	Document #	Date
LAT TECHNICAL NOTE	**LAT-TD-xxxxx-xx**	May 21, 2002
	Author(s)	Supersedes
	System or Sponsoring Office: Science Analysis Software	

Document Title

SAS Recommendations for Code Documentation

CHANGE HISTORY LOG

Revision	Effective Date	Description of Changes
0.7	March 11, 2003	HMK: Updated release.notes and implementation recommendations according to THB's suggestions
0.6	May 21, 2002	HMK: Add text on jobOptions section in mainpage file.
0.5	Mar 11, 2002	HMK: Separated into 3 distinct documents and updated code recommendations according to TkrRecon Review findings.
0.4	Jan. 23, 2002	HMK: Re-ordered the subsections in the code documentations section. Inserted location for storing images within a package. Fixed up the examples to be more consistent.
0.3	Jan. 18, 2002	HMK: Separated into 4 documents - one master and 3 subdocuments. Modified code recommendations: Inserted example implementation comment block and inserted new section concerning the ChangeLog.
0.2	Jan. 17, 2002	MSS: Modified Code Documentation Introduction to include goals of and guidelines for code documentation.
0.1	Jan. 15, 2002	HMK: Inserted new section on Requirement and Design Documents. Converted examples to use JavaDoc style Doxygen comments.
0.0	Jan. 9, 2002	initial version

1. ABSTRACT

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

2. DEFINITIONS

SAS – Science Analysis Software
TBD – To Be Determined

3. INTRODUCTION

4. REQUIREMENTS AND DESIGN DOCUMENTS

[TBD by the Documentation Task Force]

5. CODE DOCUMENTATION

5.1 Introduction

This section gives recommendations for inline documentation and introduces the tool Doxygen, which we have chosen for extraction of documentation from the code. Code documentation will take two forms: That used within Doxygen, which will be primarily for the benefit of code users; and that using standard C++ comment lines, which will be for the benefit of code maintenance and development.

5.1.1 Doxygen

The goals of the Doxygen based comments include:

- Supplying an automatically extractable summary of all classes and modules.
- Creating an encyclopedic users reference in easily viewed formats.
- Allowing developers and maintainers an overall view of the software systems.

In order to accomplish these goals, Doxygen has the following capabilities:

- Doxygen can generate online HTML documentation and/or an offline reference manual in LaTeX (convertible into hyperlinked PDF) 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 classes are visualized by means of include dependency

graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

Doxygen is a freely available tool (GNU General Public License). It is available on most UNIX flavors and Windows. GLAST SAS will note which version of Doxygen is the current GLAST standard.

5.1.2 Standard C++ Comments

C++ comment documentation should accomplish the following goals:

- Allow maintenance of code by people not involved in its development.
- Allow understanding of requirements, inputs and outputs, algorithms, and software design techniques, when reviewing code.
- Allow understanding of code structure for purposes of reusability i.e. make explicit all data structures and external code required to reuse a particular class, method, etc.

Code authors and code reviewers should keep these goals in mind explicitly while writing and reviewing GLAST software systems. Coders should always be asking themselves:

- "How will a future user or maintainer deal with this software after I am sleeping with the metaphorical fish?"
- "Have I done anything that, to me, seems clever and elegant, but to a follow-on coder will seem mostly obscure?"
- "Have I implemented an algorithm with no reference to its origins or just what it actually does?"
- "Have I so welded my code into the system framework that it is impossible to extricate with the information at hand?"

Code documentation can solve all these potential problems and more, both for users and developers.

5.2 Code Documentation Components

5.2.1 Mainpage

Doxygen document extraction will produce a series of HTML pages, including an index page. The content of the index page is determined by a file called mainpage.h. This mainpage.h file is **required** for all packages and should reside in the package *src* directory. Note that class names, which appear in the file mainpage.h will be hyperlinked to the corresponding class description. The file mainpage.h is therefore a perfect place to provide an overview of the functionality of the package. A meaningful mainpage.h should be part of every new package import. For packages whose documentation is in need of an upgrade, tackle mainpage.h first. It has the form shown in Listing 1. It is GLAST convention to add to the package description the package's requirements, which show the dependencies on other packages. It is also useful to keep a list of "todo" items on this page.

Any jobOptions parameters defined by the package will be documented in the mainpage file in a section called "jobOptions". Each parameter will be documented using the Doxygen @param

keyword. This provides useful formatting as well as a hook for scripts to search the mainpage file to extract the jobOptions parameters. See Listing 1 for an example of the jobOptions section.

