoid this catalogues are used. A catalogue is actually nothing more that a special ASCII flat file, but each row of its data is a URLs for another ASCII flat file that becomes part of the same cascade entry. In this way a single cascade entry can consist of an arbitrary number of files.

7.6.2 Flat Files

An ASCII flat file defines a single database table.

Format

The format is sometimes referred to as Comma Separated Value (CSV). Each line in the file corresponds to a row in the table. As you might suspect, values are separated by commas, although you can add additional white space (tabs and spaces) to improve readability (but heed the caution in section 7.6.4). The first row is special, it contains the column names and types. The types must valid MySQL types, see table 7.1 for some examples. If the special row is omitted or is invalid then the column names are set to C1, C2, ... etc. and all types are set to string (TEXT). Here is a simple example of a CSV file:-

```
SeqNo int, Pedestal float, SubSystem int, Gain1 float, Gain2 float
1,    1.0,    0,    10.,    100.
1,    1.1,    1,    11.,    110.
1,    1.2,    2,    12.,    120.
1,    1.3,    3,    13.,    130.
```

Its in a convention to use the file extension .csv, but it is not compulsory.

URL

Two types of URL are supported:-

- Database URL. In this case, the host and everything that follows is replaced by a hash followed by the absolute path to the file, e.g.

  ```
  mysql://#/home/minos/work/MyFile.csv
  ```

- Web URL. This allows access to files in the local file store or on the web e.g.

  ```
      file:/home/minos/work/MyTable.csv
  and http://www-pnp.physics.ox.ac.uk/~west/MyTable.csv
  ```

The table name is derived from the file name after stripping off the extension. In this example, the table name will be MyTable

7.6.3 Catalogues

These are special types of ASCII Flat File. Their data are URLs to other flat files. You cannot nest them i.e. one catalogue cannot contain a URL that is itself catalogue.

Format

The first line of the file just contains the column name “name”. The remaining lines are URLs of the flat files. Here is a simple example:-

```
name
file:/home/minos/work/MyData.csv
file:/home/minos/work/MyDataVld.csv
file:$MY_ENV/MyDataToo.csv
file:$MY_ENV/MyDataTooVld.csv
```

This catalogue defines two tables MyData and MyDataToo each with its associated auxiliary validity range table. Note that files names must be absolute but can begin with an environmental variable.

URL

The URL is identical to any other flat file with one additional constraint: the extension must be .cat or .db. Here are some valid examples:-

```
mysql://#/home/minos/work/MyCatalogue.db
file:/home/minos/work/MyCatalogue.cat
```

7.6.4 Example

The stand-alone testing of the Database Interface includes an example of an ASCII Catalogue. The URL of the cascade entry is:-

```
mysql:odbc://localhost/test#$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DemoASCIICatalogue.db
```

If you look at the file:-

```
\$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DemoASCIICatalogue.db
```

you will see it contains 4 lines, defining the tables `DEMOASCIIDATA` (a Detector Descriptions table) and `DEMOASCIICONFIG` (Algorithm Configurations table):-

```
file:$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DEMOASCIIDATA.csv
file:$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DEMOASCIIDATAVLD.csv
file:$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DEMOASCIICONFIG.csv
file:$SRT_PUBLIC_CONTEXT/DatabaseInterface/test/DEMOASCIICONFIGVLD.csv
```

In both cases, the auxiliary validity range table defines a single validity range, although there is no reason why it could not have defined any number. For the `DEMOASCIIDATA`, there are 5 rows, a header row followed by 4 rows of data:-

```
SEQNO INT, UNWANTED INT, PEDESTAL FLOAT, SUBSYSTEM INT, GAIN1 FLOAT, GAIN2 FLOAT
1,99,1.0,0,10.,100.
1,99,1.1,1,11.,110.
1,99,1.2,2,12.,120.
1,99,1.3,3,13.,130.
```

For the `DEMOASCIICONFIG` table, there are only two rows:-

```
SEQNO INT, CONFIGSTRING TEXT
1,"mybool=1 mydouble=1.23456789012345678e+200 mystring='This is a string' myint=12345"
```

Caution: Note, don't have any white space between the comma and the leading double quote of the configuration string.

7.7 MySQL Crib

This provides the absolute bare minimum to install, manage and use a MySQL database in the context of the DatabaseInterface.

7.7.1 Introduction

The following are useful URLs:-

- MySQL home page:-

  http://www.mysql.com/

- from which you can reach a documentation page:-

  http://www.mysql.com/documentation/index.html

- and the downloads for 3.23:-

  http://www.mysql.com/downloads/mysql-3.23.html

A good book on MySQL is:-

MySQL by Paul DuBois, Michael Widenius. New Riders Publishing; ISBN: 0-7357-0921-1

7.7.2 Installing

This has to be done as root. You need to get and install the following packages:-

- The server for i386 systems
- Client programs for i386 systems
- Include files and libraries for development for i386 systems
- Client shared libraries for i386 systems

On Linux the rpms are:-

- MySQL-shared-3.23.33-1.i386.rpm
- MySQL-devel-3.23.33-1.i386.rpm
- MySQL-client-3.23.33-1.i386.rpm
- MySQL-3.23.33-1.i386.rpm

7.7.3 Running mysql

mysql is a utility, used both by system administrators and users to interact with MySQL database. The command syntax is:-

```
mysql [-h host_name] [-u user_name] [-pyour_pass]
```

if you are running on the server machine, with you Unix login name and no password then:-

```
mysql
```

is sufficient. To exit type:-

```
\q
```

Note: most mysql commands are terminated with a semi-colon. If nothing happens when you type a command, the chances are that mysql is still waiting for it, so type it and press return again.

7.7.4 System Administration

This also has to be done as root. As system administrator, MySQL allows you to control access, on a user by user basis, to databases. Here are some example commands:-

```
create database Minos_db1;
grant all on    Minos_db1.*       to  smart@pplx2.physics.ox.ac.uk
grant all on    Minos_db1.*       to  smart@''%''
grant select    Minos_db1.Boring  to  dumb@pplx2.physics.ox.ac.uk
\q
```

- The first lines creates a new database called Minos_db1. With MySQL you can have multiple databases.
- The next two lines grants user smart, either logged in locally to the server, or remotely from anywhere on the network all privileges to all tables in that database.
- The next line grants user dumb, who has to be logged in locally, select (i.e. read) access to the table Boring in the same database.

7.7.5 Selecting a Database

Before you can use mysql to create, fill or examine a database table you have to tell it what database to use. For example:-

```
use Minos_db1
```

'use' is one of the few commands that does not have a trailing semi-colon.

7.7.6 Creating Tables

The following commands create, or recreate, a table and display a description of it:-

```
drop table if exists DbiDemoData1;
create table DbiDemoData1(
    SeqNo      int,
    SubSystem  int,
    Pedestal   float,
    Gain1      float,
```

```
    Gain2      float
 );
describe DbiDemoData1;
```

See table 7.1 for a list of MySQL types that the DatabaseInterface currently supports.

7.7.7 Filling Tables

The following commands add data from the file DemoData1.dat to an existing table:-

```
load data local infile 'DemoData1.dat' into table DbiDemoData1;
```

Each line of the file corresponds to a row in the table. Columns should be separated with tabs. Table 7.2 shows typical formats of the various data types.

MySQL Type	Table Row Type
CHAR	a
TINYINT	-128
SMALLINT	-32768
INT or INTEGER	-2147483647
FLOAT	-1.234567e-20
DOUBLE	1.23456789012345e+200
TEXT	'This is a string'
DATETIME	'2001-12-31 04:05:06'

Table 7.2: Example data formats.

7.7.8 Making Queries

Here is a sample query:-

```
select * from DbiDemoData2Validity where
    TimeStart <= '2001-01-11 12:00:00'
and TimeEnd    > '2000-12-22 12:00:00'
and DetectorMask & 4
order by TimeStart desc
;
```

7.8 Performance

7.8.1 Holding Open Connections

Connections to the database are either permanent i.e. open all the time or temporary i.e. they are closed as soon as a I/O operation is complete. A connection is made permanent if:-

- Connecting to a ASCII flat file database as re-opening such a database would involve re-loading all the data.
- Temporary data is written to the database for such data would be lost if the connection were closed.

In all other cases the connection is temporary so as to minimise resources (and in the case ORACLE resources that have to be paid for!). For normal operations this adds little overhead as typically there are several major database reads at the start of a production job after which little or no further database I/O occurs. However if you require the connection to remain open throughout the job then you can force any entry in the cascade to be permanent. The following code sets entry 0 in the cascade to have a permanent connection:-

```
#include "DatabaseInterface/DbiCascader.h"
#include "DatabaseInterface/DbiTableProxyRegistry.h"


//  Ask the singleton DbiTableProxyRegistry for the DbiCascader.
  const DbiCascader& cascader
              = DbiTableProxyRegistry::Instance().GetCascader();
// Request that entry 0 is permanently open.
  cascader.SetPermanent(0);
```

Note that this won't open the connection but will prevent it from closing after its next use.

7.8.2 Timing

DbiTimerManager is a static object that provides performance printout when enabled. By default it is enabled but can be disabled by:-

```
   DbiTimerManager::gTimerManager.Enable(false);
```

Chapter 8

Navigation

Last significant change: 2004/08/011

8.1 Introduction

Moving between related objects of an object structure is called *Navigation*. This chapter describes a set of tools that have been developed to help navigate MINOS object structures. It is divided into the following sections:-

- Section 8.2 provides a full explanation on how the tools are used.
- Section 8.3 explains the need for a set of general purpose navigation tools.
- Section 8.4 outlines the functionality of the tools and sketches the classes that implement them.

Each section has been designed to be read in isolation; so if you are only interested in using the tools without the whys and the wherefores, you can just concentrate on section 8.2 and ignore the others.

8.2 Using the Tools

8.2.1 Naming Convention

Throughout this chapter the character strings Xxx and Yyy are placeholders which should be replaced with the appropriate class name. There is just one exception: the header file XxxItr.h really is typed like that i.e. Xxx is not a placeholder but is to be typed literally.

So, if you see an example like this:-

```
#include "Xxx.h"
XxxItr myItr;
```

and you want to apply it to a class called MyClass, then it should be written like this:-

```
#include "Xxx.h"
MyClassItr myItr;
```

8.2.2 Introduction

This section has been designed to be read without looking at the other sections of this chapter so some level of repetition is unavoidable. However it concentrates exclusively on explaining how to use the tools without discussing design issues.

The purpose of the tools is to aid navigation of the event and detector object structures. The navigation objects are transient, that is to say they are created as necessary and are not stored permanently. The idea is that designers of the event and detector object structures provide code to create the navigation objects on demand. The end user seems a uniform interface via these navigation objects.

The heart of navigation is the iterator. Its objective is to give access to a set of objects. In the Fortran days you might store a set of items in an array and then process them sequentially using an index. In the OO world the index is replaced by an iterator which can be asked repeatedly to return the next object until there is none left. The advantage is that the you no longer care where or how the objects are stored, all you need is a method to retrieve them.

The navigation tools have two types of iterators:-

- Those that operate on a single set of objects.
- Those that operate on a set of objects that are participate in a relationship with a second set of objects.

The sets are homogeneous i.e. all the members of the set are of the same class. or at least are derived from a common class. In both cases the iterator accesses objects from a single set and provides sequential and random access, sorting and selection services. The only difference between the two types is that the second type has an additional selection facility: to select the sub-set of objects associated with a single object from the other set of the relationship.

8.2.3 Creating the Class Code

Most of the navigation code is generic, that is to say it can support operations on any sets of objects. It does this by accessing all objects via void pointers. Such pointers are very powerful, they can point at any byte in memory, and so are very dangerous as well! So you never deal directly with these pointers but always use objects that pass the right type of pointer. This means that for any set of objects you wish to navigate there has to be an associated iterator class that returns this type of object. The class is named XxxItr where Xxx denotes the particular class name. So if you have a class called MyClass the tools require the creation of a class called MyClassItr. This code can be created using a C++ macro. Assuming that your class code has the normal .h header and .cxx implementation files, then all you need do is:-

1. In MyClass.h add:-

   ```
   #include "Navigation/XxxItr.h"
   XXXITRDEF(MyClass)
   ```

2. In MyClass.cxx add:-

   ```
   XXXITRIMP(MyClass)
   ```

3. In the LinkDef.h associated with your code add:-

```
#pragma link C++ class MyClassItr;
#pragma link C++ class MyClassKeyFunc;
#pragma link C++ class MyClassKeyFunctor;
```

The purpose of MyClassKeyFunc will be explained when we deal with sorting in section 8.2.6 and MyClassKeyFunctor when we deal with functors in section 8.2.9.

Note: Remember: the header file XxxItr.h really is typed like that i.e. Xxx is not a placeholder but is to be typed literally.

8.2.4 Creating Iterators for Supported Classes

The main section on creating tools objects will be deferred until section 8.2.16. It may seem strange that the section on using the tool objects precedes the main section that describes their instantiation but, as has been explained above, the tools assist a user to navigate event and detector object structures and that instantiation of the tool objects will be the responsibility of the designers of these object structures. So, for the most part, you will need to refer to the documentation on these classes for information or build them yourself as described in section 8.2.16

8.2.5 Sequential Access

Sequential access is the mostly commonly used functionality of an OO iterator. Indeed it is all that some types of iterator can do. Iterators are very popular and several different programming styles have developed. As far as possible the navigation tools support the main ones to help you become familiar with the different styles that you will undoubtedly encounter looking at external code.

Iterating without operator overloading

The first programming style is the most natural for anyone just starting with OO: no iterator operations involve operating overloading.

Suppose you have an XxxItr called myItr and the class Xxx has a Print() member function, then this will print out every one:-

```
while ( myItr.IsValid() ) {
  Xxx* myObj = myItr.Ptr();
  myObj->Print();
  myItr.Next();
}
```

The loop operates while the iterator's member function IsValid() returns true i.e. until there is no Xxx objects to return. The member function Ptr() returns a pointer to the current Xxx which can then be asked to Print() before the iterator is moved forward by asking it to perform its Next() function.

The iterator will return a zero if asked to return a pointer when IsValid() returns false and as 0 is the value of false you can, if you like to cut down lines of code, rewrite the above as:-

```
while ( Xxx* myObj = myItr.Ptr() ) {
  myObj->Print();
  myItr.Next();
}
```

For people who like to be minimalist (which may not be conducive to code transparency), this also works:-

```
myItr.Prev();
while ( Xxx* myObj = myItr.NextPtr() ) myObj->Print();
```

Prev() moves the iterator back (even if it was at the first object) and NextPtr() moves forward and returns the pointer to the object. At any time you can move back to the first object in the set with:-

```
myItr.ResetFirst();
```

As has already been hinted at, you can iterate over the set in reverse order if you want to:-

```
myItr.ResetLast()
while ( myItr.IsValid() ) {
  Xxx* myObj = myItr.Ptr();
  myObj->Print();
  myItr.Prev();
}
```

There is also a PrevPtr() if you need it.

Functors Iterators

Functors are objects that have the function operator () overloaded. This means that the object can be called rather like a function. This style is favoured by ROOT for its iterators. The Functor style is also popular for small “single-minded” objects. The member functions of the XxxItr class have been chosen to ensure that any segment of code using ROOT iterators would continue to work if replaced by these navigation iterators. To loop over all objects in the above example with this style you would code:-

```
while ( Xxx* myObj = myItr() ) myObj->Print();
```

myItr() isn’t a function call but a call to myItr operator() member function. This does two things:-

1. Returns a pointer to the current object.
2. Moves the iterator on in the current direction (initially forward).

You can flip the sign on the current direction at any time with:-

```
myItr.SetDirection(+1);    //Move forward
myItr.SetDirection(-1);    //Move backward
```

To be helpful, if you ask the iterator to ResetFirst() it will set the default direction is set forward and if you ResetLast() it will set the default direction is backwards. There is also:-

```
myItr.Reset();
```

which moves it back to the beginning relative to the current direction i.e. it is equivalent to ResetFirst() if the default direction is forwards otherwise it is equivalent to ResetLast()

STL like Iterators

STL is the Standard Template Library which comes complete with a set of containers (objects that hold other objects) and iterators that operate on these containers. STL is part if the standard C++ so the STL style effectively becomes the standard for iterators. STL views iterators as if they were pointers to an array of objects, consequently the operators ++ and `--` move the pointer forwards and backwards and the dereference operator * returns the object itself.

The tools only partially support this style. In particular:-

- Only the prefix ++ and -- operators are supported (++myItr is O.K. but not myItr++).
- In STL you get the start and end limits of the iterator from the container; with the tools you have to use IsValid() to check.

This is how you would iterate:-

```
while ( myItr.IsValid() } {
  (*myItr).Print();
  ++myItr;
}
```

As before ResetFirst() and ResetLast() moves to the start and end of the set.

Its is possible, but probably not advisable, to mix all of the above 3 styles in a single loop!

8.2.6 Sorting with a Single Function

It is often useful to be able to process all the objects in a set in some predefined order. The tools support this and, because of the way they work allow you to sort the set without disturbing the underlying object structure. This is possible because they have their own private copy of the set stored in a NavSet object, and your XxxItr object connects to, and iterates over, this set. This also means you can create multiple iterators simultaneously and have each ordered in its own way.

The way ordering works is that you have to tell the tools how to assign a value to every member of the set and then they sort the set into ascending order based on the value. If you want to processes in descending order you can just process the set backwards of course. The type of value you assign must be a simple type: an integer, floating point, or character string and its value must be derived from public member functions of the objects in the set. To support the notion of a sortable value, the tools have a type of object called a NavKey with the following properties:-

- Can be constructed from the simple types: integer, floating point or character string.
- Can be compared to another NavKey.

Your first job is to write a little function which takes a pointer to your Xxx object and returns a NavKey. As NavKeys can be created directly from simple types, writing this function is simple. Suppose Xxx has the member function Time() that returns a Float_t, then you could create a function, say KeyFromTime:-

```
#include "Navigation/NavKey.h"
static NavKey KeyFromTime( Xxx* obj) { return obj->Time(); }
```

The expression `obj->Time()` is of type Float_t but the compiler wants a NavKey, however it knows how to create a NavKey from a Float_t so creates one. Remember in C and C++ all functions have to be defined at the file level, they cannot be embedded within other functions to hide them. The 'static' keyword at least keeps the name local to the file so the name KeyFromTime won't conflict with the same name defined in other files.

In a similar way, if Xxx has the member function Name() that returns a pointer to Char_t and PlaneNo() that returns Int_t you could write:-

```
static NavKey KeyFromName( Xxx* obj) { return obj->Name(); }
static NavKey KeyFromPlaneNo( Xxx* obj) { return obj->PlaneNo(); }
```

The function names are arbitrary, but it makes sense to be consistent and choose something logical.

Having written the function, you next job is to get it applied. It might seem reasonable for XxxItr to have a member function that could take your function as an argument and apply it. That would work but is flawed for several reasons:-

- Its not the iterator you want to sort but the NavSet it is connected to. So it is better that you ask the XxxItr for its NavSet and then pass the function to it.

- NavSets are generic: they don't know about Xxx objects so cannot accept your function. Instead you ask your XxxItr for an XxxKeyFunc. This is a little wrapper object that will accept your function and it is this that can be passed to the NavSet. This has the advantage that XxxKeyFunc objects can be "programmed" in other ways besides the method described here. The code for XxxKeyFunc class is created by the same macros that create the XxxItr code; so once you have an XxxItr you also have its companion XxxKeyFunc

This is what the code looks like to sort Xxx by Time:-

```
#include "Navigation/NavKey.h"
#include "Navigation/NavSet.h"

...


// Create an XxxKeyFunc.  Any member function that has
// the word "Create" means that you, as the caller are going
// to be the owner of a new object and will need to look after it!
XxxKeyFunc* myKf = myItr.CreateKeyFunc();

// Program it with the sort function.
mkKf->SetFun(KeyFromTime);

// Ask the iterator for its NavSet and then pass the
// XxxKeyFunC to it. The name "Adopt" means that the NavSet
// now owns the XxxKeyFunc and can delete it anytime it
// pleases e.g. when asked to adopt another.
myItr.GetSet()->AdoptSortKeyFunc(myKf);

// As you no longer own the XxxKeyFunc the safest thing is
// to clear the pointer so that you are not tempted to
// use it again!
myKf = 0;
```

> *Important: When a NavSet is given a sort function it resets all its iterators by sending them their ResetFirst() message.*

You can change the sort function at any time and as often as you like, but bear in mind that the iterator gets reset each time you do. NavSet applies the ordering at once and all iterators will process in this order. However if subsequently anyone changes the state of any objects in the underlying set that NavSet set mirrors this will not be reflected in the order processed. For example, suppose you have a set of digitizations and your NavSet of it is ordered by time. Further suppose you then recalibrate the digitizations and it changes the times to such an extent that the original time ordering is wrong, the NavSet isn't aware of the change, so its order won't change. If you know this may have happened you can refresh (see 8.2.15).

