Available on CMS information server **CMS IN 1999/009**

CMS Internal Note

The content of this note is intended for CMS internal use and distribution only

17 February 1999

Getting started with ORCA

St. Wynhoff

III. Phys. Institut, RWTH-Aachen, Aachen, Germany

Abstract

This write-up gives an introduction in the usage of ORCA, the **O**bject oriented **R**econstruction for **C**MS **A**nalysis. It shows how to carry out physics analysis or detector studies within the new framework. After a brief overview of ORCA, the technical procedure to access the code and create executable programs is discussed. Then the building blocks needed to write a program to work with ORCA are explained using one of the examples included in ORCA.

This version of the document matches the releases ORCA_1_0_X.
Some statements may not apply to other versions of ORCA.

1 Introduction

This write-up gives an introduction in the usage of ORCA [1], the Object oriented Reconstruction for CMS Analysis. The intended audience are "normal" users who want to do physics analysis or detector studies – things they did until now with CMSIM [2]. People participating in code development for ORCA itself need a deeper understanding of the framework; they may however also profit from this document. The general layout of the system and some technical or slang expressions used will be discussed. Also a step by step instruction on how to use the Examples subsystem will be given. Special emphasis is put on the predefined workspace for private programs of the users. Finally an example application is discussed in detail.

2 *O*bject Oriented *R*econstruction for *C*MS *A*nalysis

A general description of the ORCA project can be found elsewhere [3]. It is a framework for reconstruction and is intended to be used for final detector optimisations, trigger studies or global detector performance evaluation. It is an object oriented system for which C++ has been chosen as programming language. ORCA has two logical layers, *subsystems* and then within a subsystem *packages*. For an overview of the currently available subsystems see Figure 1.

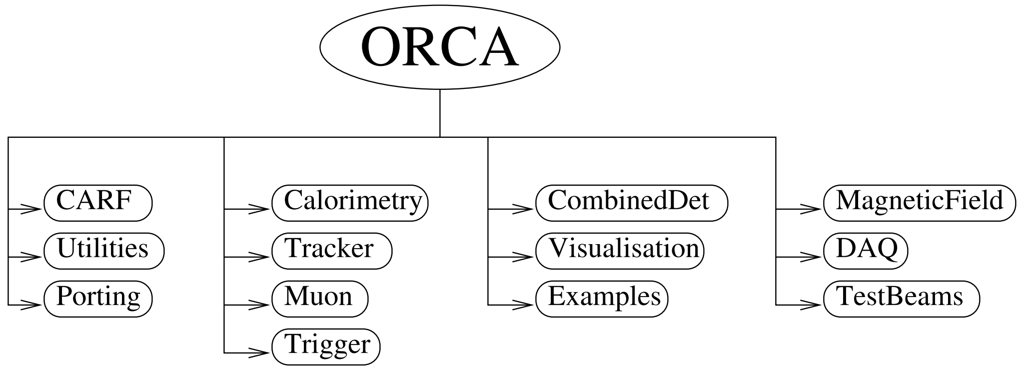

Figure 1: Subsystems of ORCA.

The packages within a subsystem typically produce a single library dedicated to cope with a certain aspect of the corresponding sub-detector (e.g. for Calorimetry: G3EcalDigits, G3HcalDigits, EcalCluster, . . .) or reconstruction utility. For this note the subsystem Examples is of special interest. There the packages correspond to typical analysis tasks performed by "normal" users.

For each package at least the following four sub-directories are present, where only the first two are of interest for external users of the packages:

- **doc** The documentation related to the package.
- **interface** The C++ header files that can be referenced from other packages in the same or in different subsystems. In case of missing documentation these header files reveal the implemented functionality of the package.
- **src** The actual source code and C++ header files local to the package.
- **test** Source code of test programs for quality control of the package.

Besides the individual packages a subsystem has a directory *domain* containing files and documentation which are common to the packages of the subsystem.

2.1 Accessing ORCA

2.1.1 Using the CVS server

ORCA is available from a dedicated CVS[1)] server running at CERN. Version 1 of ORCA is in practice usable only on CERN machines and not exportable. The architectures SUN (e.g. cms.cern.ch) and Linux are supported, a special version for Windows NT is available. The user needs a valid AFS token (`klog/kinit`) for authorisation.

A set of scripts is provided (see Sections 2.3 and 2.4) to access the source code and copy it from the server to a local directory. This directory is called private checkout area.

