How to report a bug

Bugs are tracked in GNOME's bugzilla database. Before submitting a new bug, first search through the database if the same bug has already been submitted by others (the doxygen product will be preselected). If you believe you have found a new bug, please report it.

If you are unsure whether or not something is a bug, please ask help on the users mailing list first (subscription is required).

If you send only a (vague) description of a bug, you are usually not very helpful and it will cost me much more time to figure out what you mean. In the worst-case your bug report may even be completely ignored by me, so always try to include the following information in your bug report:

- The version of doxygen you are using (for instance 1.5.3, use `doxygen --version` if you are not sure).
- The name and version number of your operating system (for instance SuSE Linux 6.4)
- It is usually a good idea to send along the configuation file as well, but please use doxygen with the `-s` flag while generating it to keep it small (use `doxygen -s -u [configName]` to strip the comments from an existing config file).
- The easiest (and often the only) way for me to fix bugs is if you can attach a small example demonstrating the problem you have to the bug report, so I can reproduce it on my machine. Please make sure the example is valid source code (could potentially compile) and that the problem is really captured by the example (I often get examples that do not trigger the actual bug!). If you intend to send more than one file please zip or tar the files together into a single file for easier processing. Note that when reporting a new bug you'll get a chance to attach a file to it only *after* submitting the initial bug description.

You can (and are encouraged to) add a patch for a bug. If you do so please use PATCH as a keyword in the bug entry form.

If you have ideas how to fix existing bugs and limitations please discuss them on the developers mailing list (subscription required). Patches can also be sent directly to dimitri@stack.nl if you prefer not to send them via the bug tracker or mailing list.

For patches please use "diff -uN" or include the files you modified. If you send more than one file please tar or zip everything, so I only have to save and download one file.

Part II

Reference Manual

Features

- Requires very little overhead from the writer of the documentation. Plain text will do, but for more fancy or structured output HTML tags and/or some of doxygen's special commands can be used.
- Supports C/C++, Java, (Corba and Microsoft) Java, Python, IDL, C#, Objective-C and to some extent D and PHP sources.
- Supports documentation of files, namespaces, packages, classes, structs, unions, templates, variables, functions, typedefs, enums and defines.

Manual for version 1.5.6

Written by Dimitri van Heesch

©1997-2007

Contents

I User Manual 4

1 Compiling from source on Unix 4

2 Installing the binaries on Unix 6

3 Known compilation problems for Unix 6

4 Compiling from source on Windows 8

5 Installing the binaries on Windows 9

6 Tools used to develop doxygen 10

7 Step 1: Creating a configuration file 11

8 Step 2: Running doxygen 12

9 Step 3: Documenting the sources 14

10 Special documentation blocks 15

11 Putting documentation after members 19

12 Documentation at other places 20

13 Special documentation blocks in Python 23

14 Special documentation blocks in VHDL 24

15 Modules 26

16 Member Groups 29

17 Subpaging 30

18 Links to web pages and mail addresses 37

19 Links to classes. 37

20 Links to files. 38

21 Links to functions. 38

- Since it is impossible to test all possible code fragments, it is very well possible, that some valid piece of C/C++ code is not handled properly. If you find such a piece, please send it to me, so I can improve doxygen's parsing capabilities. Try to make the piece of code you send as small as possible, to help me narrow down the search.
- Doxygen does not work properly if there are multiple classes, structs or unions with the same name in your code. It should not crash however, rather it should ignore all of the classes with the same name except one.
- Some commands do not work inside the arguments of other commands. Inside a HTML link (i.e <a href="...">...<a>) for instance other commands (including other HTML commands) do not work! The sectioning commands are an important exception.
- Redundant braces can confuse doxygen in some cases. For example:

```
void f (int);
```

  is properly parsed as a function declaration, but

```
const int (a);
```

  is also seen as a function declaration with name `int`, because only the syntax is analysed, not the semantics. If the redundant braces can be detected, as in

```
int *(a[20]);
```

  then doxygen will remove the braces and correctly parse the result.
- Not all names in code fragments that are included in the documentation are replaced by links (for instance when using `SOURCE_BROWSER = YES`) and links to overloaded members may point to the wrong member. This also holds for the "Referenced by" list that is generated for each function.

  For a part this is because the code parser isn't smart enough at the moment. I'll try to improve this in the future. But even with these improvements not everything can be properly linked to the corresponding documentation, because of possible ambiguities or lack of information about the context in which the code fragment is found.
- It is not possible to insert a non-member function f in a class A using the \relates or \relatesalso command, if class A already has a member with name f and the same argument list.
- There is only very limited support for member specialization at the moment. It only works if there is a specialized template class as well.
- Not all special commands are properly translated to RTF.
- Version 1.8.6 of dot (and maybe earlier versions too) do not generate proper map files, causing the graphs that doxygen generates not to be properly clickable.
- PHP only: Doxygen requires that all PHP statements (i.e. code) is wrapped in a functions/methods, otherwise you may run into parse problems.

How to help

The development of Doxygen highly depends on your input!

If you are trying Doxygen let me know what you think of it (do you miss certain features?). Even if you decide not to use it, please let me know why.

the built-in preprocessor converted it into an empty file (with >256K of newlines). Another case where this might happen is if you have lines in your code with more than 256K characters.

If you have run into such a case and want me to fix it, you should send me a code fragment that triggers the message. To work around the problem, put some line-breaks into your file, split it up into smaller parts, or exclude it from the input using EXCLUDE.

14. **When running make in the latex dir I get "TeX capacity exceeded". Now what?**

    You can edit the texmf.cfg file to increase the default values of the various buffers and then run "texconfig init".

15. **Why are dependencies via STL classes not shown in the dot graphs?**

    Doxygen is unware of the STL classes, unless the option BUILTIN_STL_SUPPORT is turned on.

16. **I have problems getting the search engine to work with PHP5 and/or windows**

    Please read `this` for hints on where to look.

17. **Can I configure doxygen from the command line?**

    Not via command line options, but doxygen can read from `stdin`, so you can pipe things through it. Here's an example how to override an option in a configuration file from the command line (assuming a unix environment):

```
( cat Doxyfile ; echo "PROJECT_NUMBER=1.0" ) | doxygen -
```

    If multiple options with the same name are specified then doxygen will use the last one. To append to an existing option you can use the += operator.

18. **How did doxygen get its name?**

    Doxygen got its name from playing with the words documentation and generator.

```
documentation -> docs -> dox
generator -> gen
```

    At the time I was looking into lex and yacc, where a lot of things start with "yy", so the "y" slipped in and made things pronounceable (the proper pronouncement is Docs-ee-gen, so with a long "e").

19. **What was the reason to develop doxygen?**

    I once wrote a GUI widget based on the Qt library (it is still available at `http://qdbttabular.sourceforge.net/` and maintained by Sven Meyer). Qt had nicely generated documentation (using an internal tool which they didn't want to release) and I wrote similar docs by hand. This was a nightmare to maintain, so I wanted a similar tool. I looked at Doc++ but that just wasn't good enough (it didn't support signals and slots and did not have the Qt look and feel I had grown to like), so I started to write my own tool...

Troubleshootinh

Known problems:

- If you have problems building doxygen from sources, please read this section first.
- Doxygen is *not* a real compiler, it is only a lexical scanner. This means that it can and will not detect errors in your source code.

22 Links to variables, typedefs, enum types, enum values and defines. 39

23 typedefs. 40

24 Output Formats 41

25 Simple aliases 42

26 Aliases with arguments 42

27 Nesting custom command 43

II Reference Manual 50

28 Format 59

29 Project related options 62

30 Build related options 65

31 Options related to warning and progress messages 67

32 Input related options 68

33 Source browsing related options 69

34 Alphabetical index options 70

35 HTML related options 70

36 LaTeX related options 73

37 RTF related options 74

38 Man page related options 75

39 XML related options 75

40 AUTOGEN_DEF related options 76

41 PERLMOD related options 76

42 Preprocessor related options 76

43 External reference options 77

44 Dot options 77
45 Search engine options 79
46 Introduction 81
47 \addtogroup <name> [(title)] 84
48 \callgraph 84
49 \callergraph 84
50 \category <name> [<header-file>] [<header-name>] 85
51 \class <name> [<header-file>] [<header-name>] 85
52 \def <name> 85
53 \defgroup <name> (group title) 86
54 \dir [<path fragment>] 86
55 \enum <name> 86
56 \example <file-name> 87
57 \file [<name>] 87
58 \fn (function declaration) 88
59 \headerfile <header-file> [<header-name>] 88
60 \hideinitializer 89
61 \ingroup (<groupname> [<groupname> <groupname>]) 89
62 \interface <name> [<header-file>] [<header-name>] 89
63 \internal 89
64 \mainpage [(title)] 89
65 \name (header) 90
66 \namespace <name> 90

or you can use `installdox` to set the links as follows:

```
installdox -lb.tag@b.chm::
```

7. **I don't like the quick index that is put above each HTML page, what do I do?**

   You can disable the index by setting DISABLE_INDEX to YES. Then you can put in your own header file by writing your own header and feed that to HTML_HEADER.

8. **The overall HTML output looks different, while I only wanted to use my own html header file**

   You probably forgot to include the stylesheet `doxygen.css` that doxygen generates. You can include this by putting

   ```
   <LINK HREF="doxygen.css" REL="stylesheet" TYPE="text/css">
   ```

   in the HEAD section of the HTML page.

9. **Why does doxygen use Qt?**

   The most important reason is to have a platform abstraction for most Unices and Windows by means of the QFile, QFileInfo, QDir, QDate, QTime and QIODevice classes. Another reason is for the nice and bug free utility classes, like QList, QDict, QString, QArray, QTextStream, QRegExp, QXML etc.

   The GUI front-end doxywizard uses Qt for... well... the GUI!