You can cancel sorting by passing a null pointer to AdoptSortKeyFunc:-

```
myItr.GetSet()->AdoptSortKeyFunc(0);
```

This won't unsort the set of course but will mean that the NavSet will not attempt to maintain order after operations such as mapping (see 8.2.8)

8.2.7 Sorting with Multiple Functions

In the previous section we have seen examples sorting by time or by plane number, but what if you wanted to sort first by plane and then, within each plane, by time? This can be done by using the AdoptSortKeyFunc method twice. For example:-

```
#include "Navigation/NavKey.h"
#include "Navigation/NavSet.h"

...

XxxKeyFunc* myKf = myItr.CreateKeyFunc();
mkKf->SetFun(KeyFromPlaneNo);
myItr.GetSet()->AdoptSortKeyFunc(myKf);

myKf = myItr.CreateKeyFunc();
mkKf->SetFun(KeyFromTime);
myItr.GetSet()->AdoptSortKeyFunc(myKf,kFALSE);

myKf = 0;
```

The second argument to AdoptSortKeyFunc, which defaults to kTRUE, controls the deletion any existing key functions. The first call above removes any existing sort function before adding KeyFromPlaneNo but the second call just adds the key function KeyFromTime to the end of the list.

In this way you can combine two or more key functions to fully specify the sort order. Each time you append a new key function, the set is resorted and the ordering further refined. Of course it is a little wasteful sorting before all the required functions are given to the `NavSet` so you can suppress sorting by passing a third argument, that defaults to kTRUE, to AdoptSortKeyFunc:-

```
  ...

// Suppress sorting  until second function has been supplied
```

```
  myItr.GetSet()->AdoptSortKeyFunc(myKf,kTRUE,kFALSE);

  ...

// Pass second function and sort.
  myItr.GetSet()->AdoptSortKeyFunc(myKf,kFALSE);
```

8.2.8 Selection

General Principles

You can apply several different forms of selection and the result is that you only see the reduced set until the selection is cancelled. The Reset* messages move the iterator to the first or last selected object and the sequential access only returns selected objects. However the Size() member function continues to return the full size of the set; use the SizeSelect() member function to see how many are currently selected. With no selection in force, SizeSelect() is equivalent to Size().

> *Important: When a selection is applied or cancelled on a NavSet the NavSet resets all its iterators by sending them their ResetFirst() message.*

If more than one form of selection is in force, they form an AND - objects are selected only if all criteria are satisfied. See also Masks (section 8.2.10) for an alternative way to select set members.

Selection by Slicing in One Dimension

Slicing means selecting only those objects whose key value lies within a specified range of values. Suppose you had sorted your Xxx objects by time and now only wanted to process those that lie within the time window 10. to 50. To do this you need to send NavSet its Slice message:-

```
   myItr.GetSet()->Slice(10.,50.);
   cout << "There are " << myItr.SizeSelect()
        << " selected objects."" << endl;
```

after which myItr will be reset to the start of the slice ready for you to process all the Xxx objects in the time range. If the lower and upper limits coincide you can supply a single argument. For example:-

```
myItr.GetSet()->Slice(30.);
```

selects objects whose time is exactly 30. Of course for floating point values, requiring an exact match like this may fail if rounding errors are involved.

Slicing will fail unless:-

- A sort is currently in force. In there is no sort you don't get an error; the slice simply doesn't reject anything.
- The type of sort key (integer, float or character string) matches the type of both arguments of the Slice call.

As with sorting, the slice will not reflect subsequent changes to the objects in the set. Use the Refresh message (8.2.15) if necessary.

To cancel slice selection call ClearSlice:-

```
myItr.GetSet()->ClearSlice();
```

As this is a change in selection, all iterators get reset.

To change the slice, first reset it and then apply the new slice e.g.:-

```
myItr.GetSet()->ClearSlice();
myItr.GetSet()->Slice(50.,100.);
```

> *Important: Don't apply a second slice without resetting; that will attempt to slice in the second dimension - see below.*

Selection by Slicing in Multiple Dimensions

Just as with sorting, you can apply multiple slices. Suppose you have applied multiple sorts first by plane number and then by time. In which case the statements:-

```
myItr.GetSet()->Slice(1,100.);
myItr.GetSet()->Slice(10.,50.);
```

would reject any Xxx objects whose plane number was not within the range 1..100 or whose time as not in the range 10.0..50.0.

The system generalises to any (reasonable!) number of dimensions: the nth sort function is used by the nth slice. As with sorting you can suppress the operation of selection until all slices have been defined by using an optional second argument that defaults to kTRUE:-

```
myItr.GetSet()->Slice(1,100.,kFALSE);
myItr.GetSet()->Slice(10.,50.);
```

The first slice over plane number remains dormant until the second slice over time is applied.

Selection by User Function

Selection by user is very similar to sorting (8.2.6); the user defines a function that returns a NavKey, wraps it in an XxxKeyFunc, and passes it to the NavSet. In this case the function has just to return true or false (non-zero or zero), and the XxxKeyFunc is passed to the NavSet by its AdoptSelectKey-Func message.

Suppose you want to select Xxxs that have either a time less than 50. or a plane number greater that 100, then you could code a selection function like this:-

```
NavKey SelectPlaneNoTime( Xxx* obj) { return obj->PlaneNo() >
100. || obj->Time() < 50.; }
```

Having written the function, you apply it like this:-

```
// Create an XxxKeyFunc.  Any member function that has
// the word "Create" means that you, as the caller are going
// to be the owner of a new object and will need to look after it!
XxxKeyFunc* myKf = myItr.CreateKeyFunc();

// Program it with the sort function.
mkKf->SetFun(KeyFromTime);

// Ask the iterator for its NavSet and then pass the
// XxxKeyFunC to it. The name "Adopt" means that the NavSet
// now owns the XxxKeyFunc and can delete it anytime it
// pleases e.g. when asked to adopt another.
myItr.GetSet()->AdoptSelectKeyFunc(myKf);

// As you no longer own the XxxKeyFunc the safest thing is
// to clear the pointer so that you are not tempted to
// use it again!
myKf = 0;
```

In the normal course of events the object set is not expected to change while navigation is in progress, but if it does while looping over them with an iterator, you may miss some or may retrieve some that subsequently should have been ignored. Navigation of a set that is changing is always a potential problem, see Refreshing Iterators (8.2.15) for a discussion.

To cancel selection by user function pass NavSet a null pointer:-

```
myItr.GetSet()->AdoptSelectKeyFunc(0);
```

As with AdoptSortKeyFunc, AdoptSelectKeyFunc can take a trailing argument that if set to kFALSE suppresses the application of the selection:-

```
myItr.GetSet()->AdoptSelectKeyFunc(myKf,kFALSE);
```

However, unlike AdoptSortKeyFunc, applying another AdoptSelectKeyFunc will replace the existing function, it nevers appends a second one. This might lead you to wonder why you would want to suppress selection. This will become clear in section 8.2.11 which discusses simple optimisation techniques.

Inverting Selection by User Function

You can invert the meaning of your selection function by using:-

```
myItr.GetSet()->InvertSelection();
```

which reapplies it but only selects those members that fail the selection cut. To switch back to the normal selection and reapply it:-

```
myItr.GetSet()->InvertSelection(kFALSE);
```

Whenever a new selection function is adopted, normal selection applies. As with AdoptSelectKeyFunc, InvertSelection can take a trailing argument that if set to kFALSE suppresses the application of the selection.

Selection by Mapping

Mapping is the process by which a relationship is navigated. Suppose you have two sets A and B that have a relationship, that is to say that each member in one set is associated with zero or more members in the other set. You can select on this association. For example, if you have an iterator over the A set, you can select those that are associated with an individual B set member by passing the NavSet a pointer to the B set member. If that member has no association, or if you pass a pointer to a B that isn't a member of the B set then, as you might expect, you get an empty A sub-set. This is also true unless your iterator was created for a set that is participating in a relationship.

To illustrate the idea suppose your Xxx set is in 1:n relationship with a second set of Yyy objects, that is to say for every Xxx there is 1 or more Yyy but each Yyy has only one Xxx. Further suppose both Xxx and Yyy have a Name() member function. This is how you could loop over all Xxx objects and list the Yyys it is associated with:-

```
//  Include the class headers (which should both use the XXXITRDEF to
//  declare the XxxItr and YyyItr classes).
#include "Xxx.h"
#include "Yyy.h"
#include "Navigation/NavSet.h"

//  Create iterators over both sets in the relationship.
//  This code will only work if this relationship is built in;
//  you will have to see other documentation for other relationships.
XxxItr itrXxx("Yyy");
YyyItr itrYyy("Xxx");

cout << "The Xxx set has " << itrXxx.Size() << " members" << endl;
cout << "The Yyy set has " << itrYyy.Size() << " members" << endl;

//  Outer loop over all the Xxx set.

while ( itrXxx.IsValid() {

  Xxx* myXxx = itrXxx.Ptr();

//  Subselect Yyy to the association with this Xxx.
  itrYyy.GetSet()->Map(myXxx);
```

```
    cout << "Xxx " << myXxx->Name() << " is associated "
         << itrYyy.SizeSelect() << " Yyy objects:-" << end;

  // Inner loop over all Yyy associated with this Xxx.

    while ( itrYyy.IsValid() (

      Yyy* myYyy = itrYyy.Ptr();
      cout << "  Yyy " << myYyy->Name() << endl;
      itrYyy.Next();
    }

  itrXxx.Next();

  }
```

If you know the result of the mapping is a single member subset, it is a bit tedious to perform the map and then pick up the first (and only) set member so, to allow a shortcut, you can pass the Map request directly to your iterator and have it return a pointer to the first set member.

> *Important: Sending a Map message to an XxxItr still results it its NavSet being sent its Map message and, as with other forms of selection, results in all the NavSet's iterators being sent their ResetFirst() message.*

In our example each Yyy is associated with exactly one Xxx so you could write a loop over all Yyy set members and list their association like this:-

```
  // Undo any selection by mapping in force on Yyy iterator.
  // This will also reset the iterator to the start of its range.
  itrYyy.Map(0);

  while ( itrYyy.IsValid() ) {

    Yyy* myYyy = itrYyy.Ptr();
    Xxx* myXxx = itrXxx.Map(myYyy);
    cout << "Yyy " << myYyy->Name() << " is associated with  "
         << myXxx->Name() << end;
    itrYyy.Next();
  }
```

Note that supplying a null pointer to NavSet's Map function:-

```
itrYyy.Map(0);
```

cancels the mapping selection and, like other changes in selection, resets the iterators.

8.2.9 Functors: Programmable Sorting and Selection

In the previous sections you have seen how it is possible to write key functions that can control sorting and selection. A disadvantage of these functions is that they have to be defined when you write them; they cannot be changed during execution. Take for example the function:-

```
NavKey SelectPlaneNoTime( Xxx* obj)
   { return obj->PlaneNo() > 100. || obj->Time() < 50.; }
```

Such a function is not much good if you don't know until execution time what plane number and time limits to impose. You might like to be able to create an object that can hold the limits you want and then use that to select. Actually this isn't too hard to do just using the power of inheritance. Along with the XxxKeyFunc class, the Navigation package also defines the abstract XxxKeyFunctor class that you can inherit from. To pursue the example we could write a functor class MyXxxKeyFunctor:-

```
class MyXxxKeyFunctor : public XxxKeyFunctor {
public:
   MyXxxKeyFunctor(Int_t pLim = 100, Float_t tLim = 50.) :
fPlaneNoLim(pLim),
fTimeLim(tLim)
{ };
  ~MyXxxKeyFunctor(){ }
  virtual NavKey operator()(const Xxx* obj) const;
  virtual XxxKeyFunctor* Clone() const {
           return new MyXxxKeyFunctor(*this); }
private:
  Int_t   fPlaneNoLim;
  Float_t fTimeLim;
};
```

It inherits from XxxKeyFunctor and has two state variables fPlaneNoLim and fTimeLim that will hold the limits. These are initialised using the constructor arguments pLim and tLim, and take defaults that correspond to the key function it will replace. It has the function operator overloaded (that's what makes it a functor). Here is its implementation:-

```
NavKey XxxKeyFunctor::operator()(const Xxx* obj) const {
  return obj->PlaneNo() > fPlaneNoLim || obj->Time() < fTimeLim; }
}
```

It also has a Clone() method that can be used to make copies of itself. In our simple example:-

```
  virtual XxxKeyFunctor* Clone() const {
            return new MyXxxKeyFunctor(*this); }
```

works because the compiler can provide a default copy constructor. If your function does clever things and allocates memory on the heap then you had better override this. Note that, although it creates a new functor of yours (`MyXxxKeyFunctor`) Clone actually returns a pointer to the abstract `XxxKeyFunctor` base class. The reason that the Clone method is there is that this allows you to safely create your functors on the stack without worrying about memory leaks for the Navigation package makes its own copy to be used for as long as it needs to.

The object is used in place of a function. For example to place a selection plane limit of 20 and a time limit of 10:-

```
  XxxKeyFunc* myKf = myItr.CreateKeyFunc();

  // Create a stack based functor and store a clone
  // in the XxxKeyFunc
  MyXxxKeyFunc functor(20,10.);
  mkKf->SetFun(functor);

  // Now store the XxxKeyFunc
  myItr.GetSet()->AdoptSelectKeyFunc(myKf);
  myKf = 0;
```

Create your functors on the stack to avoid possible memory leaks.

Although functors are probably most useful for selection they can also be used for sorting. As they are programmable they can be as complicated as you like. You could even have a functor single class with multiple modes that could be handle all selection and ordering for the Xxx class and select the required mode at construction time.

8.2.10 Masks

The Navigation tools provide a non-invasive way of associating a mask with each member of the set. Masks are of type NavMask::Mask_t which is defined to be `UShort_t` so you can have up to 16 flags to mark set members in any way you like. Currently the interface is rather crude but can be refined if required. The masks are held in a separate `NavMask` class that you can access via its owing `NavSet`:-

```
NavMask& GetMasks() { return fMasks; }
```

Once you have a `NavMask` you can get and set masks using:-

```
Mask_t GetMask(void* obj) const;
void SetMask(Mask_t mask, void* obj);
```

or set all masks using:-

```
 void SetAllMasks(Mask_t mask);
```

In all cases obj is the address of the object you want to set. Although this class is meant to be used in conjunction with a `NavSet`, you can cheat and pass any address which will then be associated with a mask.

As an example the following statements test bit 2 of the mask associated with object currently pointed to by the iterator `itr`

```
NavMask& mask = itr.GetSet()->GetMasks();
if ( mask.GetMask(itr.Ptr()) & 1<<2 ) {
  cout << ‘‘Bit 2 is set 2 << endl;
}
```

8.2.11 Optimising Sorting and Selection

Delayed Updating

As we have seen, its is possible to apply multiple sort functions and slices as well as applying a selection function. Each of these operations involve changes to the underlying NavSet and might be quite expensive. For example, suppose you first applied a slice (which would do nothing as there is no corresponding sort) and the apply a sort. This activates the slice to reject unwanted entries of the NavSet and then sorts the rest. If you are going to apply multiple operations before you start to access the data you can tell the NavSet not keep up to date, but just to record the operations. The NavSet methods ClearSlice, Slice(NavKey, NavKey), AdoptSelectKeyFunc and AdoptSortKeyFunc all have a final optional argument:-

```
Bool_t update=kTRUE
```

By setting this to kFALSE, the NavSet is not updated. When you are finished you now have to use the Update method to get things back into sync. For example:-

```
  // Program with the 2 sort functions.
     myItr.GetSet()->AdoptSortKeyFunc(myKf1,kTRUE, kFALSE);
     myItr.GetSet()->AdoptSortKeyFunc(myKf2,kFALSE,kFALSE);

...

  // Apply a couple of slices.
   myItr.GetSet()->Slice(1,100., kFALSE);
   myItr.GetSet()->Slice(10.,50.,kFALSE);

...

  // Add in a selection function.
    myItr.GetSet()->AdoptSelectKeyFunc(myKf3, kFALSE);

  // Now bring NavSet up to date.
    myItr.GetSet()->Update();
```

Avoid Excessive Slicing

As currently implemented, slicing is considered to be part of selection, and as such is more global than sorting. A consequence of this is that each time you:-

```
    myItr.GetSet()->ClearSlice();
```

you are removing slice selection and this forces a re-sort of the full set.

If, having sorted a set, you want to just loop over a subset which corresponds to a single NavKey value (say myValue), code the loop as follows:-

```
for ( myItr.Reset();
      myItr.IsValid();
      myItr.Next() ) {

  Int_t comp = myItr.GetNavKey().CompareValue(myValue);
  // Quit if we have passed the value and skip if we haven't reached it yet.
  if ( comp > 0 ) break;
  if ( comp < 0 ) continue;

  ...

}
```

This saves time by quiting from the loop once the required value has been passed.

A rewrite of Navigation is planned and the way slicing and sorting interact will be reviewed at that time.

A Final Word

One final word on optimising. If performance is really essential and all you are doing is iterating sequentially then you might consider forgoing any sort and selection features and simply use whatever iteration techniques are available from the underlying set on which NavSet is built. The Navigation package was written to give a full function access but this does come at the price of creating a NavSet which does take time to create and maintain.

8.2.12 Iterating in Multiple Dimensions

Iterating in multiple dimensions is a natural extension of sorting in multiple dimensions. Suppose you have sorted a set first by plane number and then by time, sorting in multiple dimensions would mean having an outer loop iterator that accesses each plane number in turn with a inner loop that iterates over all times for the current plane.

Code to do the above is as follows:-

```
#include "Navigation/NavKey.h"
#include "Navigation/NavSet.h"

...

XxxItr myPlaneItr;  // Set up by some code!
```

```
//  Set up the sort functions.

XxxKeyFunc* myKf = myPlaneItr.CreateKeyFunc();
mkKf->SetFun(KeyFromPlaneNo);
myPlaneItr.GetSet()->AdoptSortKeyFunc(myKf);

myKf = myPlaneItr.CreateKeyFunc();
mkKf->SetFun(KeyFromTime);
myPlaneItr.GetSet()->AdoptSortKeyFunc(myKf,kFALSE);

myKf = 0;

//  Outer loop over plane number

for ( myPlaneItr.Reset();
      myPlaneItr.IsValid();
      myPlaneItr.NextKey() ) {

//    Code to access object at myPlaneItr;

//  Inner loop over times

  for ( XxxItr myTimeItr(myPlaneItr,kTRUE);
        myTimeItr.IsValid();
        myTimeItr.Next() }

//    Code to access object at myTimeItr;

  }

}
```

You have seen all of the above functions before except the following:-

- `myPlaneItr.NextKey()`
  This moves myPlaneItr on until the value in the primary sort key (plane number in this case) changes. There is also a PrevKey(). They both return kFALSE if there are no further primary keys.

- `XxxItr myTimeItr(myPlaneItr,kTRUE)`
  This is a special form of the copy constructor (see section 8.2.14 for a discussion of the standard copy constructor). Its sets myTimeItr to be

at the same position as myPlaneItr but it asks myPlaneItr what sort key is is tied to (the rank 0 sort in this case) and ties myTimeItr to the rank one higher (rank 1 i.e. time here). It further performs a slice that is local to myTimeItr so that it only sees those objects that have the same plane number as myPlaneItr. Once myTimeItr.IsValid() returns kFALSE the next object will have a new plane number.

The scheme can be extended 3 or more dimensions by adding a further sort functions, using NextKey for all but the innermost iterator and creating all but the outermost iterator using the special copy constructor with the iterator from the level above.

8.2.13 Random Access

Although their primary function is to give sequential access to a set, iterators can also give random access using the:-

```
Bool_t Jump(Int_t index)
```

member function. If the index is valid i.e. is within range and the corresponding object is selected then the iterator moves to the new position and the function returns kTRUE. Otherwise the iterator remains unchanged (which might also be invalid) and kFALSE is returned. E.g.:-

```
if ( itr.Jump(6) ) cout << "Iterator moved to position 6" << endl;
else               cout << "Position 6 is invalid" << endl;
```

You can also save the current index using GetIndex and restore it later:-

```
Int_t savePos = itr.GetIndex();

...

// Restore position
itr.Jump(savePos);
```

In case you were wondering, there is a SetIndex(Int_t index) and it is equivalent to Jump(Int_t index)

Some points to note about jumping:-

- For a set without any selection at all the valid index range is 0..itr.Size() - 1 just like a C array.

- Sorting and selection by mapping both shuffle the set members; so attempting to use a previously saved index will probably access the wrong member!
- Other forms of selection can make a previously saved index invalid i.e. if its object is no longer selected.

