SLAC
Stanford Linear Accelerator Center

GLAST offline software

Recommendations for documentation

Document Version:	0
Document Date:	18 May 2000
Document Status:	Draft
Document Author:	THK, PK, ..

Abstract
This note gives recommendations for documentation, the expected content, the access to documentation and the tools to be used to produce the documentation.

This is a living document which will evolve with experience. Please send comments to hansl@slac.standord.edu.

1 Introduction

Types of documentation

We envisage different types of software documentation

- requirements and design documents together with justification of choices made; made available before writing the code
- inline documentation of declaration and implementation files
- user guides, which explain how to set up the environment, describe the basic functionality and use the software
- reference guides, which explain technical details
- documentation and evaluation of the algorithms

The limits between these different types of documentation are not sharp. Obviously there may be some overlap.

2 Documentation of ongoing work

It is very useful to build up documentation as the work progresses. The easiest way to make information accessible to others is via web.

Plots can be accumulated on web pages together with short explanations. Avoid fancy icons, they take time to display and have little information. A simple way to present information on the web: create with suggestive names the directories for the projects, on which you work. In each of these directories provide the files README, `index.html` (optional) and (depending on your installation) a file which makes the directory browsable. The content of the directory will be shown, when the directory is opened from netscape. The README file appears on top of the file directory. If you want only a subset of files to be displayed, provide the file index.html in the usual html style. Title lines in html files are displayed in the directory list.

To prepare a publication or note, collect the figures, and text snippets and figure captions in `.txt` or .RTF format for portability. It is useful to follow standard names for some of the directories. For example, create a directory '/imported', in which figures are collected, preferably in .eps format and following an agreed standard (figure size, font, line thickness etc, see Appendix xxx for an example for PAW figures). Names for the directories for figure captions and text snippets are proposed in the table below. ><

File or directory	Comments
README	Comments displayed above directory list
index.html	html selected list (optional)
.htaccess (at CERN)	makes directory browsable, installation-dependent
	Publication in preparation:
/imported/*.eps	• directory with figures
/captions/*.txt or *.rtf	• figure captions
/notes/*.txt or *.rtf	• text snippets

A central web page may point to the developer pages. Alternatively links may be established from the agenda and minutes pages.

3 Inline documentation

This Chapter gives recommendations for inline documentation and introduces the tool doxygen, which we have chosen for extraction of documentation from the code.

- Doxygen can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LATEX) from a set of documented source files.
- The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

- Doxygen can be configured to extract the code structure from (undocumented) source files. The relations between the various elements are visualized by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
- The documentation of large packages can be broken into smaller pieces using dogygen's tag file mechanism.

Doxygen is a freely available tool(GNU General Public License). It is developed on Linux, but runs on most other UNIX flavours. Furthermore, an executable for Windows 9x / NT is available.

3.1 Inline documentation of header files

The documentation is extracted from the declaration files (e.g. *.h). The text to be extracted has to be enclosed by /** ..*/. Any other documentation should start with '//'. For this reason implementation files (e.g. *.cpp) should contain only '//' comments.

><

```
/** this is a comment to be included in the extracted documentation */

// this comment will not be extracted by doxygen
```

Comments must be written before the class declaration, the member function declaration or data member declaration. The first sentence of a comment block will be used for short overviews. The complete text is displayed in the long description of the class or member.

We recommend to mark a block of comments by a vertical sequence of '*'s, see the example in Listing 1>lis_header<. A few stylistic remarks to the examples: all code is indented (by 2 spaces), with respect to the C++ access keywords 'public', protected' and 'private'. Use blank lines to separate different themes in the code, it helps readability.

Doxygen understands keywords and selected HTML commands. Listing 1 >lis_header< shows the most useful ones. Using too many of the keywords and HTML commands tends to obscure the code.

The example of the header in Listing 1 is organised in four groups of informations:

- Headers which are needed for the class declaration and the forward declaration of classes. Note that there should be no /** ..*/ comments. The comment '//' blocks are shown, even when the corresponding declarations are missing, to show the order in which the declarations should be made.
- The class description together with the constructor and destructor. Note the use of the keywods @see, @version, @author. The argument for @version is automatically provided by CVS through xxx.

- Examples of methods. The methods should be declared in the order of increasing restriction of access, public > protected > private. For large classes repeat the access specifiers for groups of member declarations, to improve readability. Note the use of @param and @return keywords, which allow to describe the function arguments and the return type.
- Private members and member functions.
- The example follows also the coding rules described in >ref_codingStandard<.

><>lis_header<

Listing 1 Example of a header file of class Abcd

```
#ifndef ABCD_H
#define ABCD_H

//-----------------------------------------------------------------------
//
// Environment:
//   Domain within which this class(es) operate
//-----------------------------------------------------------------------

//-------------
// C Headers --
//-------------
extern "C" {
}

//---------------
// C++ Headers --
//---------------

//----------------------
// Base Class Headers --
//----------------------

//-------------------------------
// Collaborating Class Headers --
//-------------------------------
#include "AbcCommon/AbcDefg.h"

//------------------------------------
// Collaborating Class Declarations --
//------------------------------------
class SomeClass;
class AnotherClass;

//               ---------------------
//               -- Class Interface --
//               ---------------------

