ICEDUST Conventions: Summary

v1.6, Updated: August 8, 2013

1 Framework Structuring

1.1 Projects

A project is a high-level grouping of packages, based on the packages' purpose, such as offline software, online monitors, data distribution, event display and so on.

```
PROJECTNAME/ -- packages/ --------- package1/
     |            |---------------- ....
     |
     |--------- mgr/ -------------- CMakeList.txt
     |            |---------------- ProjectConfig.cfg
     |
     |--------- documentation/ ---- cometDocOutput/
     |            |---------------- Conventions.pdf
     |            |---------------- ....
```

1.2 Packages (upon being checked-out, before building)

```
PACKAGENAME/ -- versionstring/ -- cmake/ (currently cmt)
                  |-------------- dox/
                  |-------------- README
                  | * optional *
                  |-------------- src/ (source code, header & implementation files )
                  |-------------- scripts/ ( stand alone utility scripts )
                  |-------------- app/ ( source for executables )
                  |-------------- constants/ ----- calibration_tables/
                  |                  | ------------- particle_distributions/
                  |
                  |-------------- configurations/ -- parameters.txt
                  |                  | -------------- run.cfg
                  |
                  | * external included packages contain a source tar file *
                  |-------------- package1_v1.tar.gz
```

2 Project and Package Naming (see explanation 2)

The five package functions are: Base, Sim, Calib, Recon, Analysis. *FUNCTION* in the names below should be replaced with one of these.

- Projects: *CamelCase*Project, eg OfflineProject.
- Meta Packages: comet*FUNCTION*, eg cometSim, cometBase, cometAnalysis
- Base packages: oa*CamelCase*, eg oaEvent, oaAnalysis
- High-Level physics packages: *FunctionCamelCase*, eg AnalysisTools, SimG4, CalibGlobal
- Included externals: UPPERCASE, eg ROOT, MYSQL
- Pure framework packages: Icedust*CamelCase* eg IcedustPolicy, IcedustDoc

3 Repository

- A version string contains the version, release, and an optional patch number, eg v2r1 v2r1p2.
- External included packages may have an additional number to signify local patches, eg v5r34p01n02.
- All branches must be documented with a ticket on the Trac site, whose number should be appended to the version string in the format: v0r0_t00, eg. v2r3_t127.
- Commits should only be made to the trunk, or a branch of a package or project, and not a tag.

4 Quality control (see explanation 4)

- New packages and projects must be accepted by the software group before being added to the repository.
- New package names must be based on these guidelines.

5 Geometry Names and Numbers

- When code refers to a component of the geometry, it must use the relavant naming convention.
- Numbering of repeated components must agree with the numbering used in the real, physical system (eg. ECal crystals, Drift Chamber wires).

6 Filenames and Extensions (see explanation 6)

- Filenames should match contained classes (or the main, public class)
- Implementation files' names should only differ from the header file by their extension.
- File extensions :

	C	C++
Header	*filename*.h	*filename*.hxx
Implementation	*filename*.c	*filename*.cxx

	Python	Perl
Module	*filename*.py	*filename*.pl
Script {	*filename*	*filename*
	filename.py	*filename*.pl

7 Scripting Conventions (see explanation 7)

For all scripts who are directly executable:

python	`#!`	`/usr/bin/env python2`
shell	`#!`	`/bin/bash`
perl	`#!`	`/usr/bin/env perl`

8 Executable Naming

- For executables providing standard functionality of a package, the name should be of the form *FunctionPackage* eg. RunSimG4, RunCalibApply, TestOAEvent, TestReconGlobal.
- The current list of these standard executable functionalities is: Run, Test, Validate.
- All other executables should use the package name followed by an underscore as a prefix: oaEvent_dumpevent.

9 C++ Coding Conventions (see explanation 9)

9.1 Class Conventions

	Prefix	Example	Comments
Standard classes	I	IExampleClass	
Abstract classes	IV	IVExampleBaseClass	
Exception classes	E	EExampleException	Must inherit (indirectly) from std::exception
Class data members	f	fMax, fDataPoint	Avoid public access, use private or protected
Class Methods	Capital	GetMax(), TwoFlushDump()	
Destructors			Always set as virtual
Sub-Class names		IMainClass::SubClass	If behaviour mimics an STL type, then name similarly eg. IMainClass::iterator

9.2 Namespaces

- Put all classes in the COMET:: namespace, although sub-namespaces can be used, eg: COMET::ECAL::.
- 'using namespace' directives must not be included in header files

9.3 Free Functions, Variables and Arguments

All functions, variables and arguments that are not part of a class (ie. free) must NOT use the class conventions, and should instead follow these:

	Prefix	Example	Comments
Functions	Lower case	`come_fly_with_me()`	underscores for long names
Variables & Parameters	Lower case	`myWay`	CamelCase for long names
Global Parameters	`g`		Avoid at all costs!!
enums	k	kSomeEnum	CamelCase for long names

9.4 Include Guards (see explanation 9.4)

- All header files must contain an include guard, to prevent multiple inclusion of the same file.
- All include guards should take the form: `[PACKAGE]_[FILENAME]`.
- Any non-alphanumeric characters in the filename should be replaced with underscores.
- For example, for IMCHit.hxx in oaEvent, use: `OAEVENT_IMCHIT_HXX`

9.5 Output and Error Messages

- Functions from ICOMETLog.hxx should be used instead of C++ STL output (std::cout) and error (std::cerr).
- This provides standard ways for dealing with verbosity and format.

9.6 Pointer/Memory Handling (see explanation 9.6)

- Bare pointers should almost never be returned.
- Use a IHandle instead (see below).

10 Documentation

- Each packages should contain a basic README file in it's top layer, describing it's purpose, the contents of its folders and where to find documentation.
- Package histories should be documented in doxygen markup in the `dox` directory of that package.
- All header files should be fully commented in doxygen markup.
- The package IcedustDoc is used to produce documentation from the source code. Its output can either be stored in the top-level documentation directory, or alongside the individual packages.
- All stand-alone documentation must have a version number and the date last updated just below the title.

11 Units and Constants

- HEPUnits.hxx defines the standard set of units and all derived ones and should be used for all output data.
- HEPConstants.hxx defines a set of useful constants, such as pi, the speed of light, standard temperature etc.
- Both these files are in the oaEvent package and all units are contained in the 'unit' namespace.
- The standard set of units is:

millimeter (mm)	positron charge (eplus)	luminous intensity (candela)
nanosecond (ns)	degree Kelvin (kelvin)	radian (rad)
Mega electron Volt (MeV)	the amount of substance (mole)	steradian (steradian)

ICEDUST Conventions: Explanation

It is common place in large scale computing projects to introduce coding conventions that standardize the way code looks, and how it is structured. No more is this as important as in particle physics computing projects where the software has to handle extremely large volumes of data. By agreeing a set of conventions and ensuring we all use them, code becomes more intelligible and potentially more efficient.

Difficulties in physics computing:

- Constant turn over of new physicists using the software
- Lot's of developers around the world with restricted communication
- Some developers have little experience of computing and software development

General Explanation:

The coding conventions for ICEDUST are largely built around the ROOT conventions, given that ROOT is so heavily used in the code. It also includes the experience and wisdom acquired from working on the nd280 framework, and so in many cases the conventions are similar.

The section numbers below match the corresponding section number from the summary.

2 Package naming

Tight package naming conventions were not well applied in the nd280 framework. This made the structure of the framework quite messy and difficult to understand for a newcomer.

The meaning of the five package categories is:

Meta Packages These packages serve only to cluster other packages together for the framework. They serve as a collection that can be used by CMake to build or check-out subsections of the framework. As a result, they contain little, if any, code or scripts of their own and probably only a CMakeList.txt file.

Base packages These are 'low-level' software elements and as such are used widely throughout the code. Examples are file format management packages, like oaEvent, oaRawEvent and oaAnalysis. In principle these packages could be reused for another experiment, because they contain very little COMET specific information.

High-Level physics packages These are the packages used to run the actual software and, together, provide the main set executables. They are the source of the physics information produced by the framework. This category contains 4 subdivisions, based on their functionality: Sim, Calib, Recon and Analysis.

Included externals These packages wrap external libraries and software into the framework. Upon checking out the package, there should only be a tar file and possibly a few patch files and scripts.

Pure framework packages These packages are framework management packages, with no knowledge of the experiment or it's data whatsoever. They purely provide tools for the framework to be controlled, such as repository access, package versioning, documentation building and so on.

4 Quality control

Whilst it is important to allow flexibility for everyone that works on the framework to work and produce as they feel necessary, the addition of a new package or project to the framework is big enough that we would like to have the project or package proposed and discussed before anything is committed. This should keep the framework tidier and better managed than otherwise.

6 Filenames and extensions

A python or perl script file is an executable file (has the executable bit set) and therefore contains a shebang on it's first line. Whether or not to use an extension for these files is left open to the developer.

7 Scripting Conventions

"`#!/bin/bash`" is the standard bash shebang. "`#! /bin/sh`" is bad because it could call a variety of shells.