For all of the points given above, jumping should be used with caution. It is probably best used without any form of selection and after any required sorting has been applied.

If you are confident that you know the jump will suceed you can use JumpPtr, which both jumps are retrieves the pointer (cf NextPtr):-

```
cout << "Object at position 6:" << itr.JumpPtr(6)->Print() << endl;
```

8.2.14 Cloning Iterators

There are times when it is useful to have several iterators in operation at once. Having created a single iterator, it is very simple to create a copy by cloning it with the copy constructor:-

```
XxxItr myItr2(myItr);
```

This copies the exact state, including the current position and direction of myItr into the new iterator myItr2. After this they can be operated on independently. Further iterators can be manufactured in the same way. To bring any two back into step just assign one to the other:-

```
// Move myItr to the current position and direction of myItr2.
myItr = myItr2;
```

You can also tie one iterator to a sub-slice of another. In section 8.2.12 we saw how:-

```
XxxItr myTimeItr(myPlaneItr,kTRUE)
```

created myTimeItr sliced to the current plane number value associated with myPlaneItr. An alternative way to do after creation of myTimeItr is:-

```
myTimeItr.SubSlice(myPlaneItr)
```

> *Important: Cloned iterators all operate on the same set; changes in sorting and selection will affect them all in the same way. In particular, changes to sort order and selection will cause all iterators to be reset.*

If you want your iterators to be truly independent; each with its own sort and select then create iterators from scratch rather than by cloning them.

8.2.15 Refresing Iterators

The navigation tools are primarily designed to navigate stable object structures. The idea is that some Monte Carlo or reconstruction process creates the object structure and also provides a method to create an iterator to allow you to explore the results of the process. Attempting to navigate an object structure that is still being changed is a risky business! To require that an iterator do a complete revalidation of its view of a set (as held in the NavSet), each time you used it, would incur an unreasonable overhead. Instead it assumes that nothing has changed and, if it has, this could result in:-

- Missing members, if new objects are added to the set.
- Accessing non-existent members and probably crashing the program, if existing set members are deleted.
- Accessing out of sequence, if object state changes.
- Selecting the wrong members, if object state changes.

Some operations will reflect changes, due to the way that the system is implemented. In particular:-

- Mapping rebuilds the subset (or full set in cancelling) from the Lattice and then reapplies any sort and selection so will reflect the current state of the object set.

but it would be unwise to rely on these "accidental" features.

If you really need to navigate a changing object structure because you are building it and want to use the tools, then you can, at any time, issue a refresh request to the NavSet:-

```
myItr.GetSet()->Refresh();
```

This is an expensive operation as it has to rebuild the set, resort, reselect and finally reset all iterators. In case you were wondering about the difference between Refresh and Update (see 8.2.11), after Refresh NavSet will reflect the current state of the underlying object structure while Update will only ensure that the NavSet is consistant with its internal sorting, slicing and selections.

8.2.16 Creating the Navigation Objects

Creating Iterators

The navigation tools have two types of iterators:-

- Those that operate on a single set of objects. These iterators are built from TCollection objects, which is the base class for all ROOT containers:-

```
#include "TCollection.h"

TCollection* mySet;  // A ROOT container of Xxx objects.

// Create a iterator over the set.
XxxItr myItr(mySet);

// The iterator can also be created from a ROOT iterator
// using either a pointer to a TIter:-
TIter* myRootItr2 = new TIter(mySet);
XxxItr myItr2(myRootItr2);

// or the TIter itself:-
TIter  myRootItr3(mySet);
XxxItr myItr3(myRootItr3);

// or use a default constructor ...
XxxItr myItr;
// ... and then attach it to a ROOT container
myItr.Attach(mySet);
```

  Note that the last of the examples means that an iterator can be a data member of some class and be built with the default constructor. Of course it just behaves as if its connected to an empty set until the Attach method is used to attach it to a TCollection of the right type of objects. Alternatively it can be initialised by copying another, possibly temporary, iterator to it, e.g.:-

```
//  Create a tempoary iterator just to initialise the data member.
{
  XxxItr myItr3(myRootItr3);
  myIter = myItr3;
}
```

- Those that operate on a set of objects that are participate in a relationship with a second set of objects. These iterators are built from Lattice objects:-

```
#include "Lattice/Lattice.h"

Lattice* myXxxWithYyy;  // A Lattice holding a relationship:-
                        //   o A set of Xxx objects on the left
                        //   o A set of Yyy objects on the right

// Create an iterator over the Xxx set.
XxxItr itrXxx(myXxxWithYyy, Lattice::kLeft));

/ Create an iterator over the Yyy set.
YyyItr itrYyy(myXxxWithYyy, Lattice::kRight);
```

For information on the creation of Lattice object, see 8.2.16.

Creating Lattices

The concept, and much of the code for the Lattice package, was written by Jon Thaler for the CLEO experiment. A Lattice is an object that can hold relationships between two sets of objects:-

```
Class Xxx objects   <-- Lattice --> Class Yyy objects
```

The relationship between them can be:-

1:n Each object of class Xxx is associated with 1 or more from class Yyy but each object of class Yyy is only associated with one object of class Xxx.

n:1 Each object of class Yyy is associated with 1 or more from class Xxx but each object of class Xxx is only associated with one object of class Yyy.

n:m An object of either class is associated with 1 or more object of the other.

All lattices are bi-directional; you can navigate in either direction. You update a lattice by giving a relationship in one direction and the lattice automatically builds its inverse.

Although it is possible to create Lattices directly, a helper class, called a `LatticeMaker` can simplify things for you. For example:-

```
#include "Lattice/LatticeMaker.h"

LatticeMaker maker;
Lattice* myLat = maker.CreateLattice("Xxx","Yyy", "n:m");
```

creates a `Lattice` on the heap between objects of Xxx and Yyy in an n:m relationship. n:m is the default if the last argument is omitted or invalid, other legal values are 1:n and n:1.

> *Important: LatticeMaker always creates Lattices on the heap and gives them to you; it is your responsibility to delete the Lattice when no longer needed.*

Given the `Lattice myLat` created above, and assuming the declarations:-

```
Xxx* myObjXxx;
Yyy* myObjYyy;
```

then the statement:-

```
myLat->ConnectLR(myObjXxx,myObjYyy);
```

makes a left to right connection between them.

Putting this all together, suppose:-

1. `Xxx` and `Yyy` both inherit from TObject.

2. `mySetXxx` is some type of ROOT TCollection of Xxx objects.

3. `mySetYyy` is some type of ROOT TCollection of Yyy objects.

4. There is an association function:-

   ```
   Bool_t IsAssociated(const Xxx* objXxx, const Yyy* objYyy);
   ```

   that returns true if the two objects are associated and false otherwise.

then the following code builds a general n:m Lattice between them:-

```
#include "Lattice/LatticeMaker.h"

LatticeMaker maker;
Lattice* myLat = maker.CreateLattice("Xxx","Yyy", "n:m");

TIter itrXxx(&mySetXxx);
TIter itrYyy(&mySetYyy);

// Get each member of mySetXxx, converting from TObject* to const Xxx*
itrXxx.Reset();
while (
       ( const Xxx* objXxx = dynamic_cast<const Xxx*>( itrXxx() ) )
       ){

// Get each member of mySetYyy, converting from TObject* to const Yyy*
  itrYyy.Reset();
  while (
         ( const Yyy* objYyy = dynamic_cast<const Yyy*>( itrYyy() ) )
        ) {
//  Form connection if required.
    if ( IsAssociated(objXxx,objYyy) ) lat->ConnectLR(objXxx,objYyy);
  }
}
```

This method is so generic that it is built into `LatticeMaker` itself. However, it does not know about your Xxx and Yyy classes so expects an association function of the form:-

```
Bool_t AssociateFunc(const TObject* objXxx, const TObject* objYyy);
```

such a function would be easily built from our `IsAssociated` function:-

```
Bool_t AssociateFunc(const TObject* objXxx, const TObject* objYyy) {
  const Xxx* myXxx = dynamic_cast<const Xxx*>(objXxx);
  const Yyy* myYyy = dynamic_cast<const Yyy*>(objYyy);
//  Make sure that the dynamic casts did not return 0!
  return myXxx & myYyy & IsAssociated(myXxx,myYyy);
}
```

then you can build your `Lattice` like this:-

```
LatticeMaker maker;
Lattice* myLat = maker.CreateLattice(&mySetXxx,
                                     &mySetYyy,
                                     AssociateFunc ,
                                     "n:m");
```

as before, the last argument can also be "1:n" and "n:1".

> *Important: Any member of mySetXxx or mySetYyy that has no association will not appear in the lattice.*

For 1:n and n:1 relationships, the build is not very efficient; `LatticeMaker` has to loop over entries on the side that has single multiplicity and for each will, on average, have to loop half of the other side making a total of $n*m/2$ association tests. If n and m are large, there is a more efficient method that reduces this to $n+m$ but requires that each entry in the set that has multiple associations has a unique `Long_t` key and each entry in the set that has single associations can tell you the key it is associated with. Typical functions to get these keys might look like this:-

```
Long_t GetUniqueKey(const TObject* objXxx) {
  const Xxx* myXxx = dynamic_cast<const Xxx*>(objXxx);
//  If dynamic casts returns 0 return illegal key.
  return ( myXxx ) ? myXxx->GetUniqueKey() : -1;
}

Long_t GetAssociateKey(const TObject* objYyy) {
  const Yyy* myYyy = dynamic_cast<const Yyy*>(objYyy);
//  If dynamic casts returns 0 return a different illegal key.
  return ( myYyy ) ? myYyy->GetAssociateKey() : -2;
}
```

then you can build your `Lattice` like this-

```
LatticeMaker maker;
Lattice* myLat = maker.Create1:NLattice(&mySetXxx,
                                        &mySetYyy,
                                        GetUniqueKey ,
                                        GetAssociateKey);
```

in this case the `Lattice` has to be 1:n.

8.3 Background - The need for generic tools

Over the years HEP developed a very successful and simple architecture for Monte Carlo and Reconstruction programs. It consisted of dynamic public data structures describing the events and detector that were operated upon by a set of independent processors. As Fortran 77 does not support dynamic data structures, the HEP community developed memory managers for the purpose, the last of these being ZEBRA. These managers came complete with navigational tools to traverse links of the data structure. Powerful those these managers were they had two serious drawbacks:-

- No type safety: it was the user's responsibility to interpret the meaning, and even the type, of each word in a bank.
- No implementation hiding: any changes made by a bank's designer had to be reflected in changes to all user code that accessed the bank.

Consequently the design and use of banks were very strongly coupled which serious hampered program development. One of the driving forces behind OO methodology was the need to break this coupling between implementation and use. In an OO design the components (objects now instead of banks) control all access and coupling is limited to the public interface the object's support. However this responsibility includes navigation to related objects and this can be unsatisfactory for several reasons:-

- Missing facilities. When navigation a set of objects it can be helpful to processes them ordered or selected by some property. Either the objects themselves have to provide this facility or the user has to add code, and neither seems a satisfactory for what is clearly a generic problem.
- Asymmetric navigation. If there is a relationship between objects of class A and class B it is resonable to require navigation in either direction. However this may be difficult or even impossible within the objects themselves. If the B objects are derived from A objects then the A objects may be unaware of the Bs.
- Non-uniform navigation. Relationships between objects that are always read and written together can be by means of standard C++ pointers. When ROOT writes the objects out it takes care of the pointers so

that, when read back in, the pointers are still correct even though the objects pointed to will not normally be in the same place in memory. However for relationships between objects written to separate records, say between event hits and detector planes, pointers are not adequate and schemes that convert pointers into keys that remain valid under an output input cycle have to be employed. Then navigation is more complex and is dependent on the way the object structure is organised into I/O sub structures.

The purpose of the MINOS navigational tools is to address all of the above. The tools *do not* permanently record relationships; the objects retain that responsibility. Indeed they, or some other software element, has to take on new responsibility: the creation of transient (i.e. created as required during execution) navigation objects. However, once that is done the end users see uniform fully functional navigational interface which is independent of the actual implementation.

Much of the foundation, not to mention significant portions of code, come from the Lattice classes which implement arbitrary n:m relationships by Jon Thaler for the CLEO experiment.

8.4 Function and Form

8.4.1 Functionality

This section outlines the functionality of the tools.

Navigation involves two types of activity.

Navigation within a homogenous set While navigating within a homogeneous set (i.e. a set in which all objects are of the same type) the tools provide the following functionality:-

- Sorting according user supplied function of object state.
- Selection by:-
  - user supplied function.
  - value slice. For the current sorting function the user can supply upper and lower limits in the sorted value.

  - marking. The user can switch on/off a mask associated with each object in the set.

Navigation between pairs of homogenous sets The tools support 1:n navigation, and may eventually support full n:m navigation between pairs of sets.

The tools place no restrictions on the type of object set being navigated and are non-invasive i.e. no changes need to be made to the object's class. This confers an additional advantage as it implies multiple navigational objects operating concurrently. For example, this would mean that the same set could be simultaneously sorted with several different sort criteria.

Finally the tools are type safe, that is to say, although they work for any type of object the user always deals with objects of the correct type and does not have to perform a potentially dangerous cast of a pointer to the right type.

8.4.2 Form

See List of Classes of the Navigation PACKRAT For a description of the classes that implement the tools with the aid of a few simple UML (Unified Modelling Language) diagrams.

Chapter 9

Registry

9.1 Introduction

Last significant change: 2002/07/26

A Registry is a heterogenous and hierarchical collection similar to an STL map. It is a one to one association of a key to a value. The key is of the type `const char*` and the values can be of various types (thus “heterogenous”) and are accessed in type safe ways. The currently implemented types of values are the basic ones (`int, double, char, char*`) as well as `Registry` itself (thus “hierarchical”).

Registry is used to hold configuration information for various parts of the framework, including Algorithms (see sec. 3), Job Modules (sec. 1) and the Midad library (sec. 19).

Finally, Registry can go through ROOT I/O so it is possible to save them in the data stream for later read back.

9.2 Basics of Registry

This section walks through the life and times of a typical Registry.

9.2.1 Creation

Usually one creates a Registry on the stack as it can be passed by value. When copied, each stored value will also be copied. Here is the trivial construction:

```
Registry reg;
```

This default constructor will setup the Registry to allow any new key/value pairs to be added, but it will disallow the changing of values after the initial setting. In this case the Registry is considered to have its keys unlocked and its values locked.

You can create a Registry whose values are immediately changeable by passing in `false` at construction time:

```
Registry reg(false);
```

You can also manipulate and query the value lock with the `ValuesLocked()`, `LockValues()` and `UnLockValues()` methods. There are three similar method for the key lock.

9.2.2 Setting values

To set values, not surprisingly, you use the `Set()` method. There are a number of these methods, one for each type that Registry can hold. Here is an example showing the act of setting each type:

```
Registry reg;
reg.Set("my int",42);
reg.Set("my double",6.9);
reg.Set("my character",'x');
reg.Set("my string","Hello World!");
Registry reg2;
reg.Set("my Registry",reg2);
```

In the case of int and double, you must be be aware of the type. If you want a double which happens to be an integral value, make sure you use a decimal (ie, if you want to store "2" as a double you must use "2.0").

9.2.3 Getting values

Getting a value back from a Registry takes a bit more effort as there needs to be a way to communicate success or failure at finding the key and getting the correct type for the value. The safe way is to use the `Get()` methods like:

```
int x=0;
if (!reg.Get("my int",x)) {
    cerr << "Failed to get \"my int\"\n";
}
```

However, if you are absolutely sure that your value is both in the Registry and the correct type, you can use the less safe, but syntactically sweeter `GetTYPE()` form:

```
int x = reg.GetInt("my int");
```

If there is a failure, then “zero” is returned.

To make this slightly safer, you could test if the key exists before asking for it:

```
int x = 0;
if (reg.KeyExists("my int")) x = reg.GetInt("my int");
else cerr << "No such key: \"my int\"\n";
```

9.2.4 Iterating on the keys

You can iterate over all keys in a registry via the `RegistryKey` iterator and the `Key()` method. For example:

```
Registry reg;
//... fill reg ...

Registry::RegistryKey rk = reg.Key();
cerr << "dumping "<<reg.Size()<<" keys:\n";
while ( (const char* s = rk()) ) {
    cerr << "  " << s << "\n";
}
```

Using the `Set()` examples above to fill the Registry this code would print out something like:

```
dumping 5 keys:
    my int
    my double
    my character
    my string
    my Registry
```

9.2.5 Miscellaneous Methods

Here is a grab bag of other things one can do with a Registry.

```
Registry r1, r2;
// ...
// Assign one Registry to another:
r1 = r2;

// Merge one registry with another.  This is like assignment except
// any keys in r1 but not in r2 will not be removed:
r1.Merge(r2);

// Get the number of keys in a Registry
cout << "Reg1 has " << r1.Size() << " keys\n";

// Dump a Registry to some ostream:
cout << r1 << endl << r2 << endl;
```

Chapter 10

Raw Data and Rotorooter

Last significant change: 2002/10/14

10.1 General Principles

The `RawData` package primarily provides containers (classes derived from `RawDataBlock`) that encapsulate the raw data from the DAQ (data acquisition), DCS (detector control system) and beam monitoring systems. The basic unit is a *block* which consists of three required 32-bit words and a variable number of data words. A copy of the packed data, including the header words, is held by the base class. The `RawDataBlock` supplies some methods common to all blocks. In particular one can retrieve the size of the block and read-only access to the packed data. This encapsulation supports the principle that once a block is generated from the constructor it should not be modified. Based on the block id the data may be wrapped in a class deriving from `RawDataBlock` that specializes the access methods. These access methods encapsulate the knowledge neccessary for unpacking specific blocks. Additionally the package provides a block identification class, a electronics channel identifier class, a specialized record class and record headers, and support classes for elements unpacked out of specific blocks.

The `Rotorooter` package provides the interface for DAQ, DCS and beam monitoring systems which generate raw blocks in a C (or C-like) non-Object-Oriented environment to convert them into objects and write them out in the offline framework format. To minimize the coupling between the offline and online components the information is transmitted from the online to the `rotorooter` via sockets. The online code interfaces to a series of C functions; the `rotorooter` turns the collections of blocks, sent to it as a group, into objects and creates a offline *record* to hold them. The record can be persisted (Chapter 12). There are also C functions for the online code to call to open and close files and configure parameters of the `rotorooter`.

10.2 Details of the `RawDataBlock`

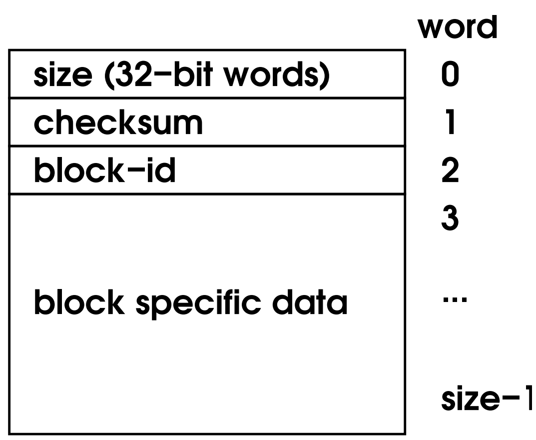

Figure 10.1: Basic `RawDataBlock` layout

The basic `RawDataBlock` provides basic read-only access to the unadulterated data. The reported size is a self inclusive count of 32-bit words; the raw data is treated as a array of 32-bit integers when moved between platforms of different architectures. All `RawDataBlocks` support the methods:

```
const Int_t* GetData() const; // as array of 32bit words
      Int_t  GetSize() const;
      Int_t  GetChecksum() const;
      Bool_t TestChecksum() const;
  RawBlockId GetBlockId() const;
```

Specialized blocks derive from `RawDataBlock` and supply the specific methods for unpacking a particular block. A block is primarily designated by the *major id* as decribed in Section 10.3 on `RawBlockId`.

The details of the block layout may evolve over time and thus the upacking may have to handle (generally minor) differences. Which version of the unpacking to use is encoded into each instance of a block by the *minor id* portion of the block id.

10.3 Details of the `RawBlockId`

The `RawBlockId` has fields that identify the minor version number, the major block number, online source, detector, and a compacted simulation flag.

Figure 10.2: `RawBlockId` layout
Compact simulation flag *csf*: (0=Data, 1=DaqFake, 2=MC, 3=Reroot)
Detector bits: CalDet, Far, Near
Data source *S*: 0=DAQ, 1=DCS

The major id numbers for DAQ and DCS (which includes beam monitoring in a logical sense) are independent. Major id values must be less than 2^{16} or in the range [0. . .65535]; minor version numbers can range from 0 to 255. Minor version numbers should be changed **every** time the layout of a block is modified or interpretation of any word or field in the block is changed. This is **critical** if the specialized `RawDataBlock` is to reliably be able to handle unpacking in all cases.