2.1.2 The public checkout area

A public checkout area (`/afs/cern.ch/cms/oo/Software/ORCA/releases`) contains the most important versions available. There one has easy access to the full source code, e.g. of the header files and the examples.

2.2 Versions of ORCA

The versions of ORCA follow a special naming scheme, containing three numbers, e.g. ORCA_1_0_0. The numbers indicate the release number (1), changes in some interfaces with a release (0) and bug-fixes within the release (0). The latest release has always the alias "current" for easier access. This version should be used by non-experts.

For the various subsystems of ORCA besides the release versions special tags are available to access code that is still in the development phase, but has been tested to some extend. They are [5]:

- **beta** The code has been tested on subsystem level.
- **alpha** The code has been tested on package level in a subsystem.

Developers may want to access the "head" version of the code. This version contains the latest changes done by the developers and does not necessarily work. Developers have also the possibility to tag code with names to their liking.

2.3 Initialising to work with ORCA

At CERN CMS users should use the command `project ORCA` to preset their environment for work with ORCA. The first versions of ORCA use some CMSIM functions and one has to initialise the correct CMSIM environment. To do this one has to issue `cms115` or the corresponding `cms{new|pro|dev|old}`

These two commands have to be issued every time you login and want to use ORCA.

2.4 Software Release Tools

To actually build ORCA the package Software Release Tools (SRT) is used. This system consists of a number of GNUmakefiles and needs a certain directory structure. You have to create this directory structure when you start to use ORCA for the first time and every time you change to a different version of ORCA. It will be filled with a local copy of ORCA or parts of ORCA that you can freely modify. If you intend to checkout all or a large part of ORCA you should create this structure in the scratch area. If you want to use only the workspace you can try to create it in your AFS space.

1. Go to the directory where you want to create your local copy, e.g.
   `cd $SCRATCH`
2. Type `newrel -t current` *`myrelease`*. A subdirectory with the name
   `myrelease` will be created and a little message reminding you how to continue will be displayed. The name of the subdirectory is arbitrary.
3. Go into the subdirectory you just created (`cd myrelease`)

[1)] Concurrent Versioning System, see [4]

4. Add the subsystems you want to work on with `addpkg` *`subsystemname`*, e.g. `addpkg Examples`. This command can be used in several ways. You can choose the "head" version with `addpkg -h Examples`, you can use the *alpha* or *beta* version with `addpkg Examples` *`version`* or you can only get a single package from a subsystem with `addpkg Examples/workspace`. The `-h` option and a *`version`* tag cannot be given at the same time.

If you get error messages from the CVS server, check if you have a valid AFS token and whether you are registered as an ORCA developer on the WWW page `http://cmsdoc.cern.ch/cmsoo/projects/ORCA/...` `.../developer_list.html`. If you don't find your name, contact the CMS librarian Chris.Williams@cern.ch.

Documentation about ORCA which is not related to a particular subsystem can be checked out with the command `addpkg ORCA/doc`. In a similar way the command `addpkg ORCA` will checkout the complete ORCA, including all subsystems.

3 Using the predefined workspace

3.1 Setup the user files

Within the Examples subsystem a workspace is provided where the user can easily compile his private analysis code and link it against the central ORCA libraries. To check out this package to a private area just do
`addpkg Examples/workspace`.
This will give you the directory structure shown in Figure 2.

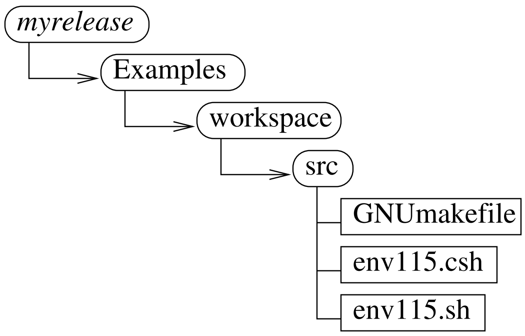

Figure 2: Files in the package workspace of subsystem Examples

Then copy your source files i.e. your own C++ analysis code to `Examples/workspace/src` or go to that directory and create links to your files.