10. **How can I exclude all test directories from my directory tree?**

    Simply put an exclude pattern like this in the configuration file:

    ```
    EXCLUDE_PATTERNS = */test/*
    ```

11. **Doxygen automatically generates a link to the class MyClass somewhere in the running text. How do I prevent that at a certain place?**

    Put a % in front of the class name. Like this: %MyClass. Doxygen will then remove the % and keep the word unlinked.

12. **My favourite programming language is X. Can I still use doxygen?**

    No, not as such; doxygen needs to understand the structure of what it reads. If you don't mind spending some time on it, there are several options:

    - If the grammar of X is close to C or C++, then it is probably not too hard to tweak src/scanner.l a bit so the language is supported. This is done for all other languages directly supported by doxygen (i.e. Java, IDL, C#, PHP).
    - If the grammar of X is somewhat different than you can write an input filter that translates X into something similar enough to C/C++ for doxygen to understand (this approach is taken for VB, Object Pascal, and Javascript, see http://www.stack.nl/~dimitri/doxygen/download.html#helpers).
    - If the grammar is completely different one could write a parser for X and write a backend that produces a similar syntax tree as is done by src/scanner.l (and also by src/tagreader.cpp while reading tag files).

13. **Help! I get the cryptic message "input buffer overflow, can't enlarge buffer because scanner uses REJECT"**

    This error happens when doxygen's lexical scanner has a rule that matches more than 256K of input characters in one go. I've seen this happening on a very large generated file (>256K lines), where

The new and easiest way is to add one comment block with a \cond command at the start and one comment block with a \endcond command at the end of the piece of code that should be ignored. This should be within the same file of course.

But you can also use Doxygen's preprocessor for this: If you put

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS

 /* code that must be skipped by Doxygen */

#endif /* DOXYGEN_SHOULD_SKIP_THIS */
```

around the blocks that should be hidden and put:

```
  PREDEFINED = DOXYGEN_SHOULD_SKIP_THIS
```

in the config file then all blocks should be skipped by Doxygen as long as `PREPROCESSING = YES`.

5. **How can I change what is after the `#include` in the class documentation?**

   In most cases you can use STRIP_FROM_INC_PATH to strip a user defined part of a path.

   You can also document your class as follows

```
/*! \class MyClassName include.h path/include.h
 *
 *  Docs for MyClassName
 */
```

   To make doxygen put

   `#include <path/include.h>`

   in the documentation of the class MyClassName regardless of the name of the actual header file in which the definition of MyClassName is contained.

   If you want doxygen to show that the include file should be included using quotes instead of angle brackets you should type:

```
/*! \class MyClassName myhdr.h "path/myhdr.h"
 *
 *  Docs for MyClassName
 */
```

6. **How can I use tag files in combination with compressed HTML?**

   If you want to refer from one compressed HTML file `a.chm` to another compressed HTML file called `b.chm`, the link in `a.chm` must have the following format:

```
<a href="b.chm::/file.html">
```

   Unfortunately this only works if both compressed HTML files are in the same directory.

   As a result you must rename the generated `index.chm` files for all projects into something unique and put all `.chm` files in one directory.

   Suppose you have a project *a* referring to a project *b* using tag file `b.tag`, then you could rename the `index.chm` for project *a* into `a.chm` and the `index.chm` for project *b* into `b.chm`. In the configuration file for project *a* you write:

```
TAGFILES = b.tag=b.chm::
```


67 \nosubgrouping 90

68 \overload [(function declaration)] 90

69 \package <name> 91

70 \page <name> (title) 91

71 \property (qualified property name) 92

72 \protocol <name> [<header-file>] [<header-name>] 92

73 \relates <name> 92

74 \relatesalso <name> 93

75 \showinitializer 93

76 \struct <name> [<header-file>] [<header-name>] 93

77 \typedef (typedef declaration) 93

78 \union <name> [<header-file>] [<header-name>] 93

79 \var (variable declaration) 94

80 \weakgroup <name> [(title)] 94

81 \attention { attention text } 94

82 \author { list of authors } 94

83 \brief {brief description} 95

84 \bug { bug description } 95

85 \cond [<section-label>] 95

86 \date { date description } 96

87 \deprecated { description } 96

88 \details {detailed decription} 96

89 \else 96

90 \elseif <section-label> 97
91 \endcond 97
92 \endif 97
93 \exception <exception-object> { exception description } 97
94 \if <section-label> 97
95 \ifnot <section-label> 98
96 \invariant { description of invariant } 98
97 \note { text } 99
98 \par [(paragraph title)] { paragraph } 99
99 \param <parameter-name> { parameter description } 99
100 \tparam <template-parameter-name> { description } 100
101 \post { description of the postcondition } 100
102 \pre { description of the precondition } 100
103 \remarks { remark text } 100
104 \return { description of the return value } 101
105 \retval <return value> { description } 101
106 \sa { references } 101
107 \see { references } 101
108 \since { text } 101
109 \test { paragraph describing a test case } 101
110 \throw <exception-object> { exception description } 102
111 \todo { paragraph describing what is to be done } 102
112 \version { version number } 102

In some (hopefully exceptional) cases you may have the documentation generated by doxygen, but not the sources nor a tag file. In this case you can use the doxytag tool to extract a tag file from the generated HTML sources. Another case where you should use doxytag is if you want to create a tag file for the Qt documentation.

The tool `doxytag` depends on the particular structure of the generated output and on some special markers that are generated by doxygen. Since this type of extraction is brittle and error-prone I suggest you only use this approach if there is no alternative. The doxytag tool may even become obsolete in the future.

Frequently Asked Questions

1. **How to get information on the index page in HTML?**

   You should use the \mainpage command inside a comment block like this:

```
/*! \mainpage My Personal Index Page
 *
 * \section intro_sec Introduction
 *
 * This is the introduction.
 *
 * \section install_sec Installation
 *
 * \subsection step1 Step 1: Opening the box
 *
 * etc...
 */
```

2. **Help, some/all of the members of my class / file / namespace are not documented?**

   Check the following:

   (a) Is your class / file / namespace documented? If not, it will not be extracted from the sources unless `EXTRACT_ALL` is set to `YES` in the config file.

   (b) Are the members private? If so, you must set `EXTRACT_PRIVATE` to `YES` to make them appear in the documentation.

   (c) Is there a function macro in your class that does not end with a semicolon (e.g. MY_-MACRO())? If so then you have to instruct doxygen's preprocessor to remove it.

   This typically boils down to the following settings in the config file:

```
ENABLE_PREPROCESSING   = YES
MACRO_EXPANSION        = YES
EXPAND_ONLY_PREDEF     = YES
PREDEFINED             = MY_MACRO()=
```

   Please read the preprocessing section of the manual for more information.

3. **When I set EXTRACT_ALL to NO none of my functions are shown in the documentation.**

   In order for global functions, variables, enums, typedefs, and defines to be documented you should document the file in which these commands are located using a comment block containing a \file (or @file) command.

   Alternatively, you can put all members in a group (or module) using the \ingroup command and then document the group using a comment block containing the \defgroup command.

   For member functions or functions that are part of a namespace you should document either the class or namespace.

4. **How can I make doxygen ignore some code fragment?**

If any of the above apply, you can use doxygen's tag file mechanism. A tag file is basically a compact representation of the entities found in the external sources. Doxygen can both generate and read tag files.

To generate a tag file for your project, simply put the name of the tag file after the GENERATE_TAGFILE option in the configuration file.

To combine the output of one or more external projects with your own project you should specify the name of the tag files after the TAGFILES option in the configuration file.

A tag file does not contain information about where the external documentation is located. This could be a directory or an URL. So when you include a tag file you have to specify where the external documentation is located. There are two ways to do this:

At configuration time: just assign the location of the output to the tag files specified after the TAGFILES configuration option. If you use a relative path it should be relative with respect to the directory where the HTML output of your project is generated.

After compile time: if you do not assign a location to a tag file, doxygen will generate dummy links for all external HTML references. It will also generate a perl script called installdox in the HTML output directory. This script should be run to replace the dummy links with real links for all generated HTML files.

Example:

Suppose you have a project `proj` that uses two external projects called `ext1` and `ext2`. The directory structure looks as follows:

```
<root>
  +- proj
  |   +- html               HTML output directory for proj
  |   +- src                sources for proj
  |   |- proj.cpp
  +- ext1
  |   +- html               HTML output directory for ext1
  |   |- ext1.tag           tag file for ext1
  +- ext2
  |   +- html               HTML output directory for ext2
  |   |- ext2.tag           tag file for ext2
  |- proj.cfg               doxygen configuration file for proj
  |- ext1.cfg               doxygen configuration file for ext1
  |- ext2.cfg               doxygen configuration file for ext2
```

Then the relevant parts of the configuration files look as follows:

proj.cfg:

```
OUTPUT_DIRECTORY  = proj
INPUT             = proj/src
TAGFILES          = ext1/ext1.tag=../../ext1/html \
                    ext2/ext2.tag=../../ext2/html
```

ext1.cfg:

```
OUTPUT_DIRECTORY  = ext1
GENERATE_TAGFILE  = ext1/ext1.tag
```

ext2.cfg:

```
OUTPUT_DIRECTORY  = ext2
GENERATE_TAGFILE  = ext2/ext2.tag
```


113\warning { warning message } 102

114\xrefitem <key> "(heading)" "(list title)" {text} 102

115\addindex (text) 103

116\anchor <word> 103

117\endlink 103

118\link <link-object> 103

119\ref <name> ["(text)"] 104

120\subpage <name> ["(text)"] 104

121\section <section-name> (section title) 105

122\subsection <subsection-name> (subsection title) 105

123\subsubsection <subsubsection-name> (subsubsection title) 105

124\paragraph <paragraph-name> (paragraph title) 105

125\dontinclude <file-name> 106

126\include <file-name> 106

127\includelineno <file-name> 107

128\line (pattern) 107

129\skip (pattern) 107

130\skipline (pattern) 107

131\until (pattern) 108

132\verbinclude <file-name> 108

133\htmlinclude <file-name> 108

134\a <word> 108

135\arg { item-description } 109

136\b <word> 109

137\c <word> 109

138\code 110

139\copydoc <link-object> 110

140\copybrief <link-object> 110

141\copydetails <link-object> 110

142\dot 110

143\msc 111

144\dotfile <file> [”caption”] 112

145\e <word> 112

146\em <word> 112

147\endcode 113

148\enddot 113

149\endmsc 113

150\endhtmlonly 113

151\endlatexonly 113

152\endmanonly 113

153\endverbatim 113

154\endxmlonly 114

155\f$ 114

156\f[114

157\f] 114

158\f{environment}{ 114

where the command with a single argument would still work as shown before.

Aliases can also be expressed in terms of other aliases, e.g. a new command \reminder can be expressed as a \xrefitem via an intermediate \xreflist command as follows:

```
ALIASES += xreflist{3}="\xrefitem \1 \"\2\" \"\3\" " \
ALIASES += reminder="\xreflist{reminders,Reminder,Reminders}" \
```

Note that if for aliases with more than one argument a comma is used as a separator, if you want to put a comma inside the command, you will need to escape it with a backslash, i.e.