The connection between class type and major block id is made by a two lines of code in the derived class implementation file:

```
#include "RawData/RawBlockRegistry.h"
REGISTERRAWBLOCK(ClassName,MajorId,IsDCS);
```

This second line is a cpp macro that expands into some helper code. There are no quotes around the *ClassName* in the cpp macro. The *MajorId* must be an integer value that fits in 16 bits and *IsDCS* is a boolean value (0 or 1). When the class is compiled and the code put into the shared library, then loading the library automatically generates a static object that registers the connection between the *ClassName* and the major id and DCS flag. This connection is necessary in order for the `rotorooter` to convert flat memory space into the correct `RawDataBlock` object.

The following code will generate a list of all known blocks and their ids:

```
#include "RawData/RawBlockRegistry.h"
RawBlockRegistry& rbr = RawBlockRegistry::Instance();
cout << rbr;
```

10.4 Details of the `RawChannelId`

10.5 Operation of the `rotorooter`

10.6 Online code interface to the `rotorooter`

The interface code that online sources should use in communicating with the `rotorooter` is built into a static library (*libRotorooterRotoTalk.a*) as part of the offline code build procedure and can be found in *lib/$SRT_SUBDIR/* relative to the release found at *$SRT_PUBLIC_CONTEXT*. The header file of prototypes *rototalk.h* is found in *Rotorooter/RotoTalk/*.

Connections to the `rotorooter` can be openned and closed using:

```
int  roto_open_DCP_connection(const char* host);
int  roto_open_DCS_connection(const char* host);
int  roto_close_connection();
```

Note: There may be a need for an additional function, i.e. `roto_open_BM_connection(const char* host)`*, if the DCS and beam monitoring are going to have their individual receivers on the same host. Currently DCP is assigned port 9011 and DCS to port 9012.*

The DAQ (DCP) can have multiple files open simultaneously as the rotorooter knows how to route individual records to the appropriate file based on the run & subrun numbers. The DCS runs in a mode that is independent of the start and stopping of runs by the DAQ; DCS files are tagged by the time passed to the `rotorooter` and only one file can be open at any one time.

```
int  roto_open_daqfile(int detector, int run, int subrun);
int  roto_close_daqfile(int detector, int run, int subrun);
```

```
int  roto_open_dcsfile(int detector, int sec, int nanosec);
int  roto_close_dcsfile();
```

The *detector* value is all cases one of: Near=1, Far=2, CalDet=4.

Data is send to the `rotorooter` as a flat buffer and a **byte** count.

```
int  roto_send_record(const void* buffer, int nbytes);
```

The sent buffer can contain any number of contiguous packed blocks in the form described in Section 10.2. These will be collected into a single `RawRecord` and output to a `TTree` in the root file.

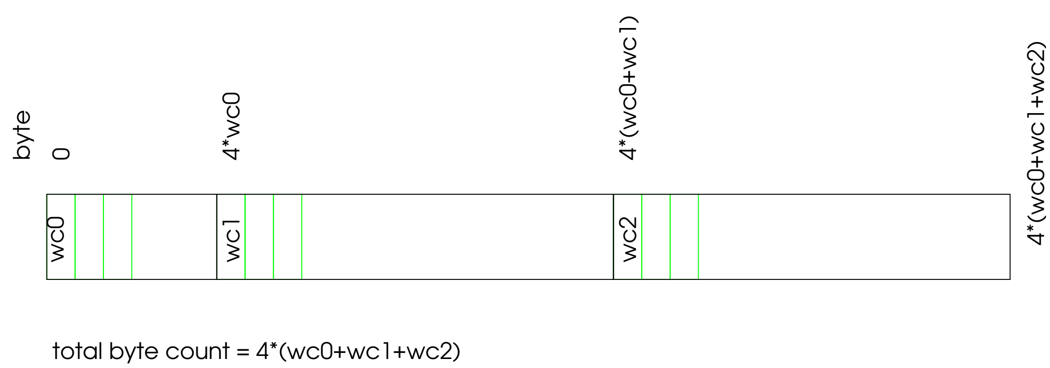

Figure 10.3: Record buffer layout
Example record of three blocks; each block leads with a 32-bit word of the inclusive word count.

Each sent buffer must contain one block that is used to create the record header: one of *RawDaqHeaderBlock* (0x200) *RawSnarlHeaderBlock* (0x101) for the DAQ, or *RawDcsHeaderBlock* (0x001) for DCS.

Finally, the `rotorooter` can be configured using:

```
int  roto_send_autosave_config(const char* stream, unsigned int nrec, unsigned int nsec);
int  roto_send_compress_config(const char* stream, int level);
int  roto_send_basketsize_config(const char* stream, int basketsize);

int  roto_request_shutdown(int priority);
```

Chapter 11

Calibrator

First version 07/2004 Nathaniel Tagg. This version describes the 'new calibrator' framework put forward to replace the older Digit-Calibrator/StripCalibrator classes and their attendant machinery.

11.1 Purpose

The Calibrator's purpose it to provide a framework for calibrating raw data. Generation of calibration constants is NOT part of it's use, although the database row classes used by the constants generators are defined in the Calibrator package.

A secondary use of the Calibrator is to provide decalibration constants for the detector simulation Monte-Carlo. Often the needs of the MC are slightly different, but in principle every calibration step should have a decalibration partner that can be used.

11.2 Structure

The `Calibrator` object is a singleton object that is used to handle all calibration calls. All calls for calibration are made to this object, and this object is the gateway for configuration and context changing.

The `Calibrator` holds pointers to "sub-calibrator" objects that do the actual work. The "sub-calibrators" all inherit from `CalScheme`, which holds prototypes for all the available calls. These sub-calibrators are responsible for holding any database tables, constants, and decision-making about the calibration steps.

The database tables held by the sub-calibrators will typically be `DbiResultPtr<>` objects, which hold rows of data. Each row typically corresponds to a channel (a strip end, a pixel spot, or a raw channel ID) and is a descendent of `DbiTableRow`. (For more information, please see the chapter on DatabaseInterface.) The row objects hold a few words of information each, and typically are little mini-calibrators.

The calibrator is invoked by several other pieces of the reconstruction framework. The time, photoelectron, drift, linearization, and strip-to-strip corrections are all done by AlgDigit, which use the Calibrator indirectly. The Calibrator handle is passed to the Plex when the AltStripEndIdList is made. The Plex calls the Calibrator with CalibStripEnd() for each strip end alternative. The Calibrator in turn calls it's own member functions: GetTime(), GetPhotoElectrons(), GetDriftCorrected(), GetLinearized() and GetStriptoStripCorrected() to apply the calibrations. These in turn call the sub-calibrators.

Attenuation correction (i.e. correction for position along the strip) and MIP calibration (i.e. conversion from pseudo-ADC to meaningful energy units) are done by Tracker, Shower-finder, and Event algorithms, which call Calibrator::GetSigMapped() and GetMIP(), which in turn call the Calibrator methods GetAttenCorrected() and GetMIP(), which call the sub-calibrators.

As an example, let us consider the application of Mapper data with the SigMappedCalScheme:

- The Tracker wants to do the map correction, so it calls `Calibrator::GetSigMapped()`.
- `Calibrator::GetSigMapped()` calls `Calibrator::GetAttenCorrected()` to perform the attenuation calibration for each end of the strip.
- `Calibrator::GetAttenCorrected()` uses the current sub-calibrator, currently SigMappedCalScheme, and calls SigMappedCalScheme::GetAttenCorrected().
- The SigMappedCalScheme converts the position it's been given into fibre distances. Then it looks up the strip end in it's table of `CalMapperFits` to find the right row.
- The row object `CalMapperFits` is called to estimate the attenuation that would be seen with the given length of fibre for that particular strip end and returns the value

- The return value is returned back up the call chain.

11.3 Configuring the Calibrator

The `Calibrator` object is a singleton object that can always be accessed with `Calibrator::Instance()`. There are two ways in which the Calibrator can be configured:

- The calibration Scheme for each type of calibration can be chosen.
- The Scheme itself can be configured.

11.3.1 Choosing Schemes

There are eight different sub-calibrators that can be set. To set sub-calibrators, simply put a line like this in your job control script:

```
Calibrator::Instance().Set(``PeCalibrator=PEGainCalScheme ''
                           ``TimeCalibrator=TimeCalScheme '');
```

The default calibrators are defined in Calibrator.cxx. Usually, the defaults will be what you want; other schemes are used to test new calibration algorithms or tables.

11.3.2 Turning Calibration Off

A general-purpose scheme has been provided that can be used in place of the defaults to provide very fast execution of code by using a set of constants instead of loading database tables. This Scheme is the `SimpleCalScheme` [1] To turn this on for all calibration modes, include the following line in your macro:

```
Calibrator::Instance().Set(``TimeCalibrator=SimpleCalScheme ''
                           ``PeCalibrator=SimpleCalScheme ''
                           ``VALinCalibrator=SimpleCalScheme ''
                           ``DriftCalibrator=SimpleCalScheme ''
```

[1] "Simple" here is used in the sense of 'idiot', but the code for this scheme is very uncomplicated as well.

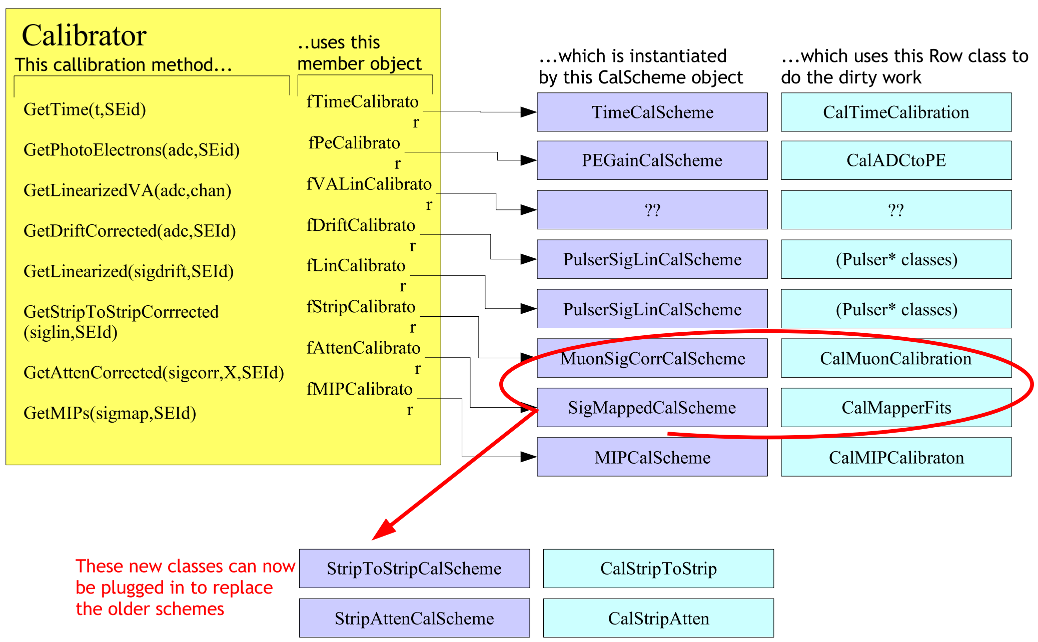

Figure 11.1: Diagram of the Calibrator

```
                              ``LinCalibrator=SimpleCalScheme ''
                              ``StripCalibrator=SimpleCalScheme ''
                              ``AttenCalibrator=SimpleCalScheme ''
                              ``MIPCalibrator=SimpleCalScheme '' );
```

This setting may be useful in cases where calibration is not important, e.g. event viewers, code testing, when the database is not available, etc.

11.3.3 Configuring the Sub-Calibrators

After the sub-calibrators have been chosen, you can configure them by getting a handle to them, then calling their `Set()` methods. You can get the handle to a sub-calibrator with the following functions:

```
Calibrator::Instance().GetTimeCalibrator()
Calibrator::Instance().GetPeCalibrator()
Calibrator::Instance().GetVALinCalibrator()
Calibrator::Instance().GetDriftCalibrator()
Calibrator::Instance().GetLinCalibrator()
Calibrator::Instance().GetStripCalibrator()
Calibrator::Instance().GetAttenCalibrator()
Calibrator::Instance().GetMIPCalibrator()
```

or, more simply:

```
Calibrator::TimeCalibrator()
Calibrator::PeCalibrator()
Calibrator::VALinCalibrator()
Calibrator::DriftCalibrator()
Calibrator::LinCalibrator()
Calibrator::StripCalibrator()
Calibrator::AttenCalibrator()
Calibrator::MIPCalibrator()
```

The individual parameters that may be set for each sub-calibrator can be found by looking at the configuration with `PrintConfig()` or by looking at the current configuration with `GetConfig()`. The parameters may be set with a variety of Set functions granted by `CfgPromptConfigurable`, for example:

```
Calibrator::Instance().Set(``PeCalibrator=SimpleCalScheme'');
Calibrator::Instance().PrintConfig();
// .. shows the current config...
Calibrator::PeCalibrator().Set(``defaultGainFD'',60.);    // ok
Calibrator::PeCalibrator().Set(``defaultGainFD=,60.'');   // ok
Calibrator::PeCalibrator().Set(``defaultGainFD'',60);     // will fail.. wrong type
```

The Set() commands are all safe, so any mistyped key or wrong value type will generate an error from CfgPromptConfigurable.

11.4 Configuration, Statistics, Errors

The current configuration of the Calibrator may be printed out at any time with `Calibrator::Instance().PrintConfig()`.

The current configuration may be saved by getting a Registry that contains the entire configuration for the Calibrator and all sub-calibrators by calling `GetCompleteConfiguration()`. This returned registry can later be applied by using `Calibrator::Instance().Set(registry)` to re-apply it. This use is intended to be used to log the configuration during batch processing.

By convention, all bad calibrations will be reported as Warnings, so turning the log level on the "Calib" stream up to Error will stop all non-critical calibration warnings.

Errors that occur during calibration are tracked. To see the current error status, you may call `Calibrator::PrintErrorStats()`, and they may be cleared with `ResetStatistics()`. The error printout contains a summary of the number of types of errors seen by each sub-calibrator, and a partial list of channels that saw problems, listed by frequency. For example:

```
        Calibrator      Calls ||    General|Miss Table|  Miss Row|  FP Error|Insuf.Data
           Unknown          0 ||          0 |         0 |         0 |         0 |         0
    TimeCalibrator      15680 ||          0 |         5 |         0 |         0 |         0
      PeCalibrator      15680 ||          0 |         0 |       680 |         0 |         0
   VALinCalibrator          0 ||          0 |         0 |         0 |         0 |         0
   DriftCalibrator      15680 ||          0 |        60 |         0 |         0 |         0
     LinCalibrator      15680 ||          0 |        10 |         0 |         0 |         0
   StripCalibrator      15680 ||          0 |         0 |         0 |         0 |         0
   AttenCalibrator       2404 ||          0 |         0 |         0 |         0 |         0
```

```
       MIPCalibrator       1202 ||        0 |         5 |         0 |         0 |         0
Channels with the most errors:
Errors:       40  [   Far|   1 Vt|  2|*E]
Errors:       40  [   Far|   1 Vt| 20|*W]
Errors:       40  [   Far|   1 Vt| 25|*E]
Errors:       40  [   Far|   1 Vt| 44|*W]
Errors:       40  [   Far|   1 Vt| 48|*E]
Errors:       40  [   Far|   1 Vt| 68|*W]
Errors:       40  [   Far|   1 Vt| 92|*W]
Errors:       40  [   Far|   1 Vt| 95|*E]
Errors:       40  [   Far|   1 Vt|116|*W]
Errors:       40  [   Far|   1 Vt|118|*E]
```

In this example, five events were run. The TimeCalibrator was unable to find any data at all for all 5 events, nor was the MIP calibrator The Drift and Lin calibrators also had problems (they look for more than 1 table each, so each was recorded).

The PeCalibrator found the tables it needed, but 680 strip ends didn't have database rows (even though the table existed). The rows corresponded to the strip ends listed below, all in plane 1, with 40 missed calibrations each.

> Note (NJT 07/04) This example is fudged; some of the error reporting in some modules isn't quite right, and I've never actually seen a missing row. However, this is a better example pedagogically.

11.5 Writing a New Sub-Calibrator

New calibration methods are easily created. Simply make a new class that inherits from CalScheme(), and implement the following functions:

Constructor The constructor should create a registry, and fill it with the default configuration for the scheme. The registry should be locked in with `InitializeConfig()`, which will call the `ConfigModified()` you define below. The constructor should *not* set any class members or initialize anything, although it should set pointers to zero and the like.

`ConfigModified()` This function should get the current configuration registry with GetConfig() and parse all the members in the configuration. If a needed value is not in the configuration, it should complain. If something is changed that could affect the database queries (i.e. changing which Task is being used) then the function should call `this->Reset(GetContext());` to force a reload of the DB tables.

`DoReset()` This function is called by Reset() whenever there is a change of the current context. It should perform a `NewQuery()` on each database table it holds, or do other things that might change with detector or time.

`PrintConfig)` This function should print out the current configuration of your scheme. This is optional but recommended.

Calibration calls One or more of the calibration calls from CalScheme should be implemented, e.g. GetPhotoElectrons() if your scheme is a p.e. calibration.

Decalibration calls For every calibration call, you should also implement the complimentary decalibration call.

The CalScheme can implement multiple different sub-calibrator types in the same class (like SimpleCalScheme) but be warned that more than one instance of the class will be created by the Calibrator, and they may be calibrated differently.

Warnings and errors should be handled correctly. All message service strings should go to `MSG(''Calib'',Msg::kWarning)` for bad calibrations or missing tables. The use of the MAXMSG macro is advised for most of these cases.

In addition, an error-counting service is in place. If you fail to find any rows in a table during DoReset(), you should make a call to `IncrementErrors(k<type>Calibrator,kMissingTable)`. If you can't find a row in a table that has rows, you should make a call to `IncrementErrors(k<type>Calibrator,kMissingRow,<channel>)`, where ¡channel¿ is a PlexStripEndId, an LedId, a RawChannelId, or a PlexPixelSpotId. There are also kFPE to track floating-point errors, kDataInsuffcient to track places where there is insufficient calibration data to make a good calibration, and kGeneralErr as a catch-all category.

11.6 Migrating from the Old DigitCalibrator and StripCalibrator

The old calibration system used a simpler configuration interface, but it was typically used by the reconstruction Algorithms, rather than directly by the user. If you have old scripts that change the calibration modes from the defaults, then you will need to change them to the new configuration method.

11.6.1 Calls Via AlgConfig

Many macros will use the AlgDigit (via the DigitListModule) to change the digit calibration modes. A few also have lines to change the strip calibration modes (attenaution and MIP correction) via calls to the tracker or showerer. Examples:

```
jc->Path(``path'').Mod(``DigitListModule'').Set(``CalPeMode=0'');
jc->Path(``path'').Mod(``TrackSRListModule'').Set(``CalMipMode=1'');
```

These configuration methods are no longer used. Use the conversion table below to interpret these calls. In general, setting a mode to zero implied turning off that calibration. In the new system, that means changing the calibrator to the 'Simple' calibrator. Otherwise, the integer was a mode flag which could imply lots of different behaviours.

Old call	Meaning	New call `Calibrator::Instance().`
CalPeMode = 0	Turns off PE cal	`Set(``PeCalibrator=SimpleCalScheme'');`
CalPeMode = 1	Turns on PE cal	(default)
CalSigLinMode=0	Turns off LI cal	`Set(``DriftCalibrator=SimpleCalScheme``` `` ``LinCalibrator=SimpleCalScheme''); ``
CalSigLinMode=n	Sets mods for LI cal	`GetDriftCalibrator().Set(``CalMode=n'');`
CalSigCorrMode=0	Turns off muon cal	`Set(``StripCalibrator=SimpleCalScheme``);`
CalSigCorrMode=1	Default muon cal	(default)
CalSigCorrMode=2	Turn on temp correction	`GetStripCalibrator().Set(``TemperatureCorrection=on'');`
CalSigCorrMode=3	Use PS muon calibration	`GetStripCalibrator().Set(``MuonSource=PSMuons'');`
CalSigCorrMode=4	Use alternate PS muons	`GetStripCalibrator().Set(``MuonSource=AltPS'');`
CalTimeMode=0	Turns off time calibration	`Set(``TimeCalibrator=SimpleCalScheme``);`
CalTimeMode=1	Turns on time calibration	(default)
CalSigMapMode=n	Mode for Atten correction	`GetAttenCalibrator().Set(``CalMode=n'');`
CalMipMode=0	Turn off MIP calibration	`Set(``MIPCalibrator=SimpleCalScheme``);`
CalMipMode=1	Turn on MIP calibration	(default)

11.6.2 Direct Calls

You may have lines such as these in your macro:

```
DigitCalibrator::Instance().SetCalModes( 1, 1, 1, 1 );
StripCalibrator::Instance().SetCalModes( 1, 1 );
```

These perform the same actions as above, but without the AlgConfig mechanism. These calls can be interpeted as in the table above:

```
DigitCalibrator::Instance().SetCalModes( 1, // CalPeMode
                                         1, // CalSigLinMode=1
                                         1, // CalSigCorrMode=1
                                         1  // CalTimeMode=1
                                        );
StripCalibrator::Instance().SetCalModes( 1, // CalSigMapMode=1
                                         1  // CalMipMode=1
                                       );
```

Chapter 12

Data Model and I/O

Last significant change: 2004/04/18

12.1 Introduction

The purpose of this chapter is to describe the model that is used as the basis for MINOS persistable records and the I/O related management of those records.