With the files accessible within the `src` directory you have to edit the `GNUmakefile` such that your source files are compiled, eventually put into a private library and linked. The convention is that files with the extension `.cpp` contain the source code of programs, while files with the extension `.cc` are put into a library. Now all you have to do is to add your `.cpp`-files as targets right after `BINS =` in the `GNUmakefile`, omitting the `.cpp` extension. If you have files to put into a library, add a (unique) library name after `LIB = .` Choose the libraries you need from the different subsystems. They will be picked up from the public checkout area (see 2.1.2) or – if present – from your private checkout within `myrelease`.

In the public checkout area you can find all existing libraries in one place. Remember that usually each package in a subsystem corresponds to a library. **This procedure will be replaced by a more convenient mechanism soon.**

3.2 Build and run the programs

To build your executable and library you can use the commands discussed in Section 4.1.

Currently ORCA reads the detector geometry from RZ files created with CMSIM. Tis file can be created with the datacards
`CFIL 'DETO' xxxx.rz`
`RSAV 'INIT'`
in the CMSIM run. Also the simulated events, more precisely the GEANT hits and/or DIGIs, are taken from

CMSIM – in form of FZ files. To choose the input files for a program one has to set environment variables (see Section 4.2). The files `env115.csh` and `env115.sh` set the necessary variables to sensible values for `csh`- and `sh`-type shells. In particular, geometry and event files are available in
`/afs/cern.ch/cms/oo/reconstruction/datafiles`.
There also the datacards used for the production of the FZ files are stored. They can be used as examples for private simulation jobs.

Finally one can run the program.

4 Building and running ORCA

4.1 Building libraries and executables

Once you have checked out some code from the repository or added your code to the workspace you will want to build the libraries and compile the programs you got. To do this it is sufficient to type `gmake` in the top directory of your checkout area (`$SCRATCH/myrelease`). The command will first analyse the dependencies as defined by the use of `#include` in the source files, then create the libraries and finally the binary targets defined in the various GNUmakefiles of your checkout area. You can give additional options for the `gmake` command:

- `gmake lib` will directly create all libraries
- `gmake bin` will directly create all executables
- `gmake tbin` will directly create all test executables. (Each package has in the subdirectory `test` software to check the proper functioning of itself. These programs are normally used by package or subsystem developers for quality control.)
- `gmake Examples.bin` will directly create all executables in the subsystem Examples
- `gmake Calorimetry.lib` will directly create all libraries in the subsystem Calorimetry

The last two version are the fastest if you just did a little modification to your code and want to recompile it. **Currently the build system does not rebuild executables automatically when a library needed for it has changed. You then have to delete the executable by hand and `gmake` again.**

The libraries will be put in the directory `$SCRATCH/myrelease/lib/`*`ID`*`/`, where *`ID`* is specific for a particular machine and compiler. The executables and test executables are stored in the corresponding `bin/`*`ID`*`/` directory. You can find out about the current *`ID`* by typing `echo $BFARCH`.

There is however another way to build if you are only interested in a single package of a single subsystem: `lgmake`. To use this command you have to preset additional environment variables by issuing the command `srt_top`[2] in the top directory of your checkout area (`$SCRATCH/myrelease`). Now you can go to the source code directory of the subsystem and package you want to work on (`cd Examples/workspace/src`) and create all binaries there with the command `lgmake`. The executables will be created in the same directory as with `gmake`.

If you want to build only one particular executable, just give the name as an argument to `lgmake` (e.g. `lgmake ExHistograms`). Then however the executable will be put in the current directory.

The use of `lgmake` is not recommended at the moment, since it might not be available in future versions of SRT. Especially if you work only with the workspace and link against the central libraries even the speed of `gmake` and `lgmake` is comparable.

4.2 Running an executable

ORCA programs use several environment variables to select input files and to configure the number of events to process, pile-up events to use, etc.. Before you can run an executable you have to set some or all of these variables. For that purpose simple scripts are provided in the `src` subdirectories of various packages, choosing reasonable

[2] Only available in `csh`, `tcsh`. If you want to use `lgmake` in `bash`,`ksh` or `zsh` set the environment variable `SRT_TOP` to the top directory of your checkout area.

values and linking to existing example input files. You can easily use them by typing for example
`source $SCRATCH/myrelease/Examples/workspace/src/env115.csh`
or
`. $SCRATCH/myrelease/Examples/workspace/src/env115.sh`.
After this you can start the executable you want to run. The environment variables set are listed in Table 1. The