Until recently, just "`#! /usr/bin/env python`" almost always gave you python 2 as the interpreter. Some systems are now beginning to use python3 as the default python, and require you to specify `python2`, if that's what you wanted. ICEDUST only supports Python 2, so all scripts should explicitly state this, in case they are used on a machine where python 3 is default.

9 C++ Coding Conventions

Packages that depend predominantly on external libraries may differ from these conventions in order to match the external library's conventions, eg. cometMC uses the geant4 conventions.

In general you should use the conventions defined by ROOT for c++ code (see `http://root.cern.ch/drupal/content/c-coding-conventions`). The conventions above take precedence over ROOT so that our conventions should be used instead of the ROOT one, where a conflict arises (eg. our code uses 'I' as a class prefix, whereas ROOT use 'T').

9.4 Include Guards

Include guards prevent a file being included twice, which would result in multiple definitions of the contained functions, classes and so on. Such a situation could easily arise if, for instance, a program included two files, but the second file itself already includes the first one. The typical solution (and the one used in ICEDUST) is to encase all the code in a header file in a set of preprocessor macros as such:

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD
//
// Code goes here
//
#endif  /* INCLUDE_GUARD */
```

9.6 Pointer and Memory Handling

Efficient memory management is very important when dealing with large amounts of data. Operations on pointers to deleted objects will cause runtime failures, such as seg faults, that could lose any processed data that was not yet saved. Similarly many calls to 'new' and 'delete' would result in slow code.

To solve this, the CHandle template class provides a 'smart' pointer that implements reference counting, so that the object it represents is only deleted once all its references are out of scope. This is great if several calling functions need to be able change the object because they can use it without worrying who 'owns' the memory that contains the object. See the oaEvent manual for more information on the CHandle class.

If on the other hand you only want a read-only, immutable version of the object, then a const pointer can be used instead of a CHandle. Information on const pointers and const_pointer casting in c++ is readily available online.

ICEDUST Conventions: Discussions

A On-going Discussions:

B Resolved discussions:

B.1 Project conventions

Use of Projects

Based on purpose:

- The suggested use of projects is to group packages by purpose (ie. Offline software, online monitoring, data distribution etc).
- In principle, a project would have very little dependency on another project, as there would be less need to interface between projects directly.
- The same package may be contained in two projects, (eg. the Online monitor project, which may need older versions of the same packages as the offline software).
- The way a project is managed in the repository would need consideration.
- Allowing people to checkout a subset of the packages within a project requires thought, and probably an extra script or two.

Based on functionality:

- We could alternatively group by functionality (ie. Analysis, Simulation, Reconstruction etc).
- This approach makes projects much more dependent on each other, and such relationship would need to be carefully defined.
- Packages such oaEvent and oaAnalysis that are used by many parts of the framework would want to go into a separate project.
- Therefore, Projects would depend directly on code and libraries found in other projects.
- Meta-packages could be removed completely.

RESOLUTION: Projects based on Purpose will be used

Top-level bin directory

- With this, the expectation was to simplify the PATH variable, and tidy up how executables are managed within a project.
- How do you manage packages with multiple versions stored concurrently?
- What happens when two executables have the same name?
- RESOLUTION: Top-level bin has been removed from the project layout

Top-level documentation directory

- Collects documentation in a directory at the top of the project, rather than storing it alongside the individual packages.
- Documents such as this one would be contained in there.
- In nd280, the package nd280Doc (now renamed to cometDoc) contains all the documentation, and is also responsible for running doxygen to produce the main documentation of the code.
- Documentation produced from code, via doxygen, would also be stored in this directory.
- What happens if there are two versions of a package stored concurrently? Where does that documentation get stored?
- RESOLUTION: Top-level documentation directory will be kept and cometDoc will be given an option to place output either alongside a package, or in this top-level directory.

B.2 Package conventions

Packages mainly involving external libraries

- Generalize statement that cometMC follows geant4 conventions in place of normal FRAMEWORK ones.
- All packages that depend predominantly on one external library should follow it's conventions.
- RESOLUTION: Now mention that all packages largely dependent on externals may differ to these conventions (previously, only mentioned cometMC).

doc/ dox subdirectory

- Discussion on whether this is necessary.
- RESOLUTION: Use 'dox' (change from doc).

High-level naming