12.2 Persistency and ROOT

Persistence describes the process of making objects permanent beyond their present application for reuse by some later application, such as when a C++ object is written to a file.

The problem of persisting C++ objects to file and retrieving those objects back in memory is not trivial. Objects may have complex structure (hierarchy of inheritance, and data members that are objects or pointers to objects) and C++ has weak built-in support for object persistency.

The solution to this problem is to supplement C++ with a framework for use of its I/O facilities. In the case of MINOS, ROOT provides this framework. In particular, ROOT provides the following persistency tools:

- Automatic **Streaming mechanism** for user-defined classes. The Streaming mechanism is used to convert object data to a stream of bytes (data buffer) to be stored on file and vice versa.

- **File management** (TFile) to organize storage of object data buffers in a file with Keys to facilitate random and sequential access of objects according to a user provided Key Name.
- **Data structures** (TTree) to coordinate sets of objects. The ROOT TTree is a generalized ntuple that can be used to store objects as well as basic data types. Synchronized and partial read of members of a set are supported.
- **Remote access** to files (TNetFile along with a ROOT supplied server daemon) to support a "Distributed Database" environment in which data may be distributed over multiple files (some remote).
- **Automatic Class Schema Evolution** such that a class definition can change, for example data members may be added and removed to produce a new version of the class, but it is still possible to read objects of an older version of this class from a data file.
- **Persistable Pointers** such that the relationship between objects written out to different trees, possibly even in different files, can be reconstructed on input.

12.3 Data Model

The data model is defined by the classes that are used to store the data, the data structures that are used to store the data objects, and the organizational scheme that is used to store the data in the data structures and the data structures in the files.

Two considerations that have gone into defining a MINOS data model are:

- A desire to make maximal use of the persistency tools provided by ROOT. For example the ROOT TTree data structure allows partial input of data members of a set (analogous to reading only a selected subset of columns of an ntuple for a particular analysis). To maximize the use of this aspect of the ROOT TTree data structure it is desirable to store frequently accessed smaller data objects on a separate branch (column) from larger data objects.

- A desire to avoid overspecializing or hardwiring record class definitions in the input and output stream management code.

12.3.1 Definitions

The MINOS data model makes use of the following terms:

- A **Record** consists of a single header and a collection of data blocks, and constitutes a single entry in a ROOT TTree. To take advantage of the partial input feature of the TTree, the header is stored on a separate branch from the collection of data blocks.

- A **Stream** consists of a tree in a single file or possibly sequence of multiple files. A Stream name may be the same as a Tree name, but need not be. For example, Stream "Cand1" may refer to TTree "Cand" in File1, while Stream "Cand2" refers to TTree "Cand" in File2.

12.3.2 More on Records

To be supplied.

12.4 Input Stream Management

Figure 12.1 illustrates the organization of MINOS records in ROOT TTrees and the organization of ROOT TTrees in data files. The diagram also illustrates how the records from the different streams are synchronized and loaded into the MomNavigator (Mom) according to each record's VldContext by the input stream manager.

In particular, note that:

- Multiple records with coincident VldContext are loaded in the same Mom object as a single record set.

- Some trees, for example the DaqMonitor tree, may contain multiple records of identical VldContext that will also be loaded into the same Mom object as part of a single record set.

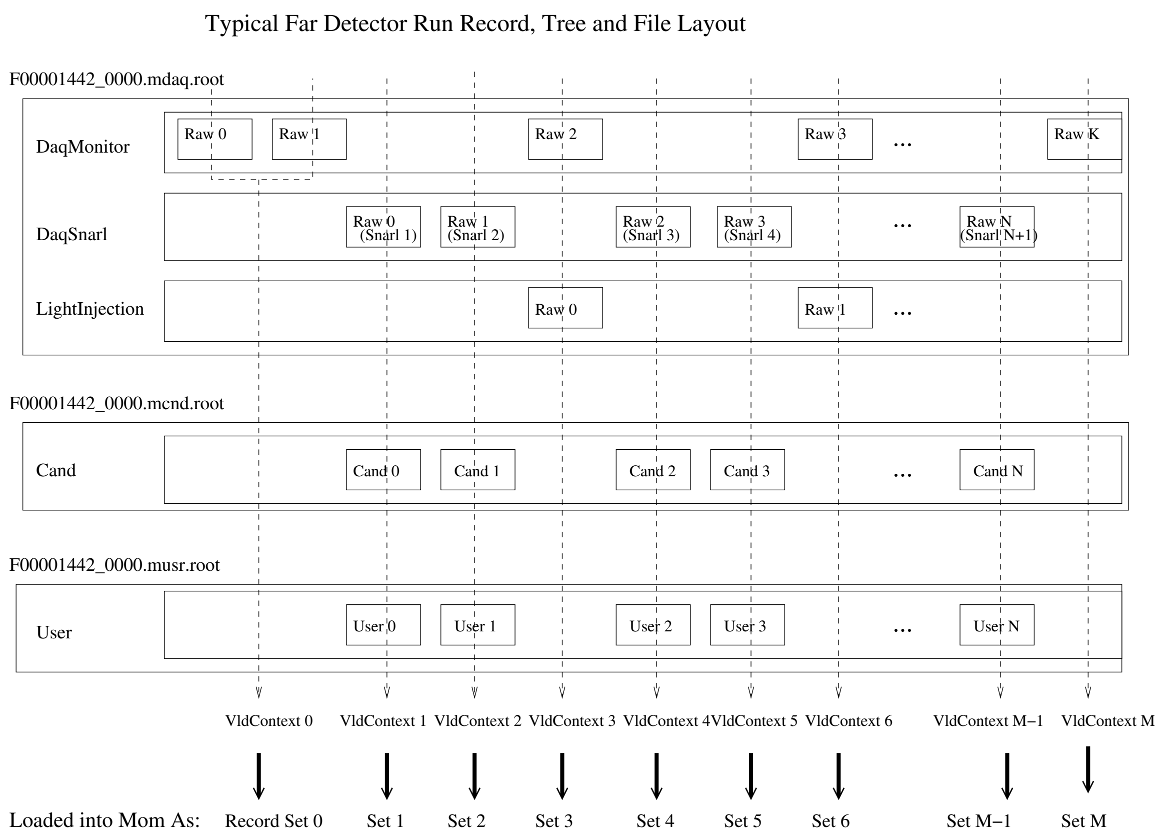

Figure 12.1: Diagram illustrating layout of MINOS records, trees and files for a typical far detector run.

12.4.1 Specifying input files

Specifying input files is done using the JobCInput::AddFile method which has the format:

```
void AddFile(const char* filename,const char* streamlist=``*'',int at = -1);
```

Specifying the AddFile method without the optional streamname argument will apply the specified file to all active input data streams. The user may optionally specify separate data files lists for separate data input streams. For example:

```
JobC j;
...
j.Input.AddFile(``testntp.root'',''NtpCand''); // to serve ntuple records
j.Input.AddFile(``F00005903_0000.cand.root'',''Cand''); // to serve candrecords
```

The AddFile method also supports wildcards in the input data file name, for example,

```
j.Input.AddFile(``/mydir/F00059*.mdaq.root'',''DaqSnarl'');
j.Input.AddFile(``/mydir/F00059*.cand.root'',''Cand'');
```

The method:

```
j.Input.DefineStream(``streamname'',''treename'');
```

allows the user to specify the data tree to attach to a streamname. This can be useful if reading two streams from 2 sets of input files but both streams serve data from the same tree, for example:

```
j.Input.DefineStream(``Rel8Cand'',''Cand'');
j.Input.DefineStream(``Rel9Cand'',''Cand'');
j.Input.Set(``Streams = Rel8Cand,Rel9Cand'');
j.Input.AddFile(``/mydir/F00059*.R0.8.0.root'',''Rel8Cand'');
j.Input.AddFile(``/mydir/F00059*.R0.9.0.root'',''Rel9Cand'');
```

Finally, JobCInput::List() can be used to view the streams attached to a given stream, for example,

```
j.List();  // to view all files attached to all streams
j.List(``DaqSnarl'');  // to view all files attached to the DaqSnarl stream
```

12.4.2 Identifying the Source of a Data Record

All records are stamped on input with the streamname, filename, treename, and tree index from which they came. (This information is stamped on the record whether it is received through the dispatcher or through offline file access.) The data is stored in a temporary (not persisted) Registry accessible through the RecRecord base class. The following code shows how to access this data given a pointer "mom" to a MomNavigator object:

```
// Iterate over all objects contained in Mom
TIter mitr = mom -> FragmentIter();
TObject* object;
 while ( (object = mitr.Next()) ) {
     RecRecord* record = dynamic_cast<RecRecord*>(object);
     if ( record ) {
        char* streamname = 0;
        if ( record -> GetTempTags().Get("stream",streamname) )
           cout << "stream " << streamname << endl;
        char* treename = 0;
        if ( record -> GetTempTags().Get("tree",treename) )
          cout << "tree " << treename << endl;
        int treeindex = 0;
        if ( record -> GetTempTags().Get("index",treeindex) )
          cout << "index " << treeindex << endl;
        char* filename = 0;
        if ( record -> GetTempTags().Get("file",filename))
        cout << " file " << filename << endl;
     }
  }
```

12.5 Output Stream Management

Data is written to output stream(s) in output file(s) via the Output job module provided by the framework. The Output job module is like any other user job module in that it may be inserted at any point in the user's job path. An example:

```
loon[0] JobC j;
```

```
loon[1] j.Path.Create(``Demo'',''DigitListModule::Get ``''Output::Put``);
loon[2] j.Path(``Demo'').Mod(``Output'').Set(``Streams=DaqSnarl,Cand'');
loon[3] j.Path(``Demo'').Mod(``Output'').Set(``FileName=recons.root'');
```

In this example, the user has specified that the Put method of the Output module be invoked at the end of the Demo job path. The output module has then been configured to write data to streams DaqSnarl and Cand, and to store these data streams in output file recons.root.

To view the list of output module configurable parameters and their default values, use the Report() method:

```
loon[0] JobC j;
loon[1] j.Path.Create(``Demo'',''DigitListModule::Get ``''Output::Put``);
loon[2] j.Path(``Demo'').Mod(``Output'').Report();
Output configured with: ['Output.config.default'
                          'AccessMode'=(string)'Recreate'
                          'AutoSaveBytes'=(int)10000000
                          'AutoSaveInt'=(int)0
                          'AutoSaveTime'=(int)0
                          'DefaultFileName'=(string)'out.root'
                          'FileName'=(string)''
          'Streams'=(string)'DaqSnarl,DaqMonitor,LightInjection,Cand,SimSnarl']
```

The AccessMode, Streams, and FileName configuration parameters are described in the following section. The AutoSave parameters are described in Section 12.5.2.

12.5.1 Specifying output streams and files

User-defined output streams

The user may design their own records, and these may be persisted through the job framework output module. The command sequence to do this is:

```
j.Path("Demo").Mod("Output").Cmd("DefineStream MyStreamName MyClassName");
j.Path("Demo").Mod("Output").Set("Streams=DaqSnarl,Cand,MyStreamName,...");
j.Path("Demo").Mod("Output").Set("FileName=outputfile.root");
```

where MyClassName is the name of the user's record class, and MyStreamName is the name of the user's record stream. (The convention is to define a record class with a name of the form XxxRecord, and the stream name is then by convention defined to be Xxx, but this is not enforced.)

The user then needs to create a job module to create the objects of MyClassName, and push these into Mom. The framework will automatically create a root TTree of name "MyStreamName" to hold records of type "MyClassName" with splitlevel 99. The output module will search Mom for objects of the user's specified type and persist them to this output tree.

DefineStream has optional arguments to further configure the user's output stream. The full interface is:

```
j.Path(``Demo'').Mod(``Output'').Cmd(
``DefineStream streamName className userName inputStreamName splitLevel'');
```

where userName (TNamed name) and inputStreamName can be used to further refine the object the output module should search for in Mom. splitLevel may be used to adjust the split level with which the root TTree is created, where the range is 0 to 99. The default of 99 splits the tree branch structure to the finest possible level. A setting of 0 means all object data will be stored on a single root tree branch.

Writing data to multiple output files

Writing data to two or more output files requires creating separate job paths to support the use of multiple output modules. Each output module can then be configured differently with regard to its output file. The separate job paths can be appended to create one job path.

12.5.2 Salvaging output data from crashed jobs

It's possible to switch on tree "autosaving" in the job control script, so that output trees are saved at regular intervals. If a crash occurs during the course of a job, the data in the output trees is then salvagable up to the last autosave. The IoOutputModule configurable parameters allow the user to specify saving the trees at regular intervals determined by the number of tree entries, the amount of time that has passed since the last save, and/or the number of bytes that have been filled in the tree. For example, the user may specify:

```
jc.Path("Demo").Mod("Output").Set("AutoSaveInt = N");  // save every N entries
```

```
and/or:
jc.Path("Demo").Mod("Output").Set("AutoSaveTime = N"); // save every N seconds
and/or:
jc.Path("Demo").Mod("Output").Set("AutoSaveBytes = N");// save every N filled bytes
```

By default, only AutoSaveBytes is activated at 10 Mbyte intervals. A caution is that setting the autosave intervals so that autosaves occur at a high frequency will cause a degradation in performance. For example, saving on the order of every 100 to 1000 entries is reasonable, but every 1 entry is not.

When reading an aborted file with the autosaved trees in a subsequent root session, the file can be opened in "UPDATE" mode and the recovery is in principle automatic, although sometimes the TFile::Recover method needs to be explicitly invoked. For example:

```
root[0] TFile* file = new TFile(‘‘abortedfile.root’’,’’UPDATE’’);
root[1] file -> Recover();
```

will recover the data stored in each tree up to last the autosave call before the crash. The recovered information will be stored in the file, so that the next open of the file will not need to go through the recovery process.

12.6 Miscellanea

12.6.1 Rules for writing a Persistable Class

Since ROOT forms the framework for MINOS persistency, it defines the rules that the user must follow when defining a class to be persisted to file. What follows is an abbreviated list of rules to remember when writing a persistable class. A much more detailed description of defining a class can be found in the ROOT documentation.

- The class header file should include a line invoking the ROOT provided

  ```
  ClassDef(ClassName,ClassVersionID) // Class Title
  ```

  macro. An illustration of use is shown in an excerpt from RecoBase/CandStripList.h:

  ```
  class CandStripList : public TObject {
    ...
    ClassDef(CandStripList,1)  //  Concrete Event Candidate Class
  };
  ```

The ClassDef statement should be included just before the closing brace. The class version number should be incremented whenever the data members of the class change, for example when the data member order is changed, a new member is added or an old member is removed, or the type of a data member changes. It should also be incremented if the list of base classes that the defined class inherits from changes.

Note that setting the version number in the ClassDef to 0 means that the objects of this class are not to be persisted.

- The class implementation file should invoke the ClassImp(ClassName) macro. An example can be found in RecoBase/CandStripList.cxx:

```
...
#include "MessageService/MsgService.h"

ClassImp(CandStripList)
...
```

- Activate the automatic schema evolution capabilities provided with ROOT by specifying the '+' option after the classname in the LinkDef.h file, e.g.:

```
#pragma link C++ class CandStripList+;
```

This should be done even with the first version of the class. Don't wait until the second version of the class to activate this or extra work will have to be done to read objects of older versions of this class stored in old data files!

- If the class is to be the top level record class persisted to a ROOT TTree, it should inherit from the Record/RecRecordImp class and have a header that inherits from the Record/RecHeader class. This is because the input stream manager uses the VldContext stored in the base header class for sorting the records from the different trees on input.

Chapter 13

Ntuples

Last significant change: 2004/07/06

13.1 Introduction

An Ntuple stores data of basic data types in an easy to access format in which data variables may be manipulated, plotted and fitted. An example of an application of an ntuple is to store frequently accessed reconstruction variables, which taken together summarize the results of a reconstruction job.

The purpose of this chapter is to describe how to create and use ntuples within the MINOS framework.

13.2 Framework supported Ntuples

The minos software framework supports the production of 3 standard ntuples:

- Standard reconstruction ntuple (NtpSR)
- Monte Carlo truth ntuple (NtpMC)
- Monte Carlo truth helper ntuple (NtpTH)

The ntuples are produced through storage of NtpXXRecord objects in a root TTree of name NtpXX. The best place to look for definition of data members stored in the standard ntuples is to look at the NtpXXRecord.h and helper class header files. These files are stored in the following packages:

- CandNtupleSR
- MCNtuple
- TruthHelperNtuple

for the NtpSR, NtpMC and NtpTH ntuples respectively.

For example, the definition of the cosmic ray data members stored in the NtpSR tree can be found in the CandNtupleSR/NtpSRCosmicRay.h file as:

```
Float_t zenith;  // zenith angle (degrees)
Float_t azimuth; // azimuthal angle measured easterly from north (degrees)
Float_t ra;           // right ascension (degrees)
Float_t rahourangle;  // right ascension (hours)
Float_t dec;          // declination (degrees)
Double_t juliandate;  // julian date (days since Jan 1, 4713 BC 12h UT)
Float_t locsiderialtime; // local sidereal time (hours)
```

13.3 Using ntuples

Ntuple trees may be analyzed four ways:

- In a bare root session, making use of a root TBrowser.
- In a bare root session, making use of MakeClass.
- In a loon session, using a script.
- Within the framework, treating the ntuple trees as input data streams.

The purpose of the following sections is to introduce each of the 4 approaches to analyzing ntuple data.

13.3.1 Analyzing ntuple data in a bare root session with TBrowser,TTree::Draw, etc.

Use of the TruthHelperNtuple

The truth helper data stored in the NtpTH tree can be used to move from the reconstructed entity (slice,shower,track) to the best match truth entries in the NtpMC tree. For example, for the first reconstructed slice in a given snarl, thslc[0].neumc will give the index of the best match initiating neutrino in the NtpMC mc array, and thslc[0].neustdhep will give the index of the best match neutrino in the NtpMC stdhep array.

Taking the example a step further, the mc.iaction associated with the first slice of a given snarl is mc[thslc[0].neumc].iaction. To select slices which satisfy the selection of best match neutrino has mc.iaction equal to 0 (neutral current), use the selection string "mc[thslc.neumc].iaction==0". The entries in thslc are implicitly looped over.

To Draw members of the slc array which satisfies this selection cut, do this:

```
root[0] TFile* file = new TFile("overlayfile.root","READ");
root[1] TTree* ntpsr = (TTree*)(file -> Get("NtpSR"));
root[2] TTree* ntpmc = (TTree*)(file -> Get("NtpMC"));
root[3] TTree* ntpth = (TTree*)(file -> Get("NtpTH"));
root[4] ntpmc -> AddFriend(ntpsr);
root[5] ntpmc -> AddFriend(ntpth);
root[6] ntpmc -> Draw("slc.index","mc[thslc.neumc].iaction==0");
```

Since slc and thslc share a common set of indices (i.e. there's a 1:1 relationship between them), this command will plot the slc indices corresponding to those with mc.iaction==0.

13.3.2 Analyzing ntuple data in a bare root session with MakeClass

For a user who wishes to analyze an ntuple within a bare root session without loading minos specific libraries, the TTree::MakeClass method can be a valuable tool. An example of how to do this is given here:

```
{
// usage: root .x sriptname.C
// The output is two files NtpSR.h and NtpSR.C

 f = new TFile("F00017566_0000.sntp.root","READ");
 t = (TTree*)(f->Get("NtpSR"));
 t -> MakeClass();
}
```

The execution of this example script results in two files: NtpSR.h and NtpSR.C. NtpSR.h is a header file for a new class NtpSR which reproduces the structure of the data members stored on the NtpSR tree. NtpSR.C implements the Loop method of the NtpSR class as a demonstration of how one might loop over the entries in the tree using this NtpSR class. The user may customize the Loop method implementation in NtpSR.C to suit his or her needs.