```
\l{SomeClass,Some text\, with an escaped comma}
```

given the alias definition of \l in the example above.

27 Nesting custom command

You can use commands as arguments of aliases, including commands defined using aliases.

As an example consider the following alias definitions

```
ALIASES += Bold{1}="<b>\1</b>"
ALIASES += Emph{1}="<em>\1</em>"
```

Inside a comment block you can now use:

```
/** This is a \Bold{bold \Emph{and} Emphasized} text fragment. */
```

which will expand to

```
/** This is a <b>bold <em>and</em> Emphasized</b> text fragment. */
```

Link to external documentationIf your project depends on external libraries or tools, there are several reasons to not include all sources for these with every run of doxygen:

Disk space: Some documentation may be available outside of the output directory of doxygen already, for instance somewhere on the web. You may want to link to these pages instead of generating the documentation in your local output directory.

Compilation speed: External projects typically have a different update frequency from your own project. It does not make much sense to let doxygen parse the sources for these external project over and over again, even if nothing has changed.

Memory: For very large source trees, letting doxygen parse all sources may simply take too much of your system’s memory. By dividing the sources into several ”packages”, the sources of one package can be parsed by doxygen, while all other packages that this package depends on, are linked in externally. This saves a lot of memory.

Availability: For some projects that are documented with doxygen, the sources may just not be available.

Copyright issues: If the external package and its documentation are copyright someone else, it may be better - or even necessary - to reference it rather than include a copy of it with your project’s documentation. When the author forbids redistribution, this is necessary. If the author requires compliance with some license condition as a precondition of redistribution, and you do not want to be bound by those conditions, referring to their copy of their documentation is preferable to including a copy.

25 Simple aliases

The simplest form of an alias is a simple substitution of the form

```
name=value
```

For example defining the following alias:

```
ALIASES += sideeffect="\par Side Effects:\n"
```

will allow you to put the command \sideeffect (or @sideeffect) in the documentation, which will result in a user-defined paragraph with heading **Side Effects:**.

Note that you can put \n's in the value part of an alias to insert newlines.

Also note that you can redefine existing special commands if you wish.

Some commands, such as \xrefitem are designed to be used in combination with aliases.

26 Aliases with arguments

Aliases can also have one or more arguments. In the alias definition you then need to specify the number of arguments between curly braces. In the value part of the definition you can place \x markers, where 'x' represents the argument number starting with 1.

Here is an example of an alias definition with a single argument:

```
ALIASES += l{1}="\ref \1"
```

Inside a comment block you can use it as follows

```
/** See \l{SomeClass} for more information. */
```

which would be the same as writing

```
/** See \ref SomeClass for more information. */
```

Note that you can overload an alias by a version with multiple arguments, for instance:

```
ALIASES += l{1}="\ref \1"
ALIASES += l{2}="\ref \1 \"\2\""
```

Note that the quotes inside the alias definition have to be escaped with a backslash.

With these alias definitions, we can write

```
/** See \l{SomeClass,Some Text} for more information. */
```

inside the comment block and it will expand to

```
/** See \ref SomeClass "Some Text" for more information. */
```


159\f} 114

160\htmlonly 114

161\image <format> <file> ["caption"] [<sizeindication>=<size>] 115

162\latexonly 115

163\manonly 116

164\li { item-description } 116

165\n 116

166\p <word> 117

167\verbatim 117

168\xmlonly 117

169\\ 117

170\@ 117

171\~[LanguageId] 118

172\& 118

173\$ 118

174\# 118

175\< 118

176\> 118

177\% 118

178PHP only commands 119

III Developers Manual 124

179Using the Perl Module output format. 128

180Using the Perl Module-based LaTeX generator. 128

181Perl Module documentation format. 130

182Data structure describing the Perl Module documentation tree. 130


```
    float x;
    /*! The y coordinate */
    float y;
  };

  /*! Creates a type name for CoordStruct */
  typedef CoordStruct Coord;

  /*!
   * This function returns the addition of \a c1 and \a c2, i.e:
   * (c1.x+c2.x,c1.y+c2.y)
   */
  Coord add(Coord c1,Coord c2)
  {
  }
```

Output Formats

24 Output Formats

The following output formats are *directly* supported by doxygen:

HTML Generated if `GENERATE_HTML` is set to `YES` in the configuration file.

LaTeX Generated if `GENERATE_LATEX` is set to `YES` in the configuration file.

Man pages Generated if `GENERATE_MAN` is set to `YES` in the configuration file.

RTF Generated if `GENERATE_RTF` is set to `YES` in the configuration file.

Note that the RTF output probably only looks nice with Microsoft's Word 97. If you have success with other programs, please let me know.

XML Generated if `GENERATE_XML` is set to `YES` in the configuration file.

Note that the XML output is still under development.

The following output formats are *indirectly* supported by doxygen:

Compiled HTML Help (a.k.a. Windows 98 help) Generated by Microsoft's HTML Help workshop from the HTML output if `GENERATE_HTMLHELP` is set to `YES`.

PostScript Generated from the LaTeX output by running `make ps` in the output directory. For the best results `PDF_HYPERLINKS` should be set to `NO`.

PDF Generated from the LaTeX output by running `make pdf` in the output directory. To improve the PDF output, you typically would want to enable the use of `pdflatex` by setting USE_PDFLATEX to `YES` in the configuration file. In order to get hyperlinks in the PDF file you also need to enable PDF_HYPERLINKS.

Custom CommandsDoxygen provides a large number of special commands, XML commands, and HTML commands. that can be used to enhance or structure the documentation inside a comment block. If you for some reason have a need to define new commands you can do so by means of an *alias* definition.

The definition of an alias should be specified in the configuration file using the ALIASES configuration tag.

```
  public:
    Test();                //!< constructor
   ~Test();                //!< destructor
    void member(int);      /**< A member function. Details. */
    void member(int,int); /**< An overloaded member function. Details */

    /** An enum type. More details */
    enum EType {
      Val1,                /**< enum value 1 */
      Val2                 /**< enum value 2 */
    };

  protected:
    int var;               /**< A member variable */
};

/*! details. */
Test::Test() { }

/*! details. */
Test::~Test() { }

/*! A global variable. */
int globVar;

/*! A global enum. */
enum GlobEnum {
                GVal1,    /*!< global enum value 1 */
                GVal2     /*!< global enum value 2 */
              };

/*!
 *  A macro definition.
 */
#define ABS(x) (((x)>0)?(x):-(x))

typedef Test B;

/*! \fn typedef Test B
 *  A type definition.
 */
```

23 typedefs.

Typedefs that involve classes, structs and unions, like

```
typedef struct StructName TypeName
```

create an alias for StructName, so links will be generated to StructName, when either StructName itself or TypeName is encountered.

Example:

```
/*! \file restypedef.cpp
 * An example of resolving typedefs.
 */

/*! \struct CoordStruct
 * A coordinate pair.
 */
struct CoordStruct
{
  /*! The x coordinate */
```


Introduction

Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D.

It can help you in three ways:

1. It 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. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
3. You can even 'abuse' doxygen for creating normal documentation (as I did for this manual).

Doxygen is developed under `Linux` and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available.

This manual is divided into three parts, each of which is divided into several sections.

The first part forms a user manual:

- Section Installation discusses how to `download`, compile and install doxygen for your platform.
- Section Getting started tells you how to generate your first piece of documentation quickly.
- Section Documenting the code demonstrates the various ways that code can be documented.
- Section Lists show various ways to create lists.
- Section Grouping shows how to group things together.
- Section Including formulas shows how to insert formulas in the documentation.
- Section Graphs and diagrams describes the diagrams and graphs that doxygen can generate.
- Section Preprocessing explains how doxygen deals with macro definitions.
- Section Automatic link generation shows how to put links to files, classes, and members in the documentation.
- Section Output Formats shows how to generate the various output formats supported by doxygen.
- Section Custom Commands show how to define and use custom commands in your comments.
- Section Linking to external documentation explains how to let doxygen create links to externally generated documentation.
- Section Frequently Asked Questions gives answers to frequently asked questions.
- Section Troubleshooting tells you what to do when you have problems.

The second part forms a reference manual:

- Section Features presents an overview of what doxygen can do.
- Section Doxygen History shows what has changed during the development of doxygen and what still has to be done.
- Section Doxygen usage shows how to use the `doxygen` program.
- Section Doxytag usage shows how to use the `doxytag` program.
- Section Doxywizard usage shows how to use the `doxywizard` program.
- Section Installdox usage shows how to use the `installdox` script that is generated by doxygen if you use tag files.
- Section Configuration shows how to fine-tune doxygen, so it generates the documentation you want.
- Section Special Commands shows an overview of the special commands that can be used within the documentation.
- Section HTML Commands shows an overview of the HTML commands that can be used within the documentation.
- Section XML Commands shows an overview of the C# style XML commands that can be used within the documentation.

The third part provides information for developers:

- Section Doxygen's Internals gives a global overview of how doxygen is internally structured.
- Section Perl Module output format documentation shows how to use the PerlMod output.
- Section Internationalization explains how to add support for new output languages.

Doxygen license

Copyright ©1997-2008 by Dimitri van Heesch.

Permission to use, copy, modify, and distribute this software and its documentation under the terms of the GNU General Public License is hereby granted. No representations are made about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. See the GNU General Public License for more details.

Documents produced by doxygen are derivative works derived from the input used in their production; they are not affected by this license.

User examples

Doxygen supports a number of output formats where HTML is the most popular one. I've gathered some nice examples (see `http://www.doxygen.org/results.html`) of real-life projects using doxygen.

These are part of a larger list of projects that use doxygen (see `http://www.doxygen.org/projects.html`). If you know other projects, let me know and I'll add them.

22 Links to variables, typedefs, enum types, enum values and defines.

All of these entities can be linked to in the same way as described in the previous section. For sake of clarity it is advised to only use patterns 3 and 7 in this case.

Example:

```
/*! \file autolink.cpp
  Testing automatic link generation.