Listing 1 Example mainpage.h file

```
/** @mainpage package templates
 *
 * @authors Documentation Task Force
 *
 * @section intro Introduction
 * Include here an overview of the package: outline its main structure, the principal services it provides and
 * comment  on dependencies.
* If you desire to insert a link to a web page do
* <A HREF="http://www-glast.slac.stanford.edu/software/CodeHowTo/codeStandards.html"> Standards </A>
*
* <hr>
* @section jobOptions jobOptions
* @param ExampleAlg.property
*   an example input integer parameter for a user-defined Gaudi algorithm
*   default value zero 0
* @param ExampleSvc.property
*   an example input integer parameter for a user-defined Gaudi service
*
* <hr>
* @section notes release.notes
* release.notes
* <hr>
* @section requirements requirements
* @verbinclude requirements
* <hr>
* @todo  [optionally include text  about more work to be done]
* @todo  Give each todo item its own line
*
*/
```

5.2.2 Inline documentation of header files

It is recommended that all Doxygen style comments reside within the header files, with the class declarations. The only exceptions occur at the top of implementation files and when declarations appear in the implementation file. The text to be extracted has to be enclosed by Doxygen style comments. Two formats are supported:

- JavaDoc style, where comment blocks look like
  /**
  * Comment text
  */
  and the one line version
  /// text
- Qt style, where comment blocks look like
  /*!
  Comment text
  */
  and the one line version
  //! text

Other documentation not meant for Doxygen processing should use the standard C++ comments '/* .. */' or '//'. In general, implementation files (*.cxx) should contain only standard C++ comments.

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

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

Doxygen style comments should be written before the class declaration, the member function declaration, or data member declaration whenever the function of these items is not immediately obvious from their declarations. It is **mandatory** that all class declarations be immediately preceded by a Doxygen style comment block. This class comment block should contain an overview of the class. The first sentence of a comment block will be used for short overviews.

In order to maximize the amount of useful information visible per unit area of a documentation page, we recommend setting off comment blocks using white space, rather than divider lines.

Again, in order to conserve screen space, do not document obvious methods or members. It is not instructive to document a default constructor with a comment such as "default constructor". Similarly, get and set routines do not require comments and in fact a comment in this case only serves to make the declarations more difficult to read. Comments should be inserted when the developer feels there is something that requires an explanation, that will make the code more understandable.

Doxygen understands keywords and selected HTML commands. Listing 2 shows the most useful ones. Detailed information concerning Doxygen usage is available in the Doxygen user manual available from the Doxygen web site http://www.doxygen.org. Please note that the comments bracketed by [..] are present to provide additional information, and are not intended to appear in an actual header file. The example follows the GLAST SAS coding rules described in http://www-glast.slac.stanford.edu/software/CodeHowTo/codeStandards.html

It is suggested to keep the documentation within the code as simple as possible. If one wishes to insert an image in the Doxygen documentation, this can be accomplished using a command of the form:
@image <format> <file> ["caption"]
where <format> may be html or latex. All images will be stored in the package's *doc/images* directory.

The example header in Listing 2 is organized into four sections:

- Headers which are needed for the class declaration and the forward declaration of classes.
- The class description is **required**. Note the use of the keyword @author. The version is automatically provided by CVS through the RCS keyword $Header$.
- 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 describe the function arguments and return type.
- Private members and member functions

Listing 2 Example header file displaying comments and sequence of statements

```
/**
 * @class ClassTemplate
 *
 * @brief This is an example class.
 *
 * This comment block is @e required for all classes.
 * If you desire to provide a link to a web page do
 * <A HREF="http://www-glast.slac.stanford.edu/software" GLAST Home Page</A>
 *
 * @author Some Body
 *
 * $Header$
 */

class ClassTemplate {

public:

   ClassTemplate();

   ~ClassTemplate();

   int getIntMember() { return m_intMember; };
   void setIntMember(const int i) { m_intMember = i; };

   /**
    * Provide detailed desciption of this function
    * @param parmeter1 Describe this parameter
    */
   void publicMemberFunction(int parameter1);

   /**
    * Provide a detailed description of this function.
    * @return Describe the possible return values.
    */
   bool anotherPublcMemberFunction();

   static int getStaticIntMember() { return m_staticIntMember; };

private:

   /// Provide a description of this class member [note that the m_ prefix is not used for static members]
   static int s_staticIntMember;
   /// Provide a description of this class member
   int m_intMember;
   /// Provide a description of this class member
   float m_floatMember;

}

#endif // CLASSTEMPLATE_H
```