Variable	Default	Description
G3CARF_MAXEVENTS	-1	Number of trigger events to process
G3CARF_MINBUNCH	999	Lower bunch in pile-up
G3CARF_MAXBUNCH	-999	Upper bunch in pile-up
G3CARF_AVEEV	0	Average (Poisson distr.) number of pile-up events to be read for each bunch crossing. To be calculated for each luminosity.
G3CARF_CMSIMDEBUG	0	Debug level for CMSIM routines
DETINPUT		Input geometry
EVNINPUT		Input trigger events
PUINPUT		Input pile-up events

Table 1: ORCA environment variables

variables G3CARF_MINBUNCH and G3CARF_MAXBUNCH are supposed to reflect the (gate) time when the detector accepts signals. Trigger events are the signal events you want to trigger on, while pile-up events are minimum bias events. If you trigger on minimum bias events (signal = background) beware of double counting.

5 A simple example analysis

In this section a simple example analysis will be discussed. The example demonstrates how to tell ORCA to build clusters in the electro-magnetic calorimeter, how to access the information (E, η, ϕ) of the clusters and how to fill some histograms. The source code is contained in three files in `Examples/Histograms/src`: `ExHistograms.cpp`, `EcalHistos.cc` and `EcalHistos.h`. The first file contains the actual analysis code, in the other two a histogram class is defined. To understand the examples the user should have some basic familiarity with the C++ programming language (see [6] for courses).

5.1 Basic building blocks of an ORCA application

The bare minimum to create an ORCA application are three methods (C++ for subroutines/functions):

`void G3Main::construct()`
This method is called at the beginning of the running. Here the dictionary of reconstructors is build and the user analysis class(es) are instantiated (C++ for called into existence). If there is more than one analysis class and one depends on another they have to be instantiated in the correct order. `UserHists` is a pointer to an object of class `EcalHistos`, `UserAnalysis` is a pointer to an object of class `EcalClusterInfo` (see below). In the most trivial case only the call to create the dictionary would be present.

```
void G3Main::construct() {
  // some printout of your input files
  cout << "G3 Interface:\n   reading geometry ";
  cout << "from file " << getenv("DETINPUT");
  cout << "\n   events from " <<  getenv("EVNINPUT");
  cout << "\n   pile-up from " <<  getenv("PUINPUT") << endl;

  // Create the dictionary
  Reconstructor::buildDict();

  // Instantiate the user histogram (capitals in filenames are ignored by ZEBRA)
  UserHists = new EcalHistos("exhistogram.hbook");
  //  UserHists = new EcalHistos(); // default filename is ecalhistos.hbook

  // instantiate the user analysis observer
  UserAnalysis = new EcalClusterInfo();
}
```

`void G3Main::destruct()`
In this method the analysis class or classes are deleted. Here the correct order is even more important. In this example the destructor of the analysis class will fill summary information into histograms. Consequently one has to first delete the analysis class and only afterwards delete the histogram class.

```
void G3Main::destruct() {
  // cleanup the user analysis observer
  delete UserAnalysis;
  // Cleanup the Histograms - here they will be saved to file
  // You need the histograms in the destructor of the observer!!
  delete UserHists;
}
```

`void Reconstructor::buildDict()`
This method gives the possibility to set certain options. In this example the digitiser and the clustering algorithm are chosen. In the most trivial case this method would be empty.

```
void Reconstructor::buildDict()
{
  string options;
  vector<const CaloAlgo<CaloCluster,CaloDigi>* > algos2test;

  cout << "================================================" << endl;
  cout << "=== Start building dictionary                ===" << endl;

  // Create Calorimeter digitizer - select where you want to get the digis
  // see Calorimetry/G3EcalDigits/interface/
  addDefault(new CaloDigiManager(new EcalDigitsFromCMSIM,options));

  // Create Calorimeter clusterer (algo is EcalFixedWindow)
  // the clusterer wants to know where to get the DIGIs from. This is
  // given as an option
  // see Calorimetry/EcalClusters/interface/
  options="Calo_Digi_Manager";
  addDefault(new CaloClusterManager(new EcalFixedWindow,options));

  // List all defined RecUnits
  debugDict();
  cout << "=== End building dictionary                  ===" << endl;
  cout << "================================================" << endl;
}
```