The following examples illustrate the use of the NtpSR.C script.

- This example invokes the Loop method to run over all entries in the TTree/TChain with which the NtpSR.h file was originally produced.

  ```
  root[0].L NtpSR.C
  root[1] NtpSR ntpsr; // instantiate the NtpSR object using its default ctor
  root[2] ntpsr.Loop();
  ```

- This example invokes the Loop method to run over all entries in the a newly created TChain:

  ```
  root[0]TChain* chain = new TChain(``NtpSR'');
  root[1]chain -> Add(``ntuple1.root'');
  root[2]chain -> Add(``ntuple2.root'');
  root[3].L NtpSR.C
  root[4]NtpSR ntpsr(chain);  //instantiate the NtpSR object to use the new chain
  root[5]ntpsr.Loop();
  ```

13.3.3 Analyzing ntuple data in a loon session with a script

An example of how to analyze ntuple data with a root script is given here.

```
{
// usage: loon sriptname.C
//
 gSystem -> Load("libCandNtupleSR.so");

 f = new TFile("F00017566_0000.sntp.root","READ");
 t = (TTree*)(f->Get("NtpSR"));
```

```
 NtpSRRecord* record = 0;
 t -> SetBranchAddress("NtpSRRecord",&record);

 // Loop over all entries in tree
 for ( int ient = 0; ient < t -> GetEntries(); ient++ ) {
   int nbytes = t->GetEntry(ient);
   cout << "Read entry, nbytes = " << ient << ‘‘,’’ << nbytes << endl;

   // Example of Accessing header data
   cout << "snarl " << (record -> GetHeader()).GetSnarl() << endl;
   // or
   RecCandHeader& header = record -> GetHeader();
   cout << "snarl " << header.GetSnarl() << endl;
   cout << "run " << header.GetRun() << endl;

   // Example of accessing event summary data
   cout << "crhdr.azimuth " << record -> crhdr.azimuth << endl;

   // Example of accessing TClonesArray (stp,trk,etc.) data
   TClonesArray& striparray = *(record->stp);
   for ( int istp = 0; istp < striparray.GetEntries(); istp++ ) {
     NtpSRStrip* ntpstrip = dynamic_cast<NtpSRStrip*>striparray[istp];
     cout << "strip index " << ntpstrip->index << endl;
   }
   record -> Clear();  // release allocated memory, record will be reused
 }

 delete record; record = 0;

}
```

13.3.4 Analyzing data from multiple ntuple trees in parallel

Analyzing data from two or more ntuples in parallel in a root session requires making use of the TTree::AddFriend mechanism.

In the simplest use case in which the ntuple trees are aligned such that there is a 1:1 relationship between entries in the separate trees, the TTree::AddFriend method may be used directly. The following example illustrates how to do this:

```
{
  TFile* file = new TFile("ntuple.root","READ");
  TTree* mctree = (TTree*)(file -> Get("NtpMC"));
  TTree* srtree = (TTree*)(file -> Get("NtpSR"));

  // Make the srtree a friend of the mctree
  mctree->AddFriend(srtree);

  // It's now possible to draw data members of either tree
  // using the mctree pointer, for example:
  // mctree->Draw("evt.vtx.z");  // branch on NtpSR tree
  // mctree->Draw("mc.vtxz");   // branch on NtpMC tree
  mctree->Draw("evt.vtx.z:mc.vtxz","","box"); // plotting SR vtx vs MC vtx
}
```

In some cases, however, the relationship between entries in different ntuple trees is not 1:1. An example of this is the writing of non-triggering truth records to the NtpMC tree, which do not result in the writing of a corresponding entry to the NtpSR tree. In this case, the 1:1 relationship between tree entries is lost.

This loss of 1:1 relationship is not a problem within our framework, which will use the VldContext stored with the records to resynchronize the entries across the two trees when reading the NtpMC and NtpSR streams in during a subsequent loon job. But the loss of 1:1 relationship is a problem when analyzing the trees in a root session using TTree::AddFriend, since this method only matches records across two or more trees by tree index.

A future version of root should support the feature of using TTree::AddFriend to synchronize by “key value” (e.g. VldContext). In the meantime, a script is provided by the MINOS framework that can be used to restore the functionality of TTree::AddFriend with non-aligned trees. The script synchronizes trees by ”key value”, so that they can be used with the TTree::AddFriend mechanism. The script, and a second script to demonstrate its use, can be found in Record/macros as AddFriendByKey.C and AlignTreesByKey.C.

Chapter 14

Leak Checker

Last significant change: 2001/09/17

14.1 Introduction

OO programs are dynamic; the objects they consist of are created and destroyed throughout the course of program execution. Objects require memory to hold their data members. The memory for objects created at execution time comes from the stack or the heap. When objects are destroyed it is essential that the memory that they occupied is recovered. This is automatic for stack based objects, but is left to the application code for heap based objects. Failing to destroy objects correctly by using the delete operator results in memory clogged with dead objects, a phenomenon referred to as a *memory leak*

The LeakChecker package is a very simple leak checker facility developed as part of the framework to provide the first level of defense against memory leaks. It can be used to allow basic leak checking to be built into package validation.

> *LeakChecker is not a substitute for a full function leak checker which should still be applied to complete programs.*

To use the checker, probes (C++ macro calls) are added to the code at each object creation and destruction. By default these macros generate nothing and so can safely be a permanent addition to the code. Activation is controlled by a preprocessor option which then generates calls to record object creation and destruction with a LeaLeakChecker object. This object can be used by validation code to query the number of objects created and the number currently active. The information can be supplied globally or for a specific class.

14.2 Inserting Leak Checker probes into code

The best place to insert the probes are into the class constructors and destructors, but this is not possible for external code such as ROOT. In this case the probes have to be added where the new and delete operators are used on them.

14.2.1 Adding probes for MINOS classes

The leak checker macros are defined in:-

```
#include "LeakChecker/Lea.h"
```

The macro LEA_CTOR should be inserted into every constructor and LEA_DTOR into the destructor. These macros deduce the class name from the file name so file names must follow the standard naming convention i.e. the class name followed by .cxx or .h, See below if this is not the case e.g. defining multiple classes in a single file.

> *It is essential that the compiler not be allowed to generate default and copy constructors as calls to these will not be recorded.*

The code for a dummy class supplying both default and copy constructors would look like this:-

```
#include "LeakChecker/Lea.h"

class MyClass {
public:
  MyClass() { LEA_CTOR; }
  MyClass(const MyClass& that) { LEA_CTOR; *this = that }
 ~MyClass() { LEA_DTOR;  }
...

}
```

If you don't want a copy constructor then the trick is simply to declare, but not define, a private one e.g:-

```
private:
  MyClass(const MyClass& that);
}
```

being private nobody else can call it so you don't have to define it.

If your class is not defined in a file with the standard naming convention then you will have to use extended forms LEA_CTOR_NM and LEA_DTOR_NM (where NM = named). These macros take two arguments: the name of the class and the address of the object. Using these macros the above example becomes:-

```
#include "LeakChecker/Lea.h"

class MyClass {
public:
  MyClass() { LEA_CTOR_NM("MyClass",this); }
  MyClass(const MyClass& that)
            { LEA_CTOR_NM("MyClass",this); *this = that }
 ~MyClass() { LEA_DTOR_NM("MyClass",this);  }
...

}
```

14.2.2 Adding probes for foreign classes

The extended macro forms LEA_CTOR_NM and LEA_DTOR_NM can be used with each new and delete operation to record construction and destruction of objects for classes where the modification of the source is not an option. Of course you should not do this for MINOS class objects that have probes built into their constructors and destructors, although if you do, this only means double counting; it won't mask a leak or spuriously indicate one.

Here is sample code recording object creation and destruction:-

```
#include "LeakChecker/Lea.h"

  TheirClass* myObj = new TheirClass(...);
  LEA_CTOR_NM("TheirClass",myObj);

  ...
```

```
  delete myObj;
  LEA_DTOR_NM("TheirClass",myObj);
  myObj = 0;
```

This records heap objects. In principle the same technique could also be used for stack based objects:-

```
#include "LeakChecker/Lea.h"

  {
    TheirClass myObj(...);
    LEA_CTOR_NM("TheirClass",&myObj);

    ...

    LEA_DTOR_NM("TheirClass",&myObj);
  }
```

The DTOR call has to be placed at the point where the object goes out of scope. Not only would this be easy to overlook but will fail if control does not fall through to the end of the compound statement. Further, stack based objects are not a source of leaks and all that is lost by not recording them is that they will not be included in the total creation statistics. For this reason you are advised:-

> *Do not use the macros to record creation and destruction of stack objects.*

14.3 Activating the probes

As explained in the introduction, by default the macro probes are disabled; producing no code at all. Consequently they can be permanent additions to the source. To activate them the code has to be compiled with the preprocessor option -DLEAK_CHECKER. This option can be selected when using the standard Makefiles by defining the environmental variable ENV_CXXFLAGS. The following shows how to activate the probes in the MyPackage package:-

```
cd MyPackage
setenv ENV_CXXFLAGS  -DLEAK_CHECKER
gmake clean lib
unsetenv ENV_CXXFLAGS
```

if not using csh, substitute the equivalent commands for setenv and unsetenv.

In this way leak checking can be activated package by package, or even class by class. Note however, probes used for foreign classes can give spurious results if not all new and delete probes are activated. If an activated package gives objects to another unactivated package to delete this will appear as a memory leak. Conversely the inverse will appear as a negative leak. These problems can be avoided by using the leak checker as part of a validation suite as described in the next section.

14.4 Accessing Leak Checker statistics

Activating the probes as described above causes statistics to be collected, class by class on object creation and destruction. To access this information you require the services of the LeaLeakChecker object which can be accessed using it static Instance() method:-

```
#include "LeakChecker/LeaLeakChecker.h"

...

LeaLeakChecker* lea = LeaLeakChecker::Instance();
```

LeaLeakChecker has methods to count the total number of objects created and the number currently active (= created - destroyed):-

```
 UInt_t GetNumCreated(const Char_t* name = 0) const;
 UInt_t GetNumActive(const Char_t* name = 0) const;
```

they take as their argument the name of the required class. If omitted, the sum for all classes is returned. You can reset the counts for any class using:-

```
 void Reset(const Char_t* name = 0);
```

again, omitting the argument means all classes.

A simple summary of the current state of the leak checker system can be sent to the MessageService as follows:-

```
 MSG("Nav",Msg::kInfo) << lea;
```

The leak checker is primarily a first line of defense, to be incorporated into stand-alone package validation. Suppose you have an XxxPackage which has a validation suite contained in the XxxValidate class and executed using the method:-

```
Bool_t RunAllTests();
```

A test function to run it and also perform leak checking could be as follows:-

```
Bool_t TestXxx() {


//  Perform full validation with leak checking and
//  return kTRUE if successful.

#include "LeakChecker/LeaLeakChecker.h"
#include "MessageService/MsgService.h"
#include "XxxPackage/XxxValidate.h"

  Bool_t  ok = kTRUE;

//  Clear leak checker.
  LeaLeakChecker* lea = LeaLeakChecker::Instance();
  lea.Reset();

//  Run validation suite within an inner scope so that
//  all stack based objects are destroyed before checking
//  for leaks.

  {
    XxxValidate v;
    ok = v.RunAllTests();
  }

//  See if leak checker is enabled by checking total created.

  if ( lea->GetNumCreated() == 0 ) {
    MSG("Xxx",Msg::kInfo) << "Warning: Leak checking disabled!!"
  << endl;
  }

//  If it has make sure that there are no objects left.
```

```
  else if ( lea->GetNumActive() ) {
    MSG("Xxx",Msg::kInfo) << "Error, leaks detected! " << lea;
    ok = kFALSE;
  }

  return ok;
}
```

14.5 A final word of warning

The leak checker describe here is not a substitute for a full function leak checker such as Purify for a number of reasons:-

- It relies on the user to manually insert probes in all the right places.
- For MINOS classes the user must ensure the compiler is not left to create default and copy constructors - although its probably best not to rely on these in any case.
- Monitoring foreign class code is clumsy, every new and delete has to be tracked down. Unless all probe points are activated together the leak checker results will be unreliable.
- Although leaks are detected there are no addition facilities to locate the source of the leak. It would be possible to add some basic facilities here if there is the demand.

Despite all of the above, the system should prove useful for any package that can be tested stand-alone. Adding the probes is not particularly onerous and then the package developer can use the system to check for leaks throughout package development on her/his own machine. In this way some of the basic leaks can be eliminated before moving onto integrated system testing using a more powerful checker.

Chapter 15

Dependency

Last significant change: 2001/02/20

As the software system becomes increasingly complex managing the dependencies among and within packages will become increasingly important. The Dependency package is a set of tools to analyzing the dependency structure of the system.

15.1 How to make the list of dependencies?

To make the list of dependencies for the packages in the repository setup your MINFAST path as you normally would then type:

```
gmake dep
```

in the Dependency package area. This will produce a set of files:

- `dep.err` is the log of errors and warnings encountered. Most of these can be ignored since the script does not try to trace all the dependencies through packages external to our software framework. However, searching this file for "Warning" will show all the circular dependencies encountered in the system.

- `all.dep` this full list of package dependencies. See below for tips on interpreting the contents of this file.

- `package.dep` where `package` is the name of a specific package in the system. This file will contain the list of dependencies within a particular package.

The dependency report files have the general format:

```
level. package dep-level dep-package
```

For a package `package level` is the layer of this package in the hierarchy. Packages with high levels depend on many other packages. For example, packages at level 0 depend only on external packages, packages at level 1 depend on things at level 0 and below, etc. The package `package` depends on the package `dep-package` which is found at layer `dep-level` in the hierarchy. Packages flagged with `<n>` (`n` is a numeric label) are circularly dependent on other packages/classes flagged with the same numbers.

15.2 Why is this useful?

I have found this useful in several ways:

- Circular dependencies among packages prevents them from being tested and developed independently from one another and greatly increases the complexity of learning, maintaining, testing, and debugging the software system.

- Within a package, a circular dependency is often the results of a weak design. If two classes need functions located within each other perhaps that's telling you that those functions should be pushed up or down into a third class. Perhaps those two classes are so closely related that you really have one class and not two?

- By laying out the dependency tree this helps identify those classes which can be tested in as unit ("sub-package"). Testing code on many levels helps bugs and other problems to be located quickly. For example, suppose you have five classes that have the dependencies diagramed in Figure 15.1.

The classes in Figure 15.1 should be tested in the following combinations:

- A alone,

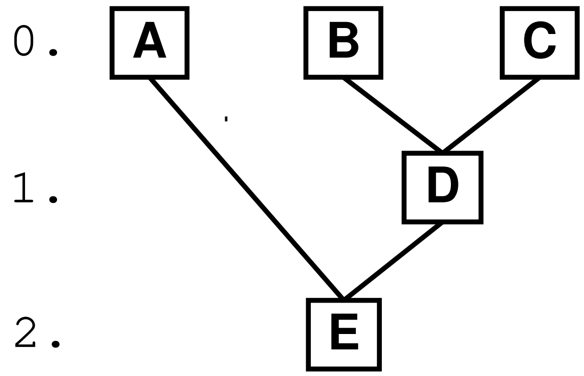

Figure 15.1: A sample set of dependencies. These five classes can be tested in units of A, B, C, DBC, EADBC.

- B alone,
- C alone,
- DBC together
- E-A-(DBC) together

If the classes are tested this way problems can be quickly located. Eg. if the test of E-C-D-A-B fails one can look at the tests of A and D-B-C for the problem. If the problem is not found there then one should look for the error in class E.

Chapter 16

R2HBOOK

Last significant change: 2001/05/30

r2hbook is a very basic utility that allows the conversion of a ROOT file containing TNtuple objects into an HBOOK file. It has been written solely as a stop gap so that physicists, who have not had the time to learn to use ROOT as an analysis tool, can use to look at first Calibration Detector data. TNtuples (ROOT's equivalent of the HBOOK row-wise n-tuple) are the only type of object that the utility can convert. All other objects found in the file are ignored with a warning. Further, the utility can only convert objects in the top level directory; it does not attempt to enter TDirectory objects. It can be used to convert the TNtuples, but not the TH1F histograms found in the file useranalysis.root produced by demojob.

To run, execute the program with input and output file names e.g.:-

```
r2hbook useranalysis.root useranalysis.hbook
```

The second argument is optional, if omitted r2hbook constructs an output file by taking the input file name, removing any trailing file type and then appending `.hbook`, so the above command could be abbreviated to:-

```
r2hbook useranalysis.root
```

Caution: r2hbook increases the HBOOK block size to 8090 to get maximum capacity. You must tell PAW the block size when opening files. For example:-

```
hist/fil 20 useranalysis.hbook 8090
```

Chapter 17

Database Maintenance

Last significant change: 2002/02/22

Eventually this chapter will probably be moved out of the User Manual into its own document.

17.1 Introduction

The DatabaseMaintenance package produces a single binary application: **dbmjob** that provides very basic database maintenance support. Specifically its current function is only as a tool to distribute data between databases.

The flow of data is shown schematically in diagram 17.1. At the heart of the system is the Master Database at Soudan. Most database updates enter the database realm here. At regular intervals dbmjob is used to export all recently updated data and these export files are distributed to all other databases where the data is imported if not already present. This is done by the local database manager again using dbmjob. These primary data flows are shown in red.

Smaller amounts of data come from secondary databases e.g. at CalDet and these are exported up to the Master Database where they join other updates for distribution.

This system relies on the ability to:-

- Record the insertion date so that updates can be incremental.
- Uniquely identify data so that it is not accidentally duplicated if attempting import more than once. For example updates to a secondary

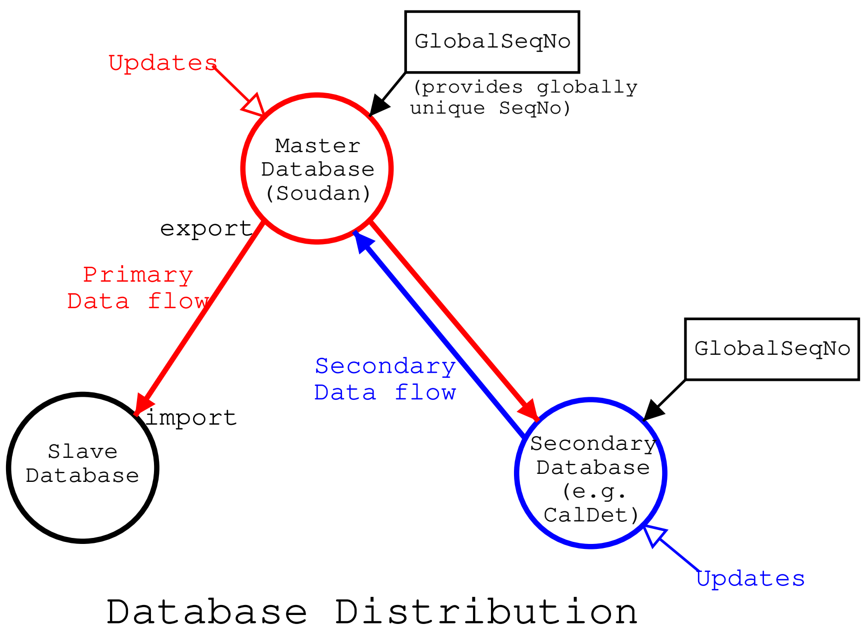

Figure 17.1:

> database might be reflected back if exporting all recent changes. However such data is ignored as duplicated data when resubmitted to the Master.

dbmjob exploits the fact that all Dbi compliant database tables come in pairs, the main data table and an auxiliary validity range table. The auxiliary table records insertion dates and have globally unique SeqNos (Sequence Numbers). The diagram shows how globally unique numbers are assigned. Every database that is a source of data has a `GlobalSeqNo` table that is used to generate sequence numbers. Each time one is allocated the count is incremented in the table. For each database the table operates in a different range of numbers hence ensuring that all are unique. dbmjob moves data in "Validity Packets" i.e. a single row in the auxiliary table and all its associated data rows. The insertion date and SeqNo on the auxiliary row allow dbmjob to support incremental updates and avoid data duplication.

All this implies a very important restriction on dbmjob:-

> *dbmjob can only distribute Dbi compliant database tables i.e. ones*

that come in pairs, the main data table and an auxiliary validity range table.