  A link to a member of the Test class: Test::member,

  More specific links to the each of the overloaded members:
  Test::member(int) and Test#member(int,int)

  A link to a protected member variable of Test: Test#var,

  A link to the global enumeration type #GlobEnum.

  A link to the define #ABS(x).

  A link to the destructor of the Test class: Test::~Test,

  A link to the typedef ::B.

  A link to the enumeration type Test::EType

  A link to some enumeration values Test::Val1 and ::GVal2
*/

/*!
  Since this documentation block belongs to the class Test no link to
  Test is generated.

  Two ways to link to a constructor are: #Test and Test().

  Links to the destructor are: #~Test and ~Test().

  A link to a member in this class: member().

  More specific links to the each of the overloaded members:
  member(int) and member(int,int).

  A link to the variable #var.

  A link to the global typedef ::B.

  A link to the global enumeration type #GlobEnum.

  A link to the define ABS(x).

  A link to a variable \link #var using another text\endlink as a link.

  A link to the enumeration type #EType.

  A link to some enumeration values: \link Test::Val1 Val1 \endlink and ::GVal1.

  And last but not least a link to a file: autolink.cpp.

  \sa Inside a see also section any word is checked, so EType,
      Val1, GVal1, ~Test and member will be replaced by links in HTML.
*/

class Test
{
```

20 Links to files.

All words that contain a dot (.) that is not the last character in the word are considered to be file names. If the word is indeed the name of a documented input file, a link will automatically be created to the documentation of that file.

21 Links to functions.

Links to functions are created if one of the following patterns is encountered:

1. <functionName>"("<argument-list>")"
2. <functionName>"()"
3. "::"<functionName>
4. (<className>"::")^n <functionName>"("<argument-list>")"
5. (<className>"::")^n <functionName>"("<argument-list>")"<modifiers>
6. (<className>"::")^n <functionName>"()"
7. (<className>"::")^n <functionName>

where n>0.

Note 1:
Function arguments should be specified with correct types, i.e. 'fun(const std::string&,bool)' or '()' to match any prototype.

Note 2:
Member function modifiers (like 'const' and 'volatile') are required to identify the target, i.e. 'func(int) const' and 'fun(int)' target different member functions.

Note 3:
For JavaDoc compatibility a # may be used instead of a :: in the patterns above.

Note 4:
In the documentation of a class containing a member foo, a reference to a global variable is made using foo, whereas #foo will link to the member.

For non overloaded members the argument list may be omitted.

If a function is overloaded and no matching argument list is specified (i.e. pattern 2 or 6 is used), a link will be created to the documentation of one of the overloaded members.

For member functions the class scope (as used in patterns 4 to 7) may be omitted, if:

1. The pattern points to a documented member that belongs to the same class as the documentation block that contains the pattern.
2. The class that corresponds to the documentation blocks that contains the pattern has a base class that contains a documented member that matches the pattern.

Future work

Although doxygen is used successfully by a lot of people already, there is always room for improvement. Therefore, I have compiled a todo/wish list (see http://www.doxygen.org/todo.html) of possible and/or requested enhancements.

Acknowledgements

Thanks go to:

- Malte Zöckler and Roland Wunderling, authors of DOC++. The first version of doxygen borrowed some code of an old version of DOC++. Although I have rewritten practically all code since then, DOC++ has still given me a good start in writing doxygen.
- All people at Troll Tech, for creating a beautiful GUI Toolkit (which is very useful as a Windows/Unix platform abstraction layer :-)
- Kevin McBride for maintaining the subversion reporsitory for doxygen.
- My brother Frank for rendering the logos.
- Harm van der Heijden for adding HTML help support.
- Wouter Slegers of Your Creative Solutions for registering the www.doxygen.org domain.
- Parker Waechter for adding the RTF output generator.
- Joerg Baumann, for adding conditional documentation blocks, PDF links, and the configuration generator.
- Matthias Andree for providing a .spec script for building rpms from the sources.
- Tim Mensch for adding the todo command.
- Christian Hammond for redesigning the web-site.
- Ken Wong for providing the HTML tree view code.
- Talin for adding support for C# style comments with XML markup.
- Petr Prikryl for coordinating the internationalisation support. All language maintainers for providing translations into many languages.
- The band Porcupine Tree for providing hours of great music to listen to while coding.
- many, many others for suggestions, patches and bug reports.

Part I

User Manual

Installation First go to the download page (`http://www.doxygen.org/download.html`) to get the latest distribution, if you did not have it already.

This section is divided into the following sections:

- Compiling from source on Unix
- Installing the binaries on Unix
- Known compilation problems for Unix
- Compiling from source on Windows
- Installing the binaries on Windows
- Tools used to develop doxygen

1 Compiling from source on Unix

If you downloaded the source distribution, you need at least the following to build the executable:

- The GNU tools flex, bison and GNU make, and strip
- In order to generate a Makefile for your platform, you need perl (see `http://www.perl.com/`).
- The configure script assume the availibility of standard Unix tools such as sed, date, find, uname, mv, cp, cat, echo, tr, cd, and rm.

To take full advantage of doxygen's features the following additional tools should be installed.

- Troll Tech's GUI toolkit Qt (see `http://www.trolltech.com/products/qt.html`) version 3.3 or higher. This is needed to build the GUI front-end doxywizard.
- A LaTeX distribution: for instance teTeX 1.0
  par (see `http://www.tug.org/interest.html#free`). This is needed for generating LaTeX, Postscript, and PDF output.
- the Graph visualization toolkit version 1.8.10 or higher
  par (see `http://www.graphviz.org/`). Needed for the include dependency graphs, the graphical inheritance graphs, and the collaboration graphs. If you compile graphviz yourself, make sure you do include freetype support (which requires the freetype library and header files), otherwise the graphs will not render proper text labels.
- For formulas or if you do not wish to use pdflatex, the ghostscript interpreter is needed. You can find it at `www.ghostscript.com`.
- In order to generate doxygen's own documentation, Python is needed, you can find it at `www.python.org`.

Compilation is now done by performing the following steps:


```
"DECLARE_MESSAGE_MAP()= " \
TRY=try \
"CATCH_ALL(e)= catch(...)" \
END_CATCH_ALL= \
"THROW_LAST()= throw"\
"RUNTIME_CLASS(class)=class" \
"MAKEINTRESOURCE(nId)=nId" \
"IMPLEMENT_REGISTER(v, w, x, y, z)= " \
"ASSERT(x)=assert(x)" \
"ASSERT_VALID(x)=assert(x)" \
"TRACE0(x)=printf(x)" \
"OS_ERR(A,B)={ #A, B }" \
__cplusplus \
"DECLARE_OLECREATE(class)= " \
"BEGIN_DISPATCH_MAP(class1, class2)= " \
"BEGIN_INTERFACE_MAP(class1, class2)= " \
"INTERFACE_PART(class, id, name)= " \
"END_INTERFACE_MAP()=" \
"DISP_FUNCTION(class, name, function, result, id)=" \
"END_DISPATCH_MAP()=" \
"IMPLEMENT_OLECREATE2(class, name, id1, id2, id3, id4,\
 id5, id6, id7, id8, id9, id10, id11)="
```

As you can see doxygen's preprocessor is quite powerful, but if you want even more flexibility you can always write an input filter and specify it after the INPUT_FILTER tag.

If you are unsure what the effect of doxygen's preprocessing will be you can run doxygen as follows:

```
doxygen -d Preprocessor
```

This will instruct doxygen to dump the input sources to standard output after preprocessing has been done (Hint: set `QUIET = YES` and `WARNINGS = NO` in the configuration file to disable any other output).

Automatic link generationMost documentation systems have special 'see also' sections where links to other pieces of documentation can be inserted. Although doxygen also has a command to start such a section (See section \sa), it does allow you to put these kind of links anywhere in the documentation. For LaTeX documentation a reference to the page number is written instead of a link. Furthermore, the index at the end of the document can be used to quickly find the documentation of a member, class, namespace or file. For man pages no reference information is generated.

The next sections show how to generate links to the various documented entities in a source file.

18 Links to web pages and mail addresses

Doxygen will automatically replace any URLs and mail addresses found in the documentation by links (in HTML).

19 Links to classes.

All words in the documentation that correspond to a documented class and contain at least one upper case character will automatically be replaced by a link to the page containing the documentation of the class. If you want to prevent that a word that corresponds to a documented class is replaced by a link you should put a % in front of the word.