5.2 User-provided classes

In addition, to do any kind of analysis one needs an observer class and to save histograms (or ntuples) a histogramming class. An observer is an object that will be automatically notified if certain things happen. Here, the analysis class inherits from the class `ActiveObserver` to become an observer. `ActiveObserver` is templated[3] and here precisely it will be an observer of objects `G3EventProxy`, i.e. events produced bye GEANT3. This results in the analysis class being notified when ORCA has read a new event and handed over a pointer to this event. Here is the class definition:

```
class EcalClusterInfo : private ActiveObserver<G3EventProxy *> {
public:
  // constructor
  EcalClusterInfo();
  // destructor
  ~EcalClusterInfo();
  // this method will do the user analysis
```

[3] C++ terminus. If something is templated it works with different types or objects in the same way. E.g. it works with `int`, `float` or like here with pointers to different kinds of classes.

```
  void myAnalysis(G3EventProxy * ev);

private:
  // don't change the name "upDate" - this method is mandatory.
  void upDate(G3EventProxy * ev) { if (ev!=0) myAnalysis(ev);}

private:
  unsigned int eventsAnalysed; // just to count events that have been analysed
  unsigned int runsAnalysed;   // count the runs
  unsigned int lastrun;
};
```

The class has a constructor, a destructor and a public method `myAnalysis`. This method will actually do the analysis. Furthermore there is a private method `upDate`. This method is mandatory for `ActiveObervers` and will be called if a new event is processed by ORCA. Here it will then just call the analysis method of the class, provided the pointer to the event is not the NULL pointer. To do some statistic of the events processed some private data members are defined. Since they are private they can only be modified by methods of our analysis class.

Let's now look at the implementation of the methods - first the constructor. It is automatically called when a new instance of the class is created. Therefore one can do here the necessary initialisations.

```
EcalClusterInfo::EcalClusterInfo() {
  cout << "===========================================================" << endl;
  cout << "=== Start create new EcalClusterInfo                    ===" << endl;
  eventsAnalysed = 0;
  runsAnalysed = 0;
  lastrun = 0;
  cout << "=== Done create new EcalClusterInfo                     ===" << endl;
  cout << "===========================================================" << endl;
}
```

Similarly the destructor is automatically called whenever an instance of the class is deleted. Consequently one can do here some final calculations and print a summary.

```
EcalClusterInfo::~EcalClusterInfo() {
  cout << "===========================================================" << endl;
  cout << "=== Start delete EcalClusterInfo                        ===" << endl;
  cout << " Number of events analysed: " << eventsAnalysed << endl;
  cout << " Number of runs analysed:   " << runsAnalysed << endl;
  UserHists->FillEventStat(eventsAnalysed,runsAnalysed);
  cout << "=== Done delete EcalClusterInfo                         ===" << endl;
  cout << "===========================================================" << endl;
}
```

Some of the summary information is filled into a histogram in this case. To avoid a crash of a program the histogram object must still exist at that point. This is the reason why in `G3Main::destruct` first the instance of the analysis class and only then that of the histogram class is deleted.

Finally follows the method where the analysis is done:

```
void EcalClusterInfo::myAnalysis(G3EventProxy * ev) {

  int nClusters=0; // just to count the clusters

  // We want to loop over all calorimetric clusters -> get an iterator.
  // The iterator needs to know about the event to use and where to ask
  // for the clusters.
  RecItr<CaloCluster> MyCluster(ev->recEvent(),"Calo_Cluster_Manager");

  // Just print the event number (see CARF/G3Event/interface/G3EventHeader.h)
  cout << "===========================================================" << endl;
  cout << "=== Private analysis of event #"<< ev->id().eventInRun()
```

```
      << " in run #" << ev->id().runNumber() << endl;

  eventsAnalysed++; // some statistics: count events and runs processed
  if (ev->id().runNumber() != lastrun) {
    lastrun = (unsigned int) ev->id().runNumber();
    runsAnalysed++;
  }

  // Here is the loop over all clusters
  while (MyCluster.next()) {
    nClusters++; // count the clusters
    // Print some of the cluster properties
    // see Calorimetry/CaloCommon/interface/CaloCluster.h
    cout << "Cluster " << nClusters <<": E=" << MyCluster->Energy()
         << ", eta="<< MyCluster->Eta()
         << ", phi="<< MyCluster->Phi() << endl;
    // Fill them to (our) histograms. Defined in EcalHistos.h
    UserHists->FillEepCluster(MyCluster->Energy(), MyCluster->Eta(),
                              MyCluster->Phi());
  }
  // Fill (our) histogram for statistics
  UserHists->FillNCluster(nClusters);

  cout << "============================================================" << endl;
};
```

The important point here is the use of an iterator, `RecItr`. Again this is a template - here it is used for objects of the class `CaloCluster`. It reduces loops over all clusters to the simple statement
`while (MyCluster.next()) {} .`
Then member functions of the `CaloCluster` class are used to access information about the current cluster, like its energy, E, (`Energy()`), its pseudo-rapidity, η, (`Eta()`) and its azimuthal angle, ϕ, (`Phi()`). This information is filled into histograms with member functions of the `EcalHisto` class.