```

Listing 1 Example of a header file of class Abcd

```

/**
 *  One short sentence with essential description.
 *  Followed by the details of the description, as many sentences as
 *  may be needed.
 *  Followed by few key words (@see, @version, @author)
 *
 *  @see AbcdManager
 *
 *  @version $Id$ (RCS keyword -> filename,revision,date,author,state)
 *
 *  @author Some Body  (originator)
 */

class Abcd {

protected:

  Abcd( SomeClass theClass );

public:

  virtual ~Abcd();


 /**
   *  Appends an item to an open record by copying.
   *
   *  @param theItem  Item to be appended; will be copied.
   *  @return         Zero if item was appended, non-zero otherwise.
   *                  Return codes are as in XyzErrors.
   */
  virtual int append( const odfXTC& theItem );

protected:

  /**
   *  Internal, protected version of Method1.
   */
  virtual int internalMethod1( SomeClass* theItemH ) = 0;

  /**
   *  Internal, protected version of build.
   */
  virtual AnotherClass* internalBuild() = 0;


```

Listing 1 Example of a header file of class Abcd

```
private:

  bool m_aParameter;

  // Disable copy constructor and assignment operator. Do not implement.
  Abcd( const Abcd& );
  Abcd& operator=( const Abcd& );

};

#endif // ABCD_H
```

3.2 Inline documentation of implementation files

As mentioned above, comments are not extracted from implementation files. It is still useful to provide '//' style comments and provide the implementations in a sequence close to the sequence of the class declaration. An example is shown in Listing 2>lis_implementation<.

>< >lis_implementation<

Listing 2 Example of documentation (not extracted by doxygen) and sequence of statements for an implementation

```
//----------------------------------------------------------------------
// File and Version Information:
//      $Id$
//
// Description:
//      Abcd implementation
//
// Author List:
//      Some Body (originator)
//
//----------------------------------------------------------------------
#include "Glast/Glast.h"

//-----------------------
// This Class's Header --
//-----------------------
#include "AbcdPackage/Abcd.h"

//-------------
// C Headers --
//-------------
extern "C" {
}

//---------------
// C++ Headers --
//---------------

//-------------------------------
// Collaborating Class Headers --
//-------------------------------
#include "AbcdPackage/SomeClass.h"
#include "AbcdPackage/AnotherClass.h"


//----------------------------------------------------------------------
// Local Macros, Typedefs, Structures, Unions and Forward Declarations
//----------------------------------------------------------------------

```

Listing 2 Example of documentation (not extracted by doxygen) and sequence of statements for an implementation

```

//              ----------------------------------------
//              -- Public Function Member Definitions --
//              ----------------------------------------

//----------------
// Constructors --
//----------------
Abcd::Abcd( SomeClass theClass )
 : m_aParameter(true)
{
}

//--------------
// Destructor --
//--------------
Abcd::~Abcd()
{
}

//-------------
// Methods   --
//-------------

int
Abcd::append( const SomeClass& theClass )
{
  ....
  return status;
}

//              -----------------------------------------------
//              -- Static Data & Function Member Definitions --
//              -----------------------------------------------

//              -------------------------------------------
//              -- Protected Function Member Definitions --
//              -------------------------------------------


```

3.3 Structuring the documentation according to packages

If doxygen is run on a full code release, all output files are put into one directory, ./output/html. To avoid this and produce structured output, yet keep the hyperlinks working, the tag file mechanism should be used.

We want to structure the document according to packages. This is achieved by using the `TAGFILES` and `GENERATE_TAGFILE` options in the configuration file. Generate one tag file (`GENERATE_TAGFILE`) for each package, which you wish to link to from within another package. Use `TAGFILES` to include the generated tag files of all ‘external’ packages. Note that the ‘uses’ statements in the CMT requirements file list the dependencies of a package. In case of cyclic package dependencies (which should be avoided anyhow) doxygen may have to be run twice to do the necessary bootstrapping.

3.4 Remember to update documentation

The importance of comments cannot be over-emphasized. Yet there is only one thing worse than an undocumented source file: a misleadingly documented source file. Often such out-of-sync comments render software components unusable. Whenever you change your code, remember to update the comments appropriately or remove them altogether (from >ref_<

3.5 Things to discuss/decide

1. Production of documentation. The documentation of the official software should be updated with each new release by a person responsible for the documentation (e.g. the software librarian).
2. The developer is responsible for the documentation of his code. The documentation of a package is the responsibility of the group or person, who is responsible for the package.
3. Documentation is reviewed together with the code. It is recommended to involve someone, who is not directly working on the development of the package, to help in the review.
4. The proposed format of header and implementation files

4 User guide

5 Documentation of algorithms

6 References

1 Doxygen, http://www.doxygen.org

2 C++ Coding Standard, Specification, CERN-UCO/1999/207,
>ref codingStandard<
http://pst.web.cern.ch/PST/handBookWorkBook/HandBook/Programming/CodingStandard/c++standard.pdf

3 ATLAS modifications to the above reference >ref codingStandard<
http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/qc/QC_CodingRules.html

4

To...	Do..	More info...
Introduce yourself to FrameMaker...	a. Follow the FrameMaker Overview b. Follow a Quick Tour c. Follow the Online Tutorials	Do Help > Overview... from any doc window Go to http://framemaker.cern.ch/, “Getting Started” (for both b. and c.)
Access available FrameMaker documentation...	a. Access the Online Help b. Obtain a printed copy of the FrameMaker 5.5 User Guide	Do Help > Contents... Buy it from COBS (http://consult.cern.ch/books/), or borrow a copy from 1-R-013
Access SDLT documentation...	Read: *Guide for using the Software Documentation Layout Templates*	http://framemaker.cern.ch/sdlt/guide/ Hard copies from 1-R-017
Edit the title block and the document metadata...	See Section 7, "Editing the title block & the document metadata", below	WARNING! Do not replace the metadata user variables with text!