17.2 Building and Running dbmjob

17.2.1 Building

The DatabaseMaintenance package is a standard Framework package and the dbmjob application is build in the standard way:-

```
cd $SRT_PUBLIC_CONTEXT    %$
gmake DatabaseMaintenance.all
```

17.2.2 Running

Before running, a Database cascade must be defined using the ENV_TSQL_* variables as described in 7.2.1. Alternatively use the -d, -u and -p switches that are also described there or use the ENV_TSQL_UPDATE_* (e.g. ENV_TSQL_UPDATE_USER) set of variables. Where they exist, they will take precedence over the equivalent ENV_TSQL_* variable. This allows for a safe read-only setting of the ENV_TSQL_* variables that can be shared by a group, with just the local database manager also having the ENV_TSQL_UPDATE_* set for write-access. Note that the job switches take priority over everything else.

To run, just type:-

```
dbmjob
```

dbmjob enters interactive mode. For help type `Help` and to quit type `Quit`. The following illustrate simple exporting and importing. For more detail consult the Help command.

Exporting Data

dbmjob always exports data from the first database in the cascade.

To export data use the `Export` command. The syntax is:-

```
Export {--Since <date>}  <table>  <file>
```

This exports the contents of <table> into <file> which can subsequently be imported into another database using the Import command. <table> can be a specific table e.g. PlexPixelSpotToStripEnd or * for all tables. For example:-

```
Export * full_backup.dat
Export -since "2001-09-27 12:00:00" PlexPixelSpotToStripEnd update.dat
```

The first updates the entire database whilst the second just records updates to PlexPixelSpotToStripEnd since midday on the 27 September 2001.

Importing Data

By default dbmjob always imports into the first database in the cascade but this can be overridden.

To Import data use the `Import` command. The syntax is:-

```
Import {--Test } {--DatabaseNumber <no>} <file>
```

This imports the contents <file> into the database. The insertion dates in the file's validity records are replaced by the current date and time so that the insertion dates in the database reflect the local insertion date. Any SeqNo already present will be skipped but the associated data is compared to the corresponding entries in the database to confirm that they are identical, neglecting differences in insertion dates. For example:-

```
Import full_backup.dat
Export --DatabaseNumber 1  update.dat
Import --Test full_backup.dat
```

The first updates the first database (Cascade number 0) whilst the second updates the second database in the cascade. The last does not import at all but still does comparisons so is a convenient way to compare a database to an import file.

Chapter 18

Dispatcher

Last significant change: 2002/06/25

18.1 Introduction

The purpose of the dispatcher is to handle the distribution of data from the online systems (DAQ and DCS) to the near-online client applications which need access to that data.

The dispatcher client applications may be local or remote to the dispatcher server. Examples of local clients are the online monitoring and database update systems. Remote clients may use the dispatcher to access the latest data produced at the detector site.

The dispatcher package provides server side and client side code as is illustrated in Figure 18.1. On the server side, the package provides a parent server (ddsparentserver) which is intended to be run continuously in the background at the detector source site. When the ddsparentserver receives a request for data from a new client, it forks and executes a child server (ddschildserver) to actually handle the data transfer with the client.

On the client side, the package provides a DDSClient class which acts as the interface by which the client may connect to and receive data from the dispatcher server. The client may either interact "directly" with the dispatcher through the DDSClient class, or indirectly within the offline framework via the IoModules package.

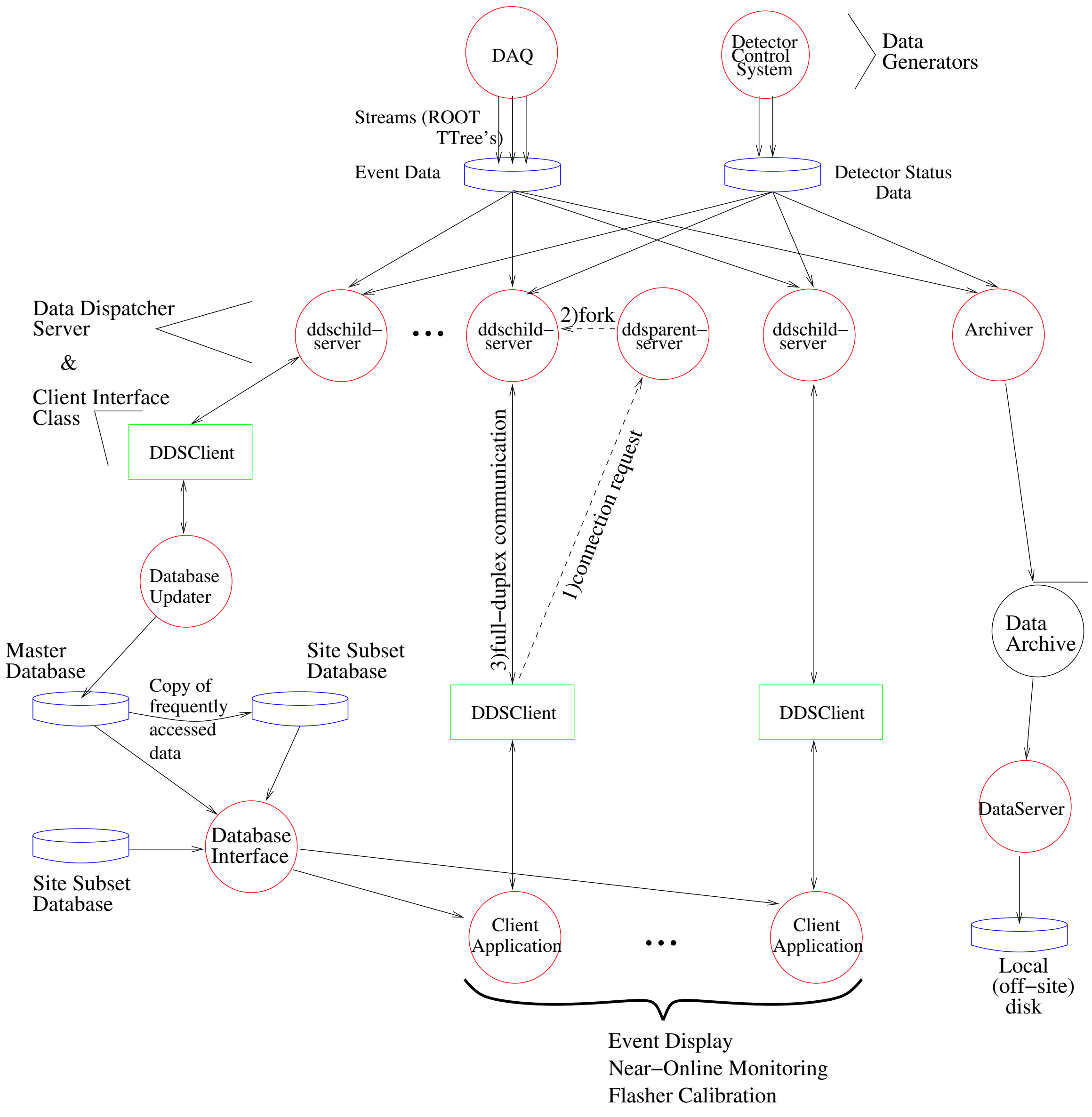

Figure 18.1: MINOS data flow diagram illustrates the role of the dispatcher in the flow of data from the online data generators to the dispatcher's near-online client applications.

18.2 Installing the Dispatcher Parent Server

The typical client of the dispatcher will not need to worry about installing the ddsparentserver, but instead will be able to use the ddsparentserver already installed and maintained at the detector site. For those that do need to worry about installing the ddsparentserver, detailed installation instructions are provided here.

If the dispatcher server will be serving open data files from a disk mounted on a pc different from the one on which the server is run, the dispatcher will need access to the files via the ROOT provided server daemon rootd. This is because rootd was determined to be more reliable at serving open files to the dispatcher server than NFS. The following installation steps should be carried out only if rootd useage is required:

- Install the rootd server for anonymous useage on the data source pc. (Instructions for doing this can be found on the ROOT web site.) Anonymous rootd useage is not strictly required, but is the most convenient way for the ddsparentserver to access the rootd server running on the remote pc. If anonymous useage is used, the rootd home directory described in the rootd installation instructions should be set to the data source directory or a directory upstream of it, e.g. if the data source directory is /mnt/data, the rootd home directory should be set to /mnt or /mnt/data. This is because anonymous rootd useage allows read-only access to only the rootd home directory or directories downstream of this home directory.

- Define the environment variable DDS_DAQ_DATA_DIR, on the host that will run the ddsparentserver, to point to the data source directory using the protocol:

  ```
  setenv DDS_DAQ_DATA_DIR root://<hostname>/<pathname>
  ```

  where the hostname is the name of the data source host running the rootd server, and the pathname is relative to the rootd home directory if the rootd server has been set up for anonymous useage. For example:

  ```
  setenv DDS_DAQ_DATA_DIR root://daqdcp.minos-soudan.org/
  ```

is appropriate for the far detector site assuming the rootd home directory corresponds to the data source directory so that no relative <pathname> to the data files need be specified.

- Define an environment variable DAQ_DATA_DIR to point to the NFS visible path to the data source directory. This is required only if the NFS pathname differs from the <pathname> specified in DDS_DAQ_DATA_DIR. Even when the rootd daemon is used to serve the file contents to the dispatcher server off a remote disk, the dispatcher server must still be able to "see" the directory containing the data using a file sharing system, e.g. NFS, so that it's file handler can access the directory contents and symbolic link 'currentfile' with unix system functions. For example:

```
setenv DAQ_DATA_DIR /mnt/data
```

  is appropriate if the NFS mounted pathname to the data source directory is /mnt/data.

If, on the other hand, the ddsparentserver will be serving either closed files from a disk on either its own or the data source pc, or open files from a disk on its own pc, the rootd server need not be used. In this case, the environment variable DDS_DAQ_DATA_DIR should be set to point to the data source directory path visible to the ddsparentserver through a file sharing system, e.g.

```
setenv DDS_DAQ_DATA_DIR  /mnt/data
```

Setting the DAQ_DATA_DIR environment variable is not necessary.

After the appropriate environment variable(s) have been defined, the ddsparentserver can be started. For example,

```
ddsparentserver -l Verbose >&! ddsparentserver.log &
```

will start the ddsparentserver running in the background (on csh). Both ddsparentserver and ddschildserver log messages will be sent to the log file ddsparenterver.log. The option -l can be used to specify the level at which log messages are written to the ddsparentserver.log file. A full list of options that the ddsparentserver supports can be obtained by specifying the option -h, for example:

```
ddsparentserver -h

Currently, the options supported are:

useage: ddsparentserver -p[port#] -m[max no. clients] -l[loglevel]
 -p: port number to listen on (default=9090)
 -m: maximum number of concurrent clients (default = 10)
 -l: loglevel at which to print messages (default=Info).
     valid arguments are: Verbose, Debug, Info, Warning, Error, Fatal
 -h: print this message
```

To shutdown the ddsparentserver, run the binary ddscomm provided by the Dispatcher package, with the '-s' option, e.g.

```
ddscomm -s
```

This will by default connect to the ddsparentserver running on the ddscomm host pc and listening on port 9090 to shut it down. A list of options supported by ddscomm can be obtained with the -h option, for example :

```
ddscomm -h
```

18.3 Running a Dispatcher Client

There are two methods of connecting to the dispatcher as a client:

- Through the Dispatcher package provided DDSClient class. An example of a driver program which connects to the dispatcher through the DDSClient class is provided in the Dispatcher package as Dispatcher/test/DemoClient.cc.

- Through the offline framework JobC interface. An example of script that uses the JobC class to connect to the dispatcher is provided in the Demo package as Demo/dispatcher.C.

Although the syntax of the two approaches are different, the basic steps that the client must take to receive data from the dispatcher is the same, and is summarized here.

First, the client must connect to the ddsparentserver. For this it must know the server host ip address, and possibly port number on which the ddsparentserver has been set up to listen if this is different than the default of 9090.

Second, the client must submit a subscription to specify the type of data they would like to receive and the mode in which they wish to receive it. Based on the client's subscription, the dispatcher will apply server-side cuts to the data before shipping the data to the client to ensure that the client's subscription is met. As part of this subscription, the client can specify:

- The stream or streams to receive data from, where, since it is only possible to receive data from one file at a time through the dispatcher, a stream name corresponds to a tree name in the dispatcher case. A data file may contain one or more ROOT trees. For example, a file produced by the Daq contains trees DaqSnarl, DaqMonitor, and LightInjection. A client may specify in their subscription that they wish to receive data from any combination of these 3 trees (streams).

- The selection cut to apply to each stream, if any. This selection cut is applied on the server side to select a subset of the data on a given stream. It is specified through a string of the same format that is used in ROOT's TTree::Draw method. (In fact, it makes use of the same ROOT TTreeFormula class used by this method.) The selection string is written in C++ language using C++ operators to specify the desired cut. Some examples of valid selection strings:

```
// A cut to select only every 5th snarl record
``((RawDaqSnarlHeader*)fHeader)->GetSnarl() % 5 == 0''

// A cut to to select every 5th snarl record that
// satisfies the runtype of 2.
``((RawDaqSnarlHeader*)fHeader)->GetSnarl() % 5 == 0
    &&((RawDaqHeader*)fHeader)->GetRunType() == 2''
```

  Selection cuts applied to record header data on the server side are especially efficient because the small headers are stored on a separate tree branch from the larger data blocks and therefore can be read in independent of the data blocks. The dispatcher server applies the selection cut to the header data, and only on selected events will it read in the larger data block and ship the entire record to the client.

- A KeepUpMode by which the client would like to keep up with the data on the server. There are three choices:
  - RecordKeepUp. In this mode the client will receive the most recent record set from the most recent file in the data source directory satisfying the client's subscription. This mode is used, for example, by the near-Online Event Display to view the most recent record produced by the detector at the detector site.
  - FileKeepUp. In this mode the client will receive record sets from the most recent file in the data source directory starting with the first record set in the file satisfying the client's subscription.
  - All. In this mode the client will receive all record sets satisfying the client's subscription from all files in the data source directory. The dispatcher server keeps track of files served to the client and will serve the files in order, even if the client has fallen behind the most recent file. This mode is used, for example, by the Database Updater and Online Monitoring systems.
- DataSource When there are multiple data source directories, this option can be used to specify the data source directory from which the client will receive data. The choices are: DDS::kDaq or DDS::kDcs, which will be parsed on the server side to use the data source environment DDS_DAQ_DATA_DIR or DDS_DCS_DATA_DIR respectively.
- MaxSyncDelay This option is only of interest when the client is subscribing to multiple streams from an open file. An open file is one in which the data source, e.g. the Daq, is still writing to the file from which the dispatcher is serving data. In this case, MaxSyncDelay specifies how long the client is willing to wait to receive perfectly synchronized and sequenced records (as determined by the record VldContext) across the different streams. Setting MaxSyncDelay = 0 means that the dispatcher will make no attempt synchronize the records across the different streams but instead will serve them as they come. In general, if the client wishes to receive perfectly synchronized data, they should specify a MaxSyncDelay that is larger than the rotorooter AutoSave period. For example, at the far detector site, the AutoSave period is set to every 10 seconds, while the (default) dispatcher MaxSyncDelay is set to 15 seconds. Note that this does not mean that the client will

need to wait 15 seconds between each record set served! This establishes the max wait time, but in most cases the delay will be much less than this.

The client may change its subscription at any time during its session with the dispatcher server by resubmitting a new subscription.

Finally, after connecting to the dispatcher and submitting a subscription, the client is ready to receive data from the dispatcher. This is done by requesting the Next record set satisfying the clients subscription. Optionally, the client can request to advance by more than 1 record set as an argument to the Next method. The client can also adjust a waittime before the dispatcher server will give up trying to locate a record set satisfying the client's subscription and return with a timeout message. By default this is set to 120 seconds.

The client can also jump to the file of interest by using either the GoToFile or GoToNext file methods as illustrated in the examples.

18.3.1 An Example Use of DDSClient Interface

This example illustrates how to write a driver program that makes use of the DDSClient class to receive data from the dispatcher. See Dispatcher/test/DemoClient.cc for more examples of useage and additional comments.

```
#include "TIterator.h"
#include "RawData/RawRecord.h"
#include "MinosObjectMap/MomNavigator.h"
#include "Dispatcher/DDSClient.h"
#include "Dispatcher/DDSSubscription.h"

int main() {

  // Create a new DDSClient object connected to the server running on
  // daqdds.minos-soudan.org
  DDSClient* ddsclient = new DDSClient("daqdds.minos-soudan.org");

  // Check validity of connected socket before using it
  if ( !ddsclient -> IsValid() ) {
    cout << "Error in creation of socket connected to server." << endl;
```

```
    delete ddsclient; ddsclient = 0;
    return 1;  // end of session
  }

  // Configure Subscription and send it to Server.
  ddsclient->GetSubscription()->SetKeepUpMode(DDS::kAll);
  ddsclient->GetSubscription()->SetStreams("DaqSnarl,DaqMonitor");
  ddsclient->GetSubscription()->SetSelection("DaqSnarl",
            "((RawDaqSnarlHeader*)fHeader)->GetSnarl() % 10 == 0
          && ((RawDaqSnarlHeader*)fHeader)->GetTrigSrc()==4");


  DDS::EMessageType msgrc = ddsclient->Subscribe();
  if (msgrc != DDS::kOk) {
    cout  << "An error message " << DDS::AsString(msgrc)
          << " was received from DDS::Subscribe." << endl;
    return 1;
  }

  // Begin receiving record sets from server.  This loop is set up to
  // continue receiving record sets until an error condition occurs, e.g.
  // a timeout.
  Int_t nentry = 0;
  msgrc = DDS::kOk;
  while (msgrc == DDS::kOk) {
    MomNavigator* mom = 0;
    UInt_t waittime = 120;
    Int_t advanceby = 1;
    msgrc = ddsclient -> Next(mom,waittime,advanceby);
    if (msgrc == DDS::kOk) {
      nentry++;
      if ( mom ) {
        TIter fiter = mom -> FragmentIter();
        RawRecord* rawrecord;
        while ( (rawrecord = dynamic_cast<RawRecord*>(fiter.Next())) ) {
          // Process the record
        }
        delete mom; mom = 0; // deleting mom record also deletes all records
```

```
      }
    }
  }

  // See Dispatcher/DDS.h for a list of DDS::EMessageType's that may be
  // returned by DDSClient methods.
  cout << "\nAfter "<<nentry<< " total entries retrieved, an error message:\n"
       << DDS::AsString(msgrc)
       << " was returned from DDSClient::Next.\n" << endl;

  ddsclient -> Shutdown();
  delete ddsclient; ddsclient = 0;

  return 0;

}
```

18.3.2 An Example Use of JobC Interface

This example shows how to use the JobC Interface to the dispatcher. The subscription is identical to the one in the DDSClient interface example. See Demo/dispatcher.C for additional options.

```
// usage: loon -bq dispatcher.C
{

  JobC j;

  j.Input.Set("Format=DDS");
  j.Input.Set("DDSServer=daqdds.minos-soudan.org");
  j.Input.Set("Streams=DaqSnarl,DaqMonitor");
  j.Input.Set("DDSKeepUpMode=All");  // default is FileKeepUp
  j.Input.Set("DDSTimeOut=120");      // (seconds) waittime
  j.Input.Select("DaqSnarl",
            "((RawDaqSnarlHeader*)fHeader)->GetSnarl() % 10 == 0
          && ((RawDaqSnarlHeader*)fHeader)->GetTrigSrc()==4");
```

```
  j.Path.Create("Demo","EventDump::Ana");
  j.Path("Demo").Mod("EventDump").Cmd("Dump Fragments");
  j.Path("Demo").Run();  // Runs until an error occurs (e.g.timeout)

}
```

Chapter 19

Minos Event Display (sub) framework: Midad

Last significant change: 2002/07/26

19.1 Introduction

The Midad sub framework allows one to display and interact with graphical representations of data objects. Of historical note, there is an obsolete previous version called MIDAD (all caps). This code is still in the repository, but should not be used.

This chapter is broken up into the following sections:

- Section 19.2 shows how one can display events with out writing a single piece of code.
- Section 19.3 gives the design of the elements that make up the framework. This is useful if one wants to configure the framework from a ROOT macro (currently the only way) or if one wants to extend Midad.
- Section 19.4 shows the details of how one configures Midad via a ROOT macro and presents some default configurations that are available.

19.2 Using and configuring Midad with out writing code.

Section to be written.

19.3 Midad Design

This section describes the design of Midad. It is usefull for the new user to briefly skim through this, particularly the description of in section 19.3.2.

19.3.1 Design Overview

Midad generally follows the Model-View-Control (MVC) pattern (see Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC) by Steve Burbeck, Ph.D. for the canonical paper), either explicitly or implicitly. The basic idea of MVC is to isolate the data (Model) from the things that displays it (Views) and the things that modifiy it (Controls).

These three concepts are formalized into actuall base classes of the same name:

Models: All Models have a modified signal which is triggered whenever their internal state is changed. Models may have other signals as well. Models export an interface to their data to be used by corresponding Controls.