The last missing part is the histogramming class. Here the definition:

```
class EcalHistos : private HB4Histogrammer {
 public:
  // The default constructor with the default arguments. Missing
  // arguments will be added automatically with the value given here
  EcalHistos(const char * filename = "ecalhistos.hbook", int lun=10)
    : HB4Histogrammer(filename, lun)
    { BaseInit(); }
  // The destructor
  ~EcalHistos() {  BaseEnd();}

  // Public methods to access the user histograms
  void FillEepCluster(float e, float eta, float phi);
  void FillNCluster(int n);
  void FillEventStat(unsigned int events, unsigned int runs);

 private: // private methods both are mandatory
  virtual void book();
  virtual void hend();

 public:  // public data is forbidden by CMS coding rules

 private: // private data
  // Since they are private, histograms are accessable only via public methods
  HP1  hEcal1,hEcal1b, hEcal2, hEcal3, hEcal4, hGlobal;
  HP2  hEcal5;
};
```

The class inherits its general functionality from the `HB4Histogrammer` class. This is taking care of the ZEBRA bookkeeping (no more `hlimit`) and the file handling. The constructor of `EcalHistos` has as arguments a filename and a unit where to open the file. Sensible default values are provided. These are automatically used when the constructor is called with one or both arguments left out.

Three public methods are defined to provide access to the (private) user histograms. The use of public class data is prohibited by the CMS coding rules[7], so one cannot fill histograms directly. There are two mandatory private methods: `book()` and `hend()`. They are required by `HB4Histogrammer` and called in the constructor and destructor of the class, respectively. Finally a number of pointers to histograms are defined. `HP1` and `HP2` are abbreviations for pointers to one- and two-dimensional histograms, defined in `EcalHistos.h`.

The source code for the methods defined in `EcalHistos.h` is located in `EcalHistos.cc`. Here is the booking of the histograms done:

```
void EcalHistos::book() {
  cout << "============================================================" << endl;
  cout << "=== Start booking user histograms                        ===" << endl;

  //                             ID   Name               Nbins  Min   Max
  hEcal1  = HP1(new CHBookHisto (100, "Ecal:E(cluster) ",  100,   0., 100.  ));
  hEcal1b = HP1(new CHBookHisto (101, "Ecal:E(cluster) ",   50,   0., 500.  ));
  hEcal2  = HP1(new CHBookHisto (200, "Ecal:n cluster"   , 101, -0.5, 100.5 ));
  hEcal3  = HP1(new CHBookHisto (300, "Ecal:eta(cluster)", 100, -2.5, 2.5   ));
  hEcal4  = HP1(new CHBookHisto (400, "Ecal:phi(cluster)", 100,   0., 3.15  ));
  hEcal5  = HP2(new CHBookHisto2(500, "Ecal:eta_vs_phi (cluster)",
                                 100, -2.5, 2.5, 100, 0., 3.15 ));
  hGlobal = HP1(new CHBookHisto (  1, "Events/Runs",         2,  0.5, 2.5   ));

  cout << "=== Done booking user histograms                         ===" << endl;
  cout << "============================================================" << endl;
}
```

The arguments needed to define a histogram are the same as in FORTRAN/HBOOK commands. If the histogram ID is omitted an ID will be generated automatically.

From the destructor of the class `hend()` will be called. Here one can do some printout or final manipulations of the histograms.

```
void EcalHistos::hend() {
  cout << "============================================================" << endl;
  cout << "=== Start cleanup user histograms                        ===" << endl;

  hEcal1->print();
  hEcal1b->print();
  hEcal2->print();

  cout << "=== Done cleanup user histograms                         ===" << endl;
  cout << "============================================================" << endl;
}
```

The histograms are filled using the public methods. These methods can include type conversions and range checkings.

```
// fill the histos with Energy, Pseudorapidity and Azimuthal angle
void EcalHistos::FillEepCluster(float energy, float eta, float phi) {
  hEcal1->fill(energy);
  hEcal1b->fill(energy);
  hEcal3->fill(eta);
  hEcal4->fill(phi);
  hEcal5->fill(eta,phi);
}

// fill the number of Ecal clusters
void EcalHistos::FillNCluster(int ncl) {
```

```
  hEcal2->fill((float) ncl);
}

// fill the number of events and runs analysed
void EcalHistos::FillEventStat(unsigned int events, unsigned int runs) {
  hGlobal->fill(1.,(float) events);
  hGlobal->fill(2.,(float) runs);
}
```

5.3 Adapting the `GNUmakefile`

Let's now have a look at the `GNUmakefile` used to compile and link the program. Only the parts in boldface are particular for this example.

```
#############################################################
# include file products
INC =

# library product
```

LIB = libExHistograms.a

```
SHAREDLIB =

# library contents
skip_files :=
LIBCCFILES  = $(filter-out $(skip_files), $(wildcard *.cc))

# subdirectories
SUBDIRS =

# programs to build
```

BINS = **ExHistograms** ExCompTwoClusterers

```
ifneq ($(BFARCH),Linux2-EGCS)
  override BINS += ExHistoEcalTracker
endif
############################################################
include SoftRelTools/standard.mk
```

override LOADLIBES += -lExHistograms

```
override LOADLIBES += -lEcalClusters -lG3EcalDigits -lCaloCommon
ifneq ($(BFARCH),Linux2-EGCS)
override LOADLIBES += -lCARFTracker -lTkExamples -lCommonPattern \
                      -lTkCommonDet -lTkCmsimInterface -lTkGeometry \
                      -lSiStripDet -lSiPixelDet -lGasDet -lCommonStripDet \
                      -lTkCommonDet -lTkUtilities -lSiStripDet -lSiPixelDet \
                      -lCommonStripDet -lTkCommonDet -lTkUtilities -ltrak -lgeane
override LOADLIBES += -lCARFTracker -lTkExamples -lCommonPattern \
                      -lTkCommonDet -lTkCmsimInterface -lTkGeometry \
                      -lSiStripDet -lSiPixelDet -lGasDet -lCommonStripDet \
                      -lTkCommonDet -lTkUtilities -lSiStripDet -lSiPixelDet \
                      -lCommonStripDet -lTkCommonDet -lTkUtilities -ltrak -lgeane
endif
override LOADLIBES += -lG3Main -lG3Event -lReco -lG3SimEvent -lG3Interface
override LOADLIBES += -lGenUtil -lCHBook4
include SoftRelTools/arch_spec_ObjectSpace.mk
include SoftRelTools/arch_spec_CLHEP.mk
include SoftRelTools/arch_spec_cmsim.mk
```

```
include SoftRelTools/arch_spec_cern.mk
include SoftRelTools/arch_spec_f77.mk
include SoftRelTools/arch_spec_X11.mk
include SoftRelTools/arch_spec_XDR.mk
```

Because the histogramming class `EcalHistos` is defined in separate source files we need to create a library. It will be called `libExHistograms.a` and all `*.cc` files will be compiled and put into this library. Were all the code in a single source file, this would not be necessary.

Next the target program is defined in the line starting with `BINS =`. `ExHistograms` is added to compile the file `ExHistograms.cpp`.

Finally to link against the library `libExHistograms.a` – since it is containing our histogram class – this library is added to the link command with a line starting with `override LOADLIBES`.

That's it! All that is left is to issue `gmake` to create the program, set the environment and run it. A normal HBOOK file called `exhistogram.hbook` will be created. It contains the histograms and can be used with PAW.

References

[1] http://cmsdoc.cern.ch/cmsoo/projects/ORCA/

[2] http://cmsdoc.cern.ch/cmsim/cmsim.html

[3] CMS-IN 1999/001, Object Oriented Reconstruction for CMS Analysis, CMS Software and Computing Group

[4] http://cmsdoc.cern.ch/cmsoo/CVS.pdf

[5] http://cmsdoc.cern.ch/cmsoo/taggin1.html

[6] http://cmsdoc.cern.ch/cmsoo/training.html

[7] CMS note 1998/071, CMS Coding rules, J.-P. Wellisch