```
{
  virtual  HRESULT   QueryInterface ( REFIID iid, void **ppv) = 0;
  virtual  ULONG   AddRef () = 0;
  virtual  ULONG   Release () = 0;
};
```

Note that the PREDEFINED tag accepts function like macro definitions (like `DECLARE_INTERFACE`), normal macro substitutions (like `PURE` and `THIS`) and plain defines (like `__cplusplus`).

Note also that preprocessor definitions that are normally defined automatically by the preprocessor (like `__cplusplus`), have to be defined by hand with doxygen's parser (this is done because these defines are often platform/compiler specific).

In some cases you may want to substitute a macro name or function by something else without exposing the result to further macro substitution. You can do this but using the := operator instead of =

As an example suppose we have the following piece of code:

```
#define QList QListT
class QListT
{
};
```

Then the only way to get doxygen interpret this as a class definition for class QList is to define:

```
PREDEFINED = QListT:=QList
```

Here is an example provided by Valter Minute and Reyes Ponce that helps doxygen to wade through the boilerplate code in Microsoft's ATL & MFC libraries:

```
PREDEFINED           = "DECLARE_INTERFACE(name)=class name" \
                       "STDMETHOD(result,name)=virtual result name" \
                       "PURE= = 0" \
                       THIS_= \
                       THIS= \
                       DECLARE_REGISTRY_RESOURCEID=// \
                       DECLARE_PROTECT_FINAL_CONSTRUCT=// \
                       "DECLARE_AGGREGATABLE(Class)= " \
                       "DECLARE_REGISTRY_RESOURCEID(Id)= " \
                       DECLARE_MESSAGE_MAP= \
                       BEGIN_MESSAGE_MAP=/* \
                       END_MESSAGE_MAP=*/// \
                       BEGIN_COM_MAP=/* \
                       END_COM_MAP=*/// \
                       BEGIN_PROP_MAP=/* \
                       END_PROP_MAP=*/// \
                       BEGIN_MSG_MAP=/* \
                       END_MSG_MAP=*/// \
                       BEGIN_PROPERTY_MAP=/* \
                       END_PROPERTY_MAP=*/// \
                       BEGIN_OBJECT_MAP=/* \
                       END_OBJECT_MAP()=*/// \
                       DECLARE_VIEW_STATUS=// \
                       "STDMETHOD(a)=HRESULT a" \
                       "ATL_NO_VTABLE= " \
                       "__declspec(a)= " \
                       BEGIN_CONNECTION_POINT_MAP=/* \
                       END_CONNECTION_POINT_MAP=*/// \
                       "DECLARE_DYNAMIC(class)= " \
                       "IMPLEMENT_DYNAMIC(class1, class2)= " \
                       "DECLARE_DYNCREATE(class)= " \
                       "IMPLEMENT_DYNCREATE(class1, class2)= " \
                       "IMPLEMENT_SERIAL(class1, class2, class3)= " \
```


1. Unpack the archive, unless you already have done that:

```
gunzip doxygen-$VERSION.src.tar.gz    # uncompress the archive
tar xf doxygen-$VERSION.src.tar       # unpack it
```

2. Run the configure script:

```
sh ./configure
```

The script tries to determine the platform you use, the make tool (which *must* be GNU make) and the perl interpreter. It will report what it finds.

To override the auto detected platform and compiler you can run configure as follows:

```
configure --platform platform-type
```

See the `PLATFORMS` file for a list of possible platform options.

If you have Qt-3.3.x installed and want to build the GUI front-end, you should run the configure script with the `--with-doxywizard` option:

```
configure --with-doxywizard
```

For an overview of other configuration options use

```
configure --help
```

3. Compile the program by running make:

```
make
```

The program should compile without problems and three binaries (`doxygen` and `doxytag`) should be available in the bin directory of the distribution.

4. Optional: Generate the user manual.

```
make docs
```

To let doxygen generate the HTML documentation.

The HTML directory of the distribution will now contain the html documentation (just point a HTML browser to the file `index.html` in the html directory). You will need the `python` interpreter for this.

5. Optional: Generate a PDF version of the manual (you will need `pdflatex`, `makeindex`, and `egrep` for this).

```
make pdf
```

The PDF manual `doxygen_manual.pdf` will be located in the latex directory of the distribution. Just view and print it via the acrobat reader.

2 Installing the binaries on Unix

After the compilation of the source code do a `make install` to install doxygen. If you downloaded the binary distribution for Unix, type:

```
./configure
make install
```

Binaries are installed into the directory `<prefix>/bin`. Use `make install_docs` to install the documentation and examples into `<docdir>/doxygen`.

`<prefix>` defaults to `/usr/local` but can be changed with the `--prefix` option of the configure script. The default `<docdir>` directory is `<prefix>/share/doc/packages` and can be changed with the `--docdir` option of the configure script.

Alternatively, you can also copy the binaries from the `bin` directory manually to some `bin` directory in your search path. This is sufficient to use doxygen.

Note:
You need the GNU install tool for this to work (it is part of the coreutils package). Other install tools may put the binaries in the wrong directory!

If you have a RPM or DEP package, then please follow the standard installation procedure that is required for these packages.

3 Known compilation problems for Unix

Qt problems

The Qt include files and libraries are not a subdirectory of the directory pointed to by QTDIR on some systems (for instance on Red Hat 6.0 includes are in /usr/include/qt and libs are in /usr/lib).

The solution: go to the root of the doxygen distribution and do:

```
mkdir qt
cd qt
ln -s your-qt-include-dir-here include
ln -s your-qt-lib-dir-here lib
export QTDIR=$PWD
```

If you have a csh-like shell you should use `setenv QTDIR $PWD` instead of the `export` command above.

Now install doxygen as described above.

Bison problems

Versions 1.31 to 1.34 of bison contain a "bug" that results in a compiler errors like this:

ce_parse.cpp:348: member 'class CPPValue yyalloc::yyvs' with constructor not allowed in union

This problem has been solved in version 1.35 (versions before 1.31 will also work).

Latex problems

The file `a4wide.sty` is not available for all distributions. If your distribution does not have it please select another paper type in the config file (see the PAPER_TYPE tag in the config file).

you have to set the EXPAND_ONLY_PREDEF tag to `YES` and specify the macro definitions after the PREDEFINED or EXPAND_AS_DEFINED tag.

A typically example where some help from the preprocessor is needed is when dealing with Microsoft's __declspec language extension. Here is an example function.

```
extern "C" void __declspec(dllexport) ErrorMsg( String aMessage,...);
```

When nothing is done, doxygen will be confused and see __declspec as some sort of function. To help doxygen one typically uses the following preprocessor settings:

```
ENABLE_PREPROCESSING   = YES
MACRO_EXPANSION        = YES
EXPAND_ONLY_PREDEF     = YES
PREDEFINED             = __declspec(x)=
```

This will make sure the __declspec(dllexport) is removed before doxygen parses the source code.

For a more complex example, suppose you have the following obfuscated code fragment of an abstract base class called `IUnknown`:

```
/*! A reference to an IID */
#ifdef __cplusplus
#define REFIID const IID &
#else
#define REFIID const IID *
#endif


/*! The IUnknown interface */
DECLARE_INTERFACE(IUnknown)
{
  STDMETHOD(HRESULT,QueryInterface) (THIS_ REFIID iid, void **ppv) PURE;
  STDMETHOD(ULONG,AddRef) (THIS) PURE;
  STDMETHOD(ULONG,Release) (THIS) PURE;
};
```

without macro expansion doxygen will get confused, but we may not want to expand the REFIID macro, because it is documented and the user that reads the documentation should use it when implementing the interface.

By setting the following in the config file:

```
ENABLE_PREPROCESSING = YES
MACRO_EXPANSION      = YES
EXPAND_ONLY_PREDEF   = YES
PREDEFINED           = "DECLARE_INTERFACE(name)=class name" \
                       "STDMETHOD(result,name)=virtual result name" \
                       "PURE= = 0" \
                       THIS_= \
                       THIS= \
       __cplusplus
```

we can make sure that the proper result is fed to doxygen's parser:

```
/*! A reference to an IID */
#define REFIID

/*! The IUnknown interface */
class  IUnknown
```

diagrams_d.h

```
#ifndef _DIAGRAM_D_H
#define _DIAGRAM_D_H
#include "diagrams_a.h"
#include "diagrams_b.h"
class C;
class D : virtual protected  A, private B { public: C m_c; };
#endif
```

diagrams_e.h

```
#ifndef _DIAGRAM_E_H
#define _DIAGRAM_E_H
#include "diagrams_d.h"
class E : public D {};
#endif
```

PreprocessingSource files that are used as input to doxygen can be parsed by doxygen's built-in C-preprocessor.

By default doxygen does only partial preprocessing. That is, it evaluates conditional compilation statements (like #if) and evaluates macro definitions, but it does not perform macro expansion.

So if you have the following code fragment

```
#define VERSION 200
#define CONST_STRING const char *

#if VERSION >= 200
  static CONST_STRING version = "2.xx";
#else
  static CONST_STRING version = "1.xx";
#endif
```

Then by default doxygen will feed the following to its parser:

```
#define VERSION
#define CONST_STRING

  static CONST_STRING version = "2.xx";
```

You can disable all preprocessing by setting ENABLE_PREPROCESSING to `NO` in the configuation file. In the case above doxygen will then read both statements, i.e:

```
  static CONST_STRING version = "2.xx";
  static CONST_STRING version = "1.xx";
```

In case you want to expand the `CONST_STRING` macro, you should set the MACRO_EXPANSION tag in the config file to `YES`. Then the result after preprocessing becomes:

```
#define VERSION
#define CONST_STRING

  static const char * version = "1.xx";
```

Note that doxygen will now expand *all* macro definitions (recursively if needed). This is often too much. Therefore, doxygen also allows you to expand only those defines that you explicitly specify. For this

HP-UX & Digital Unix problems

If you are compiling for HP-UX with aCC and you get this error:

```
/opt/aCC/lbin/ld: Unsatisfied symbols:
alloca (code)
```

then you should (according to Anke Selig) edit `ce_parse.cpp` and replace

```
extern "C" {
  void *alloca (unsigned int);
};
```

with

```
#include <alloca.h>
```

If that does not help, try removing `ce_parse.cpp` and let bison rebuild it (this worked for me).

If you are compiling for Digital Unix, the same problem can be solved (according to Barnard Schmallhof) by replacing the following in ce_parse.cpp:

```
#else /* not GNU C.  */
#if (!defined (__STDC__) && defined (sparc)) || defined (__sparc__) \
    || defined (__sparc) || defined (__sgi)
#include <alloca.h>
```

with

```
#else /* not GNU C.  */
#if (!defined (__STDC__) && defined (sparc)) || defined (__sparc__) \
    || defined (__sparc) || defined (__sgi) || defined (__osf__)
#include <alloca.h>
```

Alternatively, one could fix the problem at the bison side. Here is patch for bison.simple (provided by Andre Johansen):

```
--- bison.simple~       Tue Nov 18 11:45:53 1997
+++ bison.simple        Mon Jan 26 15:10:26 1998
@@ -27,7 +27,7 @@
 #ifdef __GNUC__
 #define alloca __builtin_alloca
 #else /* not GNU C.  */
-#if (!defined (__STDC__) && defined (sparc)) || defined (__sparc__) \
     || defined (__sparc) || defined (__sgi)
+#if (!defined (__STDC__) && defined (sparc)) || defined (__sparc__) \
     || defined (__sparc) || defined (__sgi) || defined (__alpha)
 #include <alloca.h>
 #else /* not sparc */
 #if defined (MSDOS) && !defined (__TURBOC__)
```

The generated scanner.cpp that comes with doxygen is build with this patch applied.

Sun compiler problems

It appears that doxygen doesn't work properly if it is compiled with Sun's C++ WorkShop 6 Compiler. I cannot verify this myself as I do not have access to a Solaris machine with this compiler. With GNU compiler it does work and installing Sun patch 111679-13 has also been reported as a way to fix the problem.

when configuring with `--static` I got:

```
Undefined                       first referenced
 symbol                             in file
dlclose                             /usr/lib/libc.a(nss_deffinder.o)
dlsym                               /usr/lib/libc.a(nss_deffinder.o)
dlopen                              /usr/lib/libc.a(nss_deffinder.o)
```

Manually adding `-Bdynamic` after the target rule in `Makefile.doxygen` and `Makefile.doxytag` will fix this:

```
$(TARGET): $(OBJECTS) $(OBJMOC)
        $(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(OBJMOC) $(LIBS) -Bdynamic
```

GCC compiler problems

Older versions of the GNU compiler have problems with constant strings containing characters with character codes larger than 127. Therefore the compiler will fail to compile some of the translator_xx.h files. A workaround, if you are planning to use the English translation only, is to configure doxygen with the `--english-only` option.

On some platforms (such as OpenBSD) using some versions of gcc with -O2 can lead to eating all memory during the compilation of files such as config.cpp. As a workaround use –debug as a configure option or omit the -O2 for the particular files in the Makefile.

Gcc versions before 2.95 may produce broken binaries due to bugs in these compilers.

Dot problems

Due to a change in the way image maps are generated, older versions of doxygen (<=1.2.17) will not work correctly with newer versions of graphviz (>=1.8.8). The effect of this incompatibility is that generated graphs in HTML are not properly clickable. For doxygen 1.3 it is recommended to use at least graphviz 1.8.10 or higher. For doxygen 1.4.7 or higher it is recommended to use GraphViz 2.8 or higher to avoid font issues.

Red Hat 9.0 problems

If you get the following error after running make

```
tmake error: qtools.pro:70: Syntax error
```

then first type

```
export LANG=
```

before running make.

4 Compiling from source on Windows

From version 1.5.0 onwards, build files are provided for Visual Studio 2005. Also the free (as in beer) ”Express” version of Developer Studio can be used to compile doxygen. Alternatively, you can compile doxygen the Unix way using `Cygwin` or `MinGW`.

Before you can compile doxygen you need to download and install the C++ compiler of Visual Studio. Since Microsoft apparently wants to lure everyone into using their .NET stuff, they made things somewhat difficult when you use the Express version. You need to `do some manual steps` in order to setup a proper working environment for building native win32 applications such as Doxygen.

The next step is to install unxutils (see `http://sourceforge.net/projects/unxutils`). This packages contains the tools `flex` and `bison` which are needed during the compilation process if you use

- A **dotted dark green** arrow indicates private inheritance.

The elements in the class diagram in LaTeX have the following meaning:

- A **white** box indicates a class. A **marker** in the lower right corner of the box indicates that the class has base classes that are hidden. If the box has a **dashed** border this indicates virtual inheritance.
- A **solid** arrow indicates public inheritance.
- A **dashed** arrow indicates protected inheritance.
- A **dotted** arrow indicates private inheritance.

The elements in the graphs generated by the dot tool have the following meaning:

- A **white** box indicates a class or struct or file.
- A box with a **red** border indicates a node that has *more* arrows than are shown! In other words: the graph is *truncated* with respect to this node. The reason why a graph is sometimes truncated is to prevent images from becoming too large. For the graphs generated with dot doxygen tries to limit the width of the resulting image to 1024 pixels.
- A **black** box indicates that the class' documentation is currently shown.
- A **dark blue** arrow indicates an include relation (for the include dependency graph) or public inheritance (for the other graphs).
- A **dark green** arrow indicates protected inheritance.
- A **dark red** arrow indicates private inheritance.
- A **purple dashed** arrow indicated a ”usage” relation, the edge of the arrow is labled with the variable(s) responsible for the relation. Class `A` uses class `B`, if class `A` has a member variable `m` of type C, where B is a subtype of C (e.g. C could be `B`, `B*`, `T<B>*`).

Here are a couple of header files that together show the various diagrams that doxygen can generate:

`diagrams_a.h`

```
#ifndef _DIAGRAMS_A_H
#define _DIAGRAMS_A_H
class A { public: A *m_self; };
#endif
```

`diagrams_b.h`

```
#ifndef _DIAGRAMS_B_H
#define _DIAGRAMS_B_H
class A;
class B { public: A *m_a; };
#endif
```

`diagrams_c.h`

```
#ifndef _DIAGRAMS_C_H
#define _DIAGRAMS_C_H
#include "diagrams_c.h"
class D;
class C : public A { public: D *m_d; };
#endif
```

For the first two commands one should make sure formulas contain valid commands in LaTeX's math-mode. For the third command the section should contain valid command for the specific environment.

Warning:
Currently, doxygen is not very fault tolerant in recovering from typos in formulas. It may be necessary to remove the file formula.repository that is written to the html directory to get rid of an incorrect formula.

Graphs and diagramsDoxygen has built-in support to generate inheritance diagrams for C++ classes.

Doxygen can use the "dot" tool from graphviz 1.5 to generate more advanced diagrams and graphs. Graphviz is an "open-sourced", cross-platform graph drawing toolkit and can be found at http://www.graphviz.org/

If you have the "dot" tool available in the path, you can set HAVE_DOT to YES in the configuration file to let doxygen use it.

Doxygen uses the "dot" tool to generate the following graphs:

- if GRAPHICAL_HIERARCHY is set to YES, a graphical representation of the class hierarchy will be drawn, along with the textual one. Currently this feature is supported for HTML only.

  Warning: When you have a very large class hierarchy where many classes derive from a common base class, the resulting image may become too big to handle for some browsers.
- if CLASS_GRAPH is set to YES, a graph will be generated for each documented class showing the direct and indirect inheritance relations. This disables the generation of the built-in class inheritance diagrams.
- if INCLUDE_GRAPH is set to YES, an include dependency graph is generated for each documented file that includes at least one other file. This feature is currently supported for HTML and RTF only.
- if COLLABORATION_GRAPH is set to YES, a graph is drawn for each documented class and struct that shows:
  - the inheritance relations with base classes.
  - the usage relations with other structs and classes (e.g. class A has a member variable m_a of type class B, then A has an arrow to B with m_a as label).
- if CALL_GRAPH is set to YES, a graphical call graph is drawn for each function showing the functions that the function directly or indirectly calls.
- if CALLER_GRAPH is set to YES, a graphical caller graph is drawn for each function showing the functions that the function is directly or indirectly called by.

The elements in the class diagrams in HTML and RTF have the following meaning:

- A **yellow** box indicates a class. A box can have a little marker in the lower right corner to indicate that the class contains base classes that are hidden. For the class diagrams the maximum tree width is currently 8 elements. If a tree is wider some nodes will be hidden. If the box is filled with a dashed pattern the inheritance relation is virtual.
- A **white** box indicates that the documentation of the class is currently shown.
- A **grey** box indicates an undocumented class.
- A **solid dark blue** arrow indicates public inheritance.
- A **dashed dark green** arrow indicates protected inheritance.

a CVS snapshot of doxygen (the official source releases come with pre-generated sources). Download the zip extract it to e.g. c:\tools\unxutils.

Now you need to add/adjust the following environment variables (via Control Panel/System/Advanced/Environment Variables):

- add c:\tools\unxutils\usr\local\wbin; to the start of PATH
- set BISON_SIMPLE to c:\tools\unxutils\usr\share\bison.simple

Download doxygen's source tarball and put it somewhere (e.g use c:\tools)

Now start a new command shell and type