5.2.3 Inline documentation of implementation files

As mentioned in the preceding section, Doxygen comments are reserved for the top of implementation files, all other comment blocks will be standard C++ comments. The C++ comments will explain the details of the implementation. In general, '//' comments are preferred over '/* .. */' comments since '/* .. */' blocks cannot be nested. Each implementation file should begin with a standard Doxygen comment block containing the file identifier and a short description, an example is provided in Listing 3. Please note that comments bracketed by [..] are present for informational purposes and should not appear in actual implementation files.

Listing 3 Example source file containing standard comments and sequence of statements

```
/** @file ClassTemplate.cxx
 *   @brief Class Template provides an example of code documentation for a class.
 *   @author InsertYourNameHere
 *    $Header$
 */

// [Header files]
#include "PackageName/ClassTemplate.h"

// [Declare static members]
int ClassTemplate::m_staticIntMember;

            // [ PUBLIC MEMBER FUNCTION DEFINITIONS ]

// [Constructors]
ClassTemplate::ClassTemplate() {
}

// [Destructors]
ClassTemplate::~ClassTemplate() {
}

void ClassTemplate::publicMemberFunction(int parameter1) {
  // Purpose and Method:  [include the physics if appropriate]
  // Inputs
  // Outputs
  // TDS Inputs
  // TDS Outputs
  // Dependencies:
  // Restrictions and Caveats:  e.g. "This doesn't work off-axis"

   // [Code inside function should be documented using standard C++ comments]
}

            // [ STATIC MEMBER FUNCTION DEFINITIONS ]

            // [ PRIVATE MEMBER FUNCTION DEFINITIONS ]
```

Listing 3 displays the use of a standard C++ comment within the implementation file. Each implementation file begins with a header block. Then within each important member function, a comment block should appear of the form:

```
// Purpose and Method:  [include the physics if appropriate]
// Inputs
// Outputs
// TDS Inputs
// TDS Outputs
// Dependencies:
// Restrictions and Caveats:  e.g. "This doesn't work off-axis"
```

The "Purpose and Method" section should describe what a method does and how it does it. If the method in question is just an interface to another routine that does the actual work - this should be clearly stated. The Inputs and Outputs refer to input and output parameters. Meanwhile, TDS Inputs and TDS Outputs refer to items retrieved from and put on the Gaudi Transient Data Store. The Dependencies section denote required conditions, such as algorithm X must have executed before this routine can be called. Finally, "Restrictions and Caveats" refer to special conditions, such as this routine will provide inaccurate results in the input tracks are off-axis. If a particular section has not useful information in relation to the method in question, it may be omitted. Please note, that for simple member functions, one may just provide a purpose. If the function is small (i.e. one line) and the purpose is clear - then no comment block is required.

Such comments will provide developers and future code maintainers the information they need to understand the code. Additional C++ comments should be inserted as well to point out interesting features of the routine.

5.2.4 Release Notes

Release notes are maintained for each package, in a text file called "release.notes". This file is located within the package's *doc* directory. This file should be updated when noteworthy modifications are made to a package and when new tags are created. Listing 4 provides a release.notes example from the TkrRecon package. Note that the release.notes are embedded within a Doxygen style comment block. This allows for easy reference to the release notes from the mainpage.h file.

Listing 4 Example release.notes file

```
/** @file release.notes
 * @brief Package TkrRecon
 * @verbatim
 * Coordinator: Leon Rochester
 *
 * v4r4p8  09-Mar-2002  LSR   Remove GFxxxxx and SiRecObjs, no longer used
 * v4r4p7  07-Mar-2002  TU    Mainly, add a combo vertexing to the TkrRecon sequence
 * @endverbatim
 */
```

5.2.5 Log Messages and ChangeLog

When performing a CVS check-in, CVS requires a log message. This log message is to contain details pertaining to the modifications made. The contents of these messages are automatically logged to a file named ChangeLog, stored at the top-level of a package.

The ChangeLog will be used as our modifcation history. It is imperative that the information stored in the ChangeLog is detailed enough to provide useful information for future developers and maintainers of the package. Those performing code check-ins should provide meaningful log statements. One may further modify the ChangeLog file and check it back into the CVS repository, just as any other file within a package. This allows the developer to consolidate or further annotate the comments.

5.2.6 Doxygen Document Extraction

A standard configuration file, Doxyfile, may be used to set the parameters in use when performing document extraction using Doxygen. The standard GLAST Doxyfile is stored in the GlastPolicy package. This Doxyfile will be used for all SAS Doxygen extraction. Developers who desire to modify the Doxygen configuration, may provide a Doxyfile in the package's *doc* directory. However, this should only be done when there is good reason to do so. When a local Doxyfile exists in the package, its contents are appended to the standard Doxyfile.

The standard directory for running Doxygen will be the package's *doc* directory. This is important as the Doxyfile may specify some paths that are relative to the directory where Doxygen is running.

The official production of Doxygen pages will be performed in some TBD time interval. The Doxygen output will be made web-accessible. LaTeX will be available for official Doxygen processing, allowing images to be imported and mathematical formulae to be properly processed.

[The hows and whens of Doxygen document extraction should be inserted here - is this done automatically for all tagged checkout packages? What about individual packages?]

5.2.7 Indentation and Line Width

As specified in the GLAST coding standards, blanks should be inserted rather than tabs for indentation. Please be sure your editor is set appropriately. Consistent indentation should be applied to all header and sources files. This provides readable files, as well as more polished looking Doxygen output. Be aware that automatic formatting may or may not behave as expected. There are instances where indentation may be applied inconsistently, such as for enum declarations. It is up to the developer to double-check their files to be sure that the indentation is consistent.

According to the GLAST coding standards:
"Lines should normally be no wider than 80 characters, but in circumstances where this doesn't suffice (e.g., long literal strings), keep it to 132."
This becomes especially important when processing source files using Doxygen. Otherwise, Doxygen pages containing source code will wrap off the browser's screen. It is strongly recommended that all source files contains line no wider than 80 characters. Note the exception for

long literal strings such as the file and version tag inserted to files via the $Header$ keyword - in most instances, this string will extend beyond 80 characters.

5.2.8 Updating Documentation

The importance of comments cannot be overemphasized. Yet there is only one thing worse than undocumented code: misleadingly documented code. Often such out-of-sync comments render software components unusable. Whenever you modify your code, remember to update the comments appropriately or removed them altogether. You should also be aware that when you comment out a piece of code, but leave the comments, the Doxygen short comment will be associated to the wrong piece of code.

6. SUMMARY OF RECOMMENDATIONS

Code Documentation

- Maintain a mainpage.h file in the package's *src* directory. This file contains a general description of the package and is used as the index page for Doxygen output.
- Each mainpage.h file will contain a section called "jobOptions". This section will list all input parameters defined in the package.
- All class declarations must be preceded by a Doxygen style comment block describing the class.
- Doxygen style comments immediately precede class, method, and member declarations.
- Doxygen style comments normally reside within the header file before declarations. When there is no header file, Doxygen comments reside with the declarations within the implementation file.
- Implementation files contain standard C++ comments, where the form '//' is preferred over '/* .. */'. Each important member function should contain a standard comment block of the form:

```
// Purpose and Method:  [include the physics if appropriate]
// Inputs
// Outputs
// TDS Inputs
// TDS Outputs
// Dependencies:
// Restrictions and Caveats:  e.g. "This doesn't work off-axis"
```

  Sections that are not pertinent to the method may be omitted. If the method in question is merely an interface to another routine, this should noted in the Purpose and Method section.
- Do not bother to document obvious members or methods such as default constructors, get and set routines, or Gaudi required routines - unless there is something special going on that deserves documetation.
- Use white space to set off comment blocks, rather than divider lines.
- Maintain a release.notes text file in the package's *doc* directory noting major revisions and tags.
- When checking in code to the CVS repository, provide modification details within the log message. The log messages are automatically stored within a ChangeLog file and will be used as the modification history.

- If images are inserted into the Doxygen comments, the image files should be stored within the package's *doc/images* directory.
- Make sure indentation is consistent in all source and header files. Spaces, rather than tabs should be inserted. Note that automatic formating may not behave as expected, developers are expected to double check the final form of their code to be sure it adheres to the standard.
- Lines should be restricted to 80 characters per line in general, and no more 132 characters for long literal strings.
- Keep comments up-to-date when modifying code.

7. REFERENCES

1. Doxygen, http://www.doxygen.org/
2. GLAST SAS Coding standards, http://www-glast.slac.stanford.edu/software/CodeHowTo/codeStandards.html/