Views: Views subclass View¡ModelType¿ and should implement `Configure(ModelType&)`. This method is called in response to the Model's modified signal which is connected in View¡¿::ConnectModel(). The Model is held by the View base class in a SigC::Ptr¡¿ (see memory management below).

Controls: Controls, like Views, are templated on a Model type and hold their Model in a SigC::Ptr¡¿. However, instead of responding to a Model::modified signal, they modify the Model via whatever interface it exports.

19.3.2 Main graphical objects

The main objects a user is most likely to interact with or see on the screen are the following.

Displays: Display is a base class providing a default compound widget which includes a menubar, a button bar a central box for sub classes to fill and a statusbar. Typically, Display sub classses will fill this central box with one or more Scenes and any accompanying widgets.

Scenes: A Scene is a TPad which has Range objects to control its range and knows about Scenery and SceneElements (see below).

One can Draw() non-Scenery TObjects in the usual TPad/TCanvas way.

One draws Scenery by Add()ing it to the Scene. If new'ed with manage(), the Scenery will be memory managed (it is held in a SigC::Ptr¡Scenery¿).

A Scene attaches to a Scenery's ”modified” signal in order to mark the Scene's Modified bit.

A Scene has an associated ”Scene Type” available as either a string (eg, ”U-Z”, ”U-Plane-Strip”) which is stored in the TPad's title variable (second arg to constructor) or an enum (kUZ, kUPlaneStrip). This Scene Type can be queried by the Scenery in order to know what the Scene's ”User Corrdinates” are (ie, ”U-Z” means draw in meters in the U vs. Z projection, ”UPlaneStrip” means draw in plane# on X and strip# on Y).

Scenery: Scenery is the TPad/ROOT side of a graphical View of a Model (the full class will likely inherit from both Scenery and some View, however this isn't a requirement - one could create non-View scenery).

The Scenery base class provides the ”modified” signal (so it is kind of a model from the consideration of a Scene). The inheriting class should call ”modified.emit()” if it wants the Scene notified that it should be updated.

The Scenery base class overloads the ExecuteEvent() and Distanceto-Primitive() methods in order to provide various signals related to mouse movement and clicking and track which graphical TObject is under or closest to the pointer.

Scenery is a TObject holding collections of one or more graphical primitives (plain TObjects or TObject+SceneElement objects, see below). To implement the graphical object a Scenery subclass should call Scenery::ClearPrimitives() create and give new primitives to Scenery::AddPrimitive() in response to any change in the scenery.

Scenery has various virtual methods which provide hooks into the rest of the system (eg: GetInfo() to return a string for display in the status bar).

Scenery can get a pointer to its Scene via GetScene().

SceneElements: A graphical "primitive" can inherit from just some TObject or both a TObject and a SceneElement in order to provide more features, such as custom context menus, triggering info placed in the status bar when the mouse pointer is over them, etc.

19.3.3 Update Policy for Scene and Scenery

The Scene and Scenery classes sort of follow the View and Model patterns with Scenes being Views (or rather composite-Views) of multiple Scenery Models. However, the difference is that there is a 1:n relationship between Scene:Scenery. This causes problems with updating. When each Scenery changes its "modified" signal triggers the "dirty" bit in the Scene to be set (ie, TPad::Modified() is called). The modifications don't take effecut until TPad::Update() is called.

The trouble comes in when one has multiple Scenery objects changing and deciding how to call TPad::Update(). Naively and simply one could call Update after each Scenery is modified. However, this causes multiple redraws if multiple Scenery exists in the Scene. This is slow an unsightly.

To fix this, Update() needs to be called "out of band" of the Scene/Scenery connection after all Scenery has reported their modification to the Scene. One way to do this is to have each Controller trigger a "global_update" signal after all of its signals have been dispatched and to which all Scenes (and other classes in a similar predicament) can attach.

19.3.4 Memory Management

MIDAD make heavy use of SigC::Ptr¡¿s to provide flexible memory management. All Gui classes hold their children and Controls and Views hold their

Models in SigC::Ptr¡¿s. Also Scenes hold their Scenery and Ranges are held in this manner.

This allows the object held in a Ptr to be deleted when the Ptr is deleted if the object was created on the heap with manage() and there are no other Ptrs referencing it. Ie:

```
SigC::Ptrobj;
{
    SigC::Ptr obj1 = manage(new SomeObject());
    SigC::Ptr obj2 = manage(new SomeObject());
    obj = obj2
    // obj1, obj2 lives
}
// obj1 is deleted, obj2 lives on.
```

To be manageed in this way the object must inherit from SigC::Object. If an object is created on the heap with out manage or created on the stack then the deletion of the Ptr will not cause a deletion of the object.

A consequence of using this method of memory management is that it can be dangerous to hold a managed object in a plain pointer. Gui child widgets, Models, Scenery, and Ranges are so managed so should be always stored in Ptrs.

19.4 Configuring Midad in a ROOT macro

The Midad sub framework is accessed by loading in `libMidad.so`. The main external interface is the `Mint` object. However, for most people, once created the instance of `Mint` can be safely ignored as it attaches and talks to the main `JobControl JobC` object (see section 1). However, one can use it to control Midad from a ROOT macro file.

19.4.1 Minimal setup

Here is how one would minimally configure Midad. First, a minimal Job-Control set up:

```
JobC* setup_jobc()
{
    gSystem->Load("libMidad.so");

    JobC* jc = new JobC;
    jc->Path.Create("MinimalMidad",
                    "DigitListModule::Get "
                    "DigitListModule::Reco ");
    jc->Input.Set("Streams=DaqSnarl");
    return jc;
}
```

It is important to load in `libMidad.so` before proceeding. Next, a minimal Midad setup:

```
Mint* start_midad(JobC* jc)
{
    Mint* mint = new Mint(*jc);

    // Tell Midad about digits
    Registry digits(false);
    digits.Set("CandName","canddigitlist");
    digits.Set("CandType","CandDigitListHandle");
    mint->MakeModel("digits","DigitList",&digits);

    // Create a "Multi" display
    mint->SpawnDisplay("my multi display","Multi");

    // Add some digit "scenery"
    mint->AddScenery("my multi display","All","DigitList","digits");

    return mint;
}
```

At this point one can use `jc->Run(1)` or the buttons provided by the display to navigate the events. You should be able to see un-DeMuxed candidate digitizations.

19.4.2 Detailed setup

Chapter 20

Monte Carlo Information

Begun: 2004/10/04, M. Kordosky `kordosky@fnal.gov`

This section documents some of the Monte Carlo truth information which may be accessed by offline jobs. The emphasis is on the StdHep block. The information was gathered from emails, GEANT documentation, private conversations and "legacy" web-pages (most of which were authored by Robert Hatcher).

20.1 Accessing Truth Information

It is possible to configure GMINOS to record a list of interesting processes in each event. The information eventually shows up as a **TClonesArray** of **TParticle** objects stored inside **SimSnarlRecord**s. Inside the offline code (e.g., in a JobModule) users may retrieve the information via:

```
SimSnarlRecord* simrec = dynamic_cast<SimSnarlRecord*>
                           (mom->GetFragment("SimSnarlRecord"));

const TClonesArray* simstdheparray = dynamic_cast<const TClonesArray*>
                                       (simrec -> FindComponent("TClonesArray",
                                          "StdHep"));

// loop over array of TParticles
Int_t nstdhep = simstdheparray->GetLast()+1;
for ( Int_t istd = 0; istd < nstdhep; istd++ ) {
  TParticle* simstdhep =
     dynamic_cast<TParticle*>(simstdheparray->At(istd));
  // Do whatever you want
```

```
  // See NtpMCModule::FillNtpMCStdHep() for examples
}
```

In addition the truth information is recorded by the JobControl module **NtpMCModule** as **NtpMCStdHep** objects which are stored in **NtpMCRecord**s. That's a mouthful! Exploring the information in **NtpMCRecord**s is pretty easy inside a loon session:

```
// open a file
loon [0] TFile f("f24110001_0000.sntp.R1.9.root");

// what's in the file?
loon [1] .ls
TFile**         f24110001_0000.sntp.R1.9.root   Persistency file
 TFile*         f24110001_0000.sntp.R1.9.root   Persistency file
  KEY: TTree    NtpMC;1 Persistency stream tree
  KEY: TTree    NtpSR;1 Persistency stream tree
  KEY: TTree    NtpTH;1 Persistency stream tree
  KEY: TNamed   FileEnd;1       PerFile End Key

// what do NtpMC records look like?
loon [2] NtpMC->Show(0);
======> EVENT:0
 NtpMCRecord     = NULL
 fUniqueID       = 0
 fBits           = 50331648
 fName           =
 fTitle          =
 fHeader.fUniqueID = 0
 fHeader.fBits   = 50331648
 fHeader.fVldContext.fUniqueID = 0
 fHeader.fVldContext.fBits = 50331648
 fHeader.fVldContext.fDetector = 2
 fHeader.fVldContext.fSimFlag = 4
 fHeader.fVldContext.fTimeStamp.fSec = 1079981341
 fHeader.fVldContext.fTimeStamp.fNanoSec = 51
 fHeader.fRun    = 24110001
 fHeader.fSubRun = 1
 fHeader.fRunType = 0
 fHeader.fErrorCode = 0
 fHeader.fSnarl  = 0
 fHeader.fTrigSrc = 0
 fHeader.fTimeFrame = -1
 fHeader.fEvent  = -1
 mc              = 1
 mc.fUniqueID    = 0
```

```
mc.fBits        = 50331648
mc.index        = 0
mc.inu          = 12
mc.inunoosc     = 14
mc.itg          = 2212
mc.iboson       = -99999
mc.iresonance   = 1003
mc.iaction      = 0
mc.a            = 56.000000
mc.z            = 26.000000
mc.sigma        = 216.959991
mc.x            = 0.056173
mc.y            = 0.337257
mc.q2           = -0.134168
mc.w2           = 3.070345
mc.emfrac       = 0.000000
mc.vtxx         = 169.134476
mc.vtxy         = -101.748550
mc.vtxz         = 959.161987
mc.p4neu[4]     = 0.000005 , 0.220568 , 3.796557 , 3.802959

mc.p4neunoosc[4] = 0.000005 , 0.220566 , 3.796557 , 3.802959

mc.p4tgt[4]     = 0.201376 , -0.069906 , 0.001095 , 0.928162

mc.p4shw[4]     = 0.266551 , 0.203926 , 1.338785 , 1.321045

mc.p4mu1[4]     = 0.000000 , 0.000000 , 0.000000 , 0.000000

mc.p4mu2[4]     = 0.000000 , 0.000000 , 0.000000 , 0.000000

mc.p4el1[4]     = 0.000000 , 0.000000 , 0.000000 , 0.000000

mc.p4el2[4]     = 0.000000 , 0.000000 , 0.000000 , 0.000000

mc.p4tau[4]     = 0.000000 , 0.000000 , 0.000000 , 0.000000

stdhep          = 9
stdhep.fUniqueID = 0, 0, 0, 0, 0, 0, 0, 0, 0
stdhep.fBits    = 50331648, 50331648, 50331648, 50331648,
                  50331648, 50331648, 50331648, 50331648,
                  50331648
stdhep.index    = 0, 1, 2, 3, 4, 5, 6, 7, 8
stdhep.parent[2] = -1 , -1
, -1 , -1
```

```
, 1 , 1
, 1 , 1
, 0 , 0
, 2 , 2
, 5 , 5
, 5 , 5
, 5 , 5

 stdhep.child[2] = 4 , 4
, 2 , 3
, 5 , 5
, -1 , -1
, -1 , -1
, 6 , 8
, -1 , -1
, -1 , -1
, -1 , -1

 stdhep.IstHEP   = 0, 0, 11, 1, 1, 3, 1, 1, 1
 stdhep.IdHEP    = 12, 1056026000, 2212, 1055025000, 12,
                   10000222, -211, 211, 2212
 stdhep.mass     = 0.000000, 52.103485, 0.903351, 51.174881,
                   0.000000, 1.752240, 0.139568, 0.139568,
                   0.938280
 stdhep.p4[4]    = 0.000005 , 0.220568 , 3.796557 , 3.802959
, 0.000000 , 0.000000 , 0.000000 , 52.103485
, 0.201376 , -0.069906 , 0.001095 , 0.928162
, -0.201376 , 0.069906 , -0.001095 , 51.175323
, -0.266546 , 0.016642 , 2.457773 , 2.472240
, 0.467927 , 0.134020 , 1.339880 , 2.258881
, 0.093360 , 0.191389 , 0.293592 , 0.388615
, 0.254709 , 0.230348 , 0.752490 , 0.838843
, 0.119858 , -0.287717 , 0.293798 , 1.031423

 stdhep.vtx[4]   = 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
, 1.691345 , -1.017485 , 9.591620 , 0.000000
```

The **NtpMC** tree contains **NtpMCRecord** objects. The objects have

Iaction Codes

code	interaction type
0	neutral current
1	charged current

Table 20.1:

three main components:

Header Event "time" as well as run, snarl and event numbers.

MC An array of **NtpMCTruth** objects. For Far detector events the array should only hold one object. The **NtpMCTruth** objects record information on the final state of the neutrino interaction. This includes the neutrino flavor and momentum 4-vector, true vertex, interaction type (CC,NC), interaction mode (QE, resonance production, DIS, coherent pion production) and event kinematics (e.g. x, y, q-squared, W-squared).

StdHep An array of **NtpMCStdHep** objects. Each object corresponds to one particle and has words for the particle type (**IdHEP**), mass (**mass**), momentum 4-vector (**p4[4]**) and production vertex (**vtx[4]**). Additionally a status code (**IstHEP**) is provided to indicate the mechanism by which the particle was produced. Finally, two pairs of words (**parent[2]**, **child[2]**) provide a range of indices indicating the immediate parent(s) and daughters of the particle. Array indexing begins at zero. An index of -1 indicates those particles that initiated the event. In neutrino events these are the neutrino and the target nucleus.

20.2 Various Numerical Codes

Two codes are used to denote the sort of interaction that occurred between the neutrino and the target. These codes, known as "Iaction" and "Iresonance", are shown in Tables 20.1-20.2. The tables are valid for NEUGEN3 but perhaps not for earlier versions.

Table 20.3 lists those particles most commonly seen in our events. The

Iresonance Codes

code	interaction mode
1001	Quasi-elastic
1002	Resonance production
1003	Deep inelastic scattering
1004	Coherent pion production

Table 20.2:

Common IdHEP Codes

Code	Particle
22	photon
11, 13, 15	electron, muon, tau
12, 14, 16	neutrinos: electron, muon, tau
211, 111, 221	charged pion, neutral pion, eta
213, 113, 223	charged rho, neutral rho, omega
321, 311, 310, 130	charged kaon, neutral kaon, k-short, k-long
2212, 2112	proton, neutron
1114, 2114, 2214, 2224	Delta(1232): -, 0, +, ++
28	“geantino” (placeholder)
100200-100255	neugen2 resonance
10000200-10000255	neugen3 resonance

Table 20.3:

numbering convention and the full table are documented in the PDG[1]. In particular many resonances have been left out of the table above. Antiparticles have a negative sign in front of the numeric code (e.g. a negative pion is -211). Geantinos are used by NEUGEN for bookkeeping.

Atomic nuclei are marked with a 10 digit code, such as `1056026000`. The code is `1AAAZZZJJJ` where `AAA` is the atomic mass, `ZZZ` is the charge and `JJJ` is the (angular) spin. Currently the spin portion is unused. Therefore `1056026000` denotes Fe-56.

The IstHEP status codes are identical to those listed in the GEANT manual except that 200 is added to each code. There are some special status codes which are used in the final state from NEUGEN and listed in Table 20.5. Those codes are valid for NEUGEN3 but perhaps not for earlier versions.

20.3 StdHep controls in GMINOS

GMINOS has a handy feature which allows the user to "throttle" the output to StdHep. At a minimum each neutrino induced event includes the final state from NEUGEN. If a resonance was produced it is usually (always?) decayed and the daughters are also shown in StdHep. Two FFREAD[2] cards, `SAVT` and `C2ND`, are used to control the StdHep output. The `SAVT` card denotes the event numbers for which the StdHep information is to be saved. Therefore:

```
SAVT 1 7 28
```

would output the StdHep block for events one, seven and twenty-eight. One can save a range of events by using negative numbers:

```
SAVT -1 -28
```

That line would output the StdHep block for events one through twenty-eight. To write out every event you might do:

```
SAVT -1 -9999999
```

[1]http://pdg.lbl.gov/2002/montecarlorppbook.pdf
[2]http://wwwasdoc.web.cern.ch/wwwasdoc/WWW/ffread/ffmain/ffmain.html

IstHEP Status Codes

Name	Code	Description
NEXT	201	particle has reached the boundary of current volume
MULS	202	multiple scattering
LOSS	203	continuous energy loss
FIEL	204	bending in magnetic field
DCAY	205	particle decay
PAIR	206	photon pair-production or muon direct pair production
COMP	207	Compton scattering
PHOT	208	photoelectric effect
BREM	209	bremsstrahlung
DRAY	210	delta ray production
ANNI	211	positron annihilation
HADR	212	hadronic interaction
ECOH	213	hadronic elastic coherent scattering
EVAP	214	nuclear evaporation
FISS	215	nuclear fission
ABSO	216	nuclear absorption
ANNH	217	anti-proton annihilation
CAPT	218	neutron capture
EINC	219	hadronic elastic incoherent scattering
INHE	220	hadronic inelastic scattering
MUNU	221	muon-nuclear interaction
TOFM	222	exceeded time of flight cut
PFIS	223	nuclear photo-fission
SCUT	224	the particle due to bending in magnetic field was unexpectedly crossing volume boundaries and the step has been halved to avoid this
RAYL	225	Rayleigh effect
PARA	226	parameterization activated
PRED	227	error matrix computed (GEANT tracking)
LOOP	228	not used
NULL	229	no mechanism is active, usually at the entrance of a new volume
STOP	230	particle has fallen below energy threshold and tracking stops
LABS	301	Cerenkov photon absorption
LREF	302	Cerenkov photon reflection/refraction
SMAX	303	step limited by STEMAX
SCOR	304	correction against loss of precision in boundary crossing
CKOV	305	Cerenkov photon generation
REFL	306	Cerenkov photon reflection
REFR	307	Cerenkov photon refraction
SYNC	308	synchrotron radiation generation
STRA	309	PAI or ASHO model used for energy loss fluctuations

Table 20.4:

NEUGEN final state codes

Code	Description
0	primaries (neutrino and nucleus)
1	stable (not a resonance) final state particle
3	decay (usually of a resonance)
11	marks the target nucleon

Table 20.5:

The `C2ND` card is used to set a momentum threshold (in GeV/c) for each GEANT process (listed in descending order from 1 to 39 in Table 20.4). Positive thresholds values will save all secondaries produced via the specified process provided that at least one secondary is above the threshold. Negative thresholds will save only those secondaries with a momentum larger than the (absolute value of) the threshold value. For example:

```
C2ND 12=0.01
C2ND 13=-0.05
```

would save all secondaries from 'HADR' (code 12) processes in which one secondary had a momentum larger than 10MeV/c but for 'ECOH' (code 13) processes would only save secondaries with a momentum larger than 50MeV/c. A range of values can be given as:

```
C2ND 60*0.02
C2ND 12=-0.1
```

which initially sets all sixty thresholds to 20MeV/c and then resets the 'HADR' threshold to 100MeV/c.

Currently the `C2ND` card only saves secondaries but in the future it may be helpful to save the primaries as well.

20.4 Regarding `TrackId`

GEANT operates by propagating individual particles through the detector geometry - in other words it makes "tracks". Interactions may create new tracks. GEANT keeps track of the tracks (clever eh?) by assigning a unique number, called `TrackId`, to each track. The tracks do not have to appear in the StdHep output in order to be tagged with a `TrackId`. **DigiScintHit**s record the ID of the track that created them and so propagate the information into the offline framework where it is used by, among other things, the **Truthifier** code. At one point in the past the author noticed negative `TrackId`s which made him fear that there was a bug. As it turns out, it's a feature, as Robert explains:

> TrackId represents an index into the StdHep table – but that table would grow to outrageous size if every particle were recorded. So instead we use the convention that -TrackId means a hit from a particle that the true TrackId begat (at some level above). Thus deep in an EM shower initiated by a high energy electron (TrackId=5 for instance) some lowly electron is going to have TrackId=-5 but the state of all the e+/e-'s in the shower are not recorded.

More recently someone noticed that hits created about seven micro-seconds after the start of the event were being attributed, via `TrackId`, to a pion produced at the (neutrino) vertex. In light of Robert's comment above, it's likely that the hits were produced by the neutron progeny of the original pion.