```
cd c:\tools
gunzip doxygen-x.y.z.src.tar.gz
tar xvf doxygen-x.y.z.src.tar
```

to unpack the sources.

Now your environment is setup to build doxygen and doxytag.

Inside the doxygen-x.y.z directory you will find a winbuild directory containing a Doxygen.sln file. Open this file in Visual Studio. You can now build the Release or Debug flavor of Doxygen and Doxytag by right-clicking the project in the solutions explorer, and selecting Build.

Note that compiling Doxywizard currently requires Qt version 3 (see http://www.trolltech.com/products/qt/qt3). If you do not have a commercial license, you can build Doxywizard with the open source version (see http://qtwin.sourceforge.net/qt3-win32/compile-msvc-2005.php), but I have not tried this myself.

Also read the next section for additional tools you may need to install to run doxygen with certain features enabled.

5 Installing the binaries on Windows

Doxygen comes as a self-installing archive, so installation is extremely simple. Just follow the dialogs.

After installation it is recommended to also download and install GraphViz (version 2.8 or better is highly recommended). Doxygen can use the dot tool of the GraphViz package to render nicer diagrams, see the HAVE_DOT option in the configuration file.

If you want to produce compressed HTML files (see GENERATE_HTMLHELP) in the config file, then you need the Microsoft HTML help workshop. You can download it from Microsoft.

In order to generate PDF output or use scientific formulas you will also need to install LaTeX and Ghostscript.

For LaTeX a number of distributions exists. Popular onces that should work with doxygen are MikTex and XemTex.

Ghostscript can be downloaded from Sourceforge.

After installing LaTeX and Ghostscript you'll need to make sure the tools latex.exe, pdflatex.exe, and gswin32c.exe are present in the search path of a command box. Follow these instructions if you are unsure and run the commands from a command box to verify it works.

6 Tools used to develop doxygen

Doxygen was developed and tested under Linux & MacOSX using the following open-source tools:

- GCC version 3.3.6 (Linux) and 4.0.1 (MacOSX)
- GNU flex version 2.5.33 (Linux) and 2.5.4 (MacOSX)
- GNU bison version 1.75
- GNU make version 3.80
- Perl version 5.8.1
- VIM version 6.2
- Firefox 1.5
- Troll Tech's tmake version 1.3 (included in the distribution)
- teTeX version 2.0.2
- CVS 1.12.12

Getting StartedThe executable `doxygen` is the main program that parses the sources and generates the documentation. See section Doxygen usage for more detailed usage information.

The executable `doxytag` is only needed if you want to generate references to external documentation (i.e. documentation that was generated by doxygen) for which you do not have the sources. See section Doxytag usage for more detailed usage information.

Optionally, the executable `doxywizard` can be used, which is a graphical front-end for editing the configuration file that is used by doxygen and for running doxygen in a graphical environment. For Mac OS X doxywizard will be started by clicking on the Doxygen application icon.

The following figure shows the relation between the tools and the flow of information between them (it looks complex but that's only because it tries to be complete):

Including FormulasDoxygen allows you to put LaTeX formulas in the output (this works only for the HTML and LaTeX output, not for the RTF nor for the man page output). To be able to include formulas (as images) in the HTML documentation, you will also need to have the following tools installed

- `latex`: the LaTeX compiler, needed to parse the formulas. To test I have used the teTeX 1.0 distribution.
- `dvips`: a tool to convert DVI files to PostScript files I have used version 5.92b from Radical Eye software for testing.
- `gs`: the GhostScript interpreter for converting PostScript files to bitmaps. I have used Aladdin GhostScript 8.0 for testing.

There are three ways to include formulas in the documentation.

1. Using in-text formulas that appear in the running text. These formulas should be put between a pair of \f$ commands, so

```
The distance between \f$(x_1,y_1)\f$ and \f$(x_2,y_2)\f$ is
\f$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\f$.
```

   results in:

   The distance between (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

2. Unnumbered displayed formulas that are centered on a separate line. These formulas should be put between \f[and \f] commands. An example:

```
\f[
  |I_2|=\left| \int_{0}^T \psi(t)
           \left\{
              u(a,t)-
              \int_{\gamma(t)}^a
              \frac{d\theta}{k(\theta,t)}
              \int_{a}^\theta c(\xi)u_t(\xi,t)\,d\xi
           \right\} dt
        \right|
\f]
```

   results in:

$$|I_2| = \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^a \frac{d\theta}{k(\theta,t)} \int_a^\theta c(\xi) u_t(\xi,t)\, d\xi \right\} dt \right|$$

3. Formulas or other latex elements that are not in a math environment can be specified using \f{environment}, where `environment` is the name of the LaTeX environment, the corresponding end command is \f}. Here is an example for an equation array

```
\f{eqnarray*}{
     g &=& \frac{Gm_2}{r^2} \\
       &=& \frac{(6.673 \times 10^{-11}\,\mbox{m}^3\,\mbox{kg}^{-1}\,
           \mbox{s}^{-2})(5.9736 \times 10^{24}\,\mbox{kg})}{(6371.01\,\mbox{km})^2} \\
       &=& 9.82066032\,\mbox{m/s}^2
\f}
```

   which results in:

$$\begin{aligned} g &= \frac{Gm_2}{r^2} \\ &= \frac{(6.673 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2})(5.9736 \times 10^{24}\,\mathrm{kg})}{(6371.01\,\mathrm{km})^2} \\ &= 9.82066032\,\mathrm{m/s}^2 \end{aligned}$$

level group (the group is displayed as a subsection of the "Static Public Members" section for instance). If two or more members have different types, then the group is put at the same level as the automatically generated groups. If you want to force all member-groups of a class to be at the top level, you should put a \nosubgrouping command inside the documentation of the class.

Example:

```
/** A class. Details */
class Test
{
  public:
    //@{
    /** Same documentation for both members. Details */
    void func1InGroup1();
    void func2InGroup1();
    //@}

    /** Function without group. Details. */
    void ungroupedFunction();
    void func1InGroup2();
  protected:
    void func2InGroup2();
};

void Test::func1InGroup1() {}
void Test::func2InGroup1() {}

/** @name Group2
 *  Description of group 2.
 */
//@{
/** Function 2 in group 2. Details. */
void Test::func2InGroup2() {}
/** Function 1 in group 2. Details. */
void Test::func1InGroup2() {}
//@}

/*! \file
 *  docs for this file
 */

//@{
//! one description for all members of this group
//! (because DISTRIBUTE_GROUP_DOC is YES in the config file)
#define A 1
#define B 2
void glob_func();
//@}
```

Here Group1 is displayed as a subsection of the "Public Members". And Group2 is a separate section because it contains members with different protection levels (i.e. public and protected).

17 Subpaging

Information can be grouped into pages using the \page and \mainpage commands. Normally, this results in a flat list of pages, where the "main" page is the first in the list.

Instead of adding structure using the approach decribed in section modules it is often more natural and convienent to add additional structure to the pages using the \subpage command.

For a page A the \subpage command adds a link to another page B and at the same time makes page B a subpage of A. This has the effect of making two groups GA and GB, where GB is part of GA, page A is put in group GA, and page B is put in group GB.

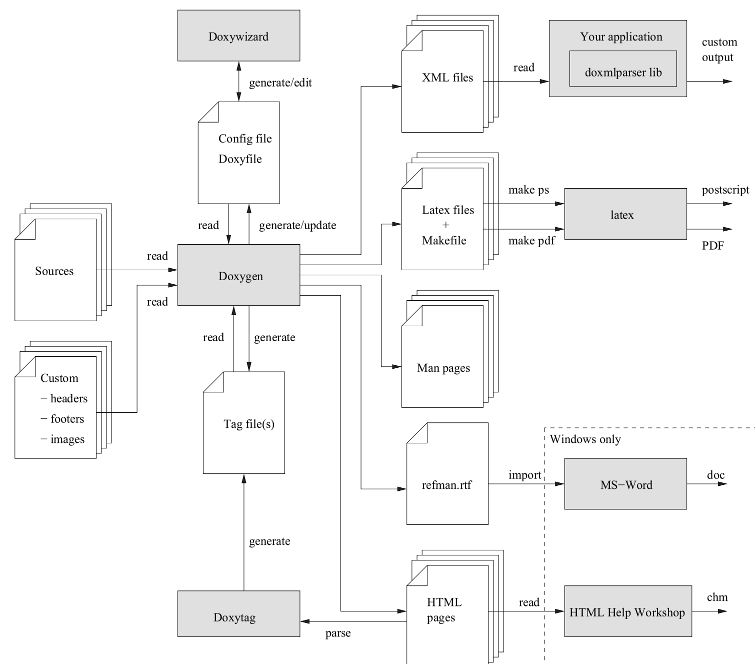

Figure 1: Doxygen information flow

7 Step 1: Creating a configuration file

Doxygen uses a configuration file to determine all of its settings. Each project should get its own configuration file. A project can consist of a single source file, but can also be an entire source tree that is recursively scanned.

To simplify the creation of a configuration file, doxygen can create a template configuration file for you. To do this call `doxygen` from the command line with the `-g` option:

```
doxygen -g <config-file>
```

where <config-file> is the name of the configuration file. If you omit the file name, a file named `Doxyfile` will be created. If a file with the name <config-file> already exists, doxygen will rename it to <config-file>.bak before generating the configuration template. If you use - (i.e. the minus sign) as the file name then doxygen will try to read the configuration file from standard input (`stdin`), which can be useful for scripting.

The configuration file has a format that is similar to that of a (simple) Makefile. It consists of a number of assignments (tags) of the form:

TAGNAME = VALUE or

TAGNAME = VALUE1 VALUE2 ...

You can probably leave the values of most tags in a generated template configuration file to their default value. See section Configuration for more details about the configuration file.

If you do not wish to edit the config file with a text editor, you should have a look at doxywizard, which is a GUI front-end that can create, read and write doxygen configuration files, and allows setting configuration options by entering them via dialogs.

For a small project consisting of a few C and/or C++ source and header files, you can leave INPUT tag empty and doxygen will search for sources in the current directory.

If you have a larger project consisting of a source directory or tree you should assign the root directory or directories to the INPUT tag, and add one or more file patterns to the FILE_PATTERNS tag (for instance `*.cpp *.h`). Only files that match one of the patterns will be parsed (if the patterns are omitted a list of source extensions is used). For recursive parsing of a source tree you must set the RECURSIVE tag to `YES`. To further fine-tune the list of files that is parsed the EXCLUDE and EXCLUDE_PATTERNS tags can be used. To omit all `test` directories from a source tree for instance, one could use:

```
EXCLUDE_PATTERNS = */test/*
```

Doxygen looks at the file's extension to determine how to parse a file. If a file has an `.idl` or `.odl` extension it is treated as an IDL file. If it has a `.java` extension it is treated as a file written in Java. Files ending with `.cs` are treated as C# files and the `.py` extension selects the Python parser. Finally, files with the extensions `.php`, `.php4`, `.inc` or `.phtml` are treated as PHP sources. Any other extension is parsed as if it is a C/C++ file, where files that end with `.m` are treated as Objective-C source files.

If you start using doxygen for an existing project (thus without any documentation that doxygen is aware of), you can still get an idea of what the structure is and how the documented result would look like. To do so, you must set the EXTRACT_ALL tag in the configuration file to `YES`. Then, doxygen will pretend everything in your sources is documented. Please note that as a consequence warnings about undocumented members will not be generated as long as EXTRACT_ALL is set to `YES`.

To analyse an existing piece of software it is useful to cross-reference a (documented) entity with its definition in the source files. Doxygen will generate such cross-references if you set the SOURCE_BROWSER tag to `YES`. It can also include the sources directly into the documentation by setting INLINE_SOURCES to `YES` (this can be handy for code reviews for instance).

8 Step 2: Running doxygen

To generate the documentation you can now enter:

```
doxygen <config-file>
```

Depending on your settings doxygen will create `html`, `rtf`, `latex`, `xml` and/or `man` directories inside the output directory. As the names suggest these directories contain the generated documentation in HTML, RTF, LaTeX, XML and Unix-Man page format.

The default output directory is the directory in which `doxygen` is started. The root directory to which the output is written can be changed using the OUTPUT_DIRECTORY. The format specific directory within the output directory can be selected using the HTML_OUTPUT, RTF_OUTPUT, LATEX_OUTPUT, XML_OUTPUT, and MAN_OUTPUT tags of the configuration file. If the output directory does not exist, `doxygen` will try to create it for you (but it will *not* try to create a whole path recursively, like `mkdir -p` does).


```
 */

/** @defgroup group5 The Fifth Group
 *  This is the fifth group
 *  @{
 */

/** @page mypage1 This is a section in group 5
 *  Text of the first section
 */

/** @page mypage2 This is another section in group 5
 *  Text of the second section
 */

/** @} */ // end of group5

/** @addtogroup group1
 *
 *  More documentation for the first group.
 *  @{
 */

/** another function in group 1 */
void func2() {}

/** yet another function in group 1 */
void func3() {}

/** @} */ // end of group1
```

16 Member Groups

If a compound (e.g. a class or file) has many members, it is often desired to group them together. Doxygen already automatically groups things together on type and protection level, but maybe you feel that this is not enough or that that default grouping is wrong. For instance, because you feel that members of different (syntactic) types belong to the same (semantic) group.

A member group is defined by a

```
//@{
  ...
//@}
```

block or a

```
/*@{*/
  ...
/*@}*/
```

block if you prefer C style comments. Note that the members of the group should be physcially inside the member group's body.

Before the opening marker of a block a separate comment block may be placed. This block should contain the @name (or \name) command and is used to specify the header of the group. Optionally, the comment block may also contain more detailed information about the group.

Nesting of member groups is not allowed.

If all members of a member group inside a class have the same type and protection level (for instance all are static public members), then the whole member group is displayed as a subgroup of the type/protection

```
This is the \ref group_label "link" to this group.
```

The priorities of grouping definitions are (from highest to lowest): \ingroup, \defgroup, \addtogroup, \weakgroup. The last command is exactly like \addtogroup with a lower priority. It was added to allow "lazy" grouping definitions: you can use commands with a higher priority in your .h files to define the hierarchy and \weakgroup in .c files without having to duplicate the hierarchy exactly.

Example:

```
/** @defgroup group1 The First Group
 *  This is the first group
 *  @{
 */

/** @brief class C1 in group 1 */
class C1 {};

/** @brief class C2 in group 1 */
class C2 {};

/** function in group 1 */
void func() {}

/** @} */ // end of group1

/**
 *  @defgroup group2 The Second Group
 *  This is the second group
 */

/** @defgroup group3 The Third Group
 *  This is the third group
 */

/** @defgroup group4 The Fourth Group
 *  @ingroup group3
 *  Group 4 is a subgroup of group 3
 */

/**
 *  @ingroup group2
 *  @brief class C3 in group 2
 */
class C3 {};

/** @ingroup group2
 *  @brief class C4 in group 2
 */
class C4 {};

/** @ingroup group3
 *  @brief class C5 in @link group3 the third group@endlink.
 */
class C5 {};

/** @ingroup group1 group2 group3 group4
 *  namespace N1 is in four groups
 *  @sa @link group1 The first group@endlink, group2, group3, group4
 *
 *  Also see @ref mypage2
 */
namespace N1 {};

/** @file
 *  @ingroup group3
 *  @brief this file in group 3
```


8.1 HTML output

The generated HTML documentation can be viewed by pointing a HTML browser to the `index.html` file in the `html` directory. For the best results a browser that supports cascading style sheets (CSS) should be used (I'm using Mozilla, Safari, Konqueror, and sometimes IE6 to test the generated output).

Some of the features the HTML section (such as GENERATE_TREEVIEW) require a browser that supports DHTML and Javascript.

If you plan to use the search engine (see SEARCHENGINE), you should view the HTML output via a PHP-enabled web server (e.g. apache with the PHP module installed).

8.2 LaTeX output

The generated LaTeX documentation must first be compiled by a LaTeX compiler (I use a recent teTeX distribution). To simplify the process of compiling the generated documentation, `doxygen` writes a `Makefile` into the `latex` directory.

The contents and targets in the `Makefile` depend on the setting of USE_PDFLATEX. If it is disabled (set to `NO`), then typing `make` in the `latex` directory a dvi file called `refman.dvi` will be generated. This file can then be viewed using `xdvi` or converted into a PostScript file `refman.ps` by typing `make ps` (this requires `dvips`).

To put 2 pages on one physical page use `make ps_2on1` instead. The resulting PostScript file can be send to a PostScript printer. If you do not have a PostScript printer, you can try to use ghostscript to convert PostScript into something your printer understands.

Conversion to PDF is also possible if you have installed the ghostscript interpreter; just type `make pdf` (or `make pdf_2on1`).

To get the best results for PDF output you should set the PDF_HYPERLINKS and USE_PDFLATEX tags to `YES`. In this case the `Makefile` will only contain a target to build `refman.pdf` directly.

8.3 RTF output

Doxygen combines the RTF output to a single file called refman.rtf. This file is optimized for importing into the Microsoft Word. Certain information is encoded using field. To show the actual value you need to select all (Edit - select all) and then toggle fields (right click and select the option from the drop down menu).

8.4 XML output

The XML output consists of a structured "dump" of the information gathered by doxygen. Each compound (class/namespace/file/...) has its own XML file and there is also an index file called index.xml.

A file called combine.xslt XSLT script is also generated and can be used to combine all XML files into a single file.

Doxygen also generates two XML schema files index.xsd (for the index file) and compound.xsd (for the compound files). This schema file describes the possible elements, their attributes and how they are structured, i.e. it the describes the grammar of the XML files and can be used for validation or to steer XSLT scripts.

In the addon/doxmlparser directory you can find a parser library for reading the XML output produced by doxygen in an incremental way (see addon/doxmlparser/include/doxmlintf.h for the interface of the library)

8.5 Man page output

The generated man pages can be viewed using the `man` program. You do need to make sure the man directory is in the man path (see the `MANPATH` environment variable). Note that there are some limitations to the capabilities of the man page format, so some information (like class diagrams, cross references and formulas) will be lost.

9 Step 3: Documenting the sources

Although documenting the sources is presented as step 3, in a new project this should of course be step 1. Here I assume you already have some code and you want doxygen to generate a nice document describing the API and maybe the internals as well.

If the EXTRACT_ALL option is set to `NO` in the configuration file (the default), then doxygen will only generate documentation for *documented* members, files, classes and namespaces. So how do you document these? For members, classes and namespaces there are basically two options:

1. Place a *special* documentation block in front of the declaration or definition of the member, class or namespace. For file, class and namespace members it is also allowed to place the documention directly after the member. See section Special documentation blocks to learn more about special documentation blocks.
2. Place a special documentation block somewhere else (another file or another location) *and* put a *structural command* in the documentation block. A structural command links a documentation block to a certain entity that can be documented (e.g. a member, class, namespace or file). See section Documentation at other places to learn more about structural commands.

Files can only be documented using the second option, since there is no way to put a documentation block before a file. Of course, file members (functions, variable, typedefs, defines) do not need an explicit structural command; just putting a special documentation block in front or behind them will do.

The text inside a special documentation block is parsed before it is written to the HTML and/or LaTeX output files.

During parsing the following steps take place:

- The special commands inside the documentation are executed. See section Special Commands for an overview of all commands.
- If a line starts with some whitespace followed by one or more asterisks (*) and then optionally more whitespace, then all whitespace and asterisks are removed.
- All resulting blank lines are treated as a paragraph separators. This saves you from placing new-paragraph commands yourself in order to make the generated documentation readable.
- Links are created for words corresponding to documented classes (unless the word is preceded by a %; then the word will not be linked and the % sign is removed).
- Links to members are created when certain patterns are found in the text. See section Automatic link generation for more information on how the automatic link generation works.
- HTML tags that are in the documentation are interpreted and converted to LaTeX equivalents for the LaTeX output. See section HTML Commands for an overview of all supported HTML tags.

Documenting the code

To avoid putting \ingroup commands in the documentation for each member you can also group members together by the open marker `@{` before the group and the closing marker `@}` after the group. The markers can be put in the documentation of the group definition or in a separate documentation block.

Groups themselves can also be nested using these grouping markers.

You will get an error message when you use the same group label more than once. If you don't want doxygen to enforce unique labels, then you can use \addtogroup instead of \defgroup. It can be