- High level packages are currently subdivided into 4 categories functionality they provide (analysis, simulation etc). The recommended name for one of these packages is cometFunctionName, ie cometSimMC for the monte carlo package, cometCalibApply.
- Does the comet prefix serve much purpose now we have the function after it?
- Without this, names would be shorter and simpler.
- RESOLUTION: Remove comet as a high-level package prefix

Project naming

- Projects are named with Icedust as a prefix. This clashes with framework management packages ie. IcedustOffline is a project, IcedustPolicy is a framework management package.
- RESOLUTION: Projects should use just Project as a suffix: eg. OfflineProject

dataFiles, controlFiles subdirectory

- Suggested that 'Files' is redundant.
- Maybe dataFiles -> staticData
- Tidy up the various kinds of runtime input and output files found in the framework.
- Data files to come from databases
- RESOLUTION: dataFiles → constants
- RESOLUTION: controlFiles → configurations

B.3 Scripting Conventions

Python Shebangs

- Do we want to require every python file to contain a shebang?
- As the 'file' command would use the shebang to identify the type of code, this will help this command out.
- At the moment the conventions state that only files intended to be executable should have a shebang.
- RESOLUTION: Only require shebangs in files that are to have their executable bit set.

Scripting Shebangs

- Should make sure people use `#! /bin/bash` and not `#! /bin/sh` or otherwise.
- RESOLUTION: Specify the shebang to be used for all major scripting languages

B.4 C++ conventions

Class data member prefix

- Alternatives to `f: '_', 'm_'`
- fData, _Data, m_Data
- _ may be confusable with system/ internal variables.
- RESOLUTION: Have stuck with f prefix.

Class and function prefixes

- Move away from 'T' for classes ?
  - Could help distinguish our code from ROOT code
  - Suggestion to use 'C'
- Files containing single 'free' functions should not use the class prefix (ie. 'T', 'C' etc).
- Namespaces should help distinguish the source of the code (COMET:: means it's not from ROOT or the STL etc)
- The mixin prefix 'TM' should be dropped - use documentation in stead.
- RESOLUTION: Switched from T prefix to C
- RESOLUTION: The mixin prefix 'TM' has been dropped - use documentation in stead.
- C clashes with COMET, which is prefixed to many classes.
- RESOLUTION: Switch to I instead of C

File extensions

- Implementation files should only ever differ from the header by the file extension.
- Using .hxx and .cxx allows shell globbing unlike .hh and .cc
- RESOLUTION: Made explicit how implementation files must have the same name as the header file.
- Add a convention for the file extension of c++ program files (files that contain the 'main' function used to create an executable)?
- RESOLUTION: Leave this out

Global variables / singleton classes

- Singleton classes were overused or misused in nd280
- Conventions should be careful of this.
- RESOLUTION: Suggest example classes should be used to write new classes.

Getters / setters policy

- Emphasize that thy should only be written when necessary
- If they don't act as regular getters or setters then a different name should be used.
- RESOLUTION: Suggest example classes should be used to write new classes.

Include statement placing

- Should we specify where these should be?
- All in header, or as few as possible in header?
- RESOLUTION: Left out of conventions

Comment formatting

- Develop a format for comments ?
- Coder's initials could be used to indicate who wrote what, and to make statements about future intentions
- Svn and Git both provide mechanisms to annotate code lines with the author, so probably not necessary in most cases.
- RESOLUTION: Ignored from Conventions

Code style

- Should code be automatically checked and reformatted, on commit?
- Programs such as astyle, http://sourceforge.net/projects/astyle/?source=navbar, could be used.
- RESOLUTION: Left out of conventions

Namespace

- At the moment all code is in a namespace called COMET.
- Should this now be ICEDUST or ID now that we've chosen ICEDUST as the framework name?
- RESOLUTION: Leave namespace as COMET.

Standard Output and Error

- TCOMETLog (in oaEvent) provides a standardised way to control all outputs from all packages
- Provides verbosity control and so on
- Therefore should not use std::cout or std::cerr and instead use these methods.
- RESOLUTION: Include statement of this in document

THandle and TBorrowed

- Only mention THandle
- This section needs expanding

B.5 Geometry naming and numbering

- Components that require numbering should make sure the numbering used in the code is consistent with the numbering used in real life.
- RESOLUTION: Added section that states this.

B.6 Units

- Do we wish to standardize the unit system employed as done in Geant4?
- Getters returning lengths, material densities, energies etc would all need to follow these guidelines.
- Do we instead just rely on clear documentation of each method?
- RESOLUTION: Mention using the HEPUnits.hxx class from oaEvent for unit standardisation.

B.7 Naming conventions for executables

- If a package has a 'run' executable it should name it in a standardised way.
- Likewise for 'testing' executables and any other common package applications.
- RESOLUTION: Add executable naming conventions to the document.

B.8 Other

- Capitalisation needs clarity in the conventions
- Give conventions for constants and enums, eg kSomeValue or kSOME_VALUE ? RESOLUTION: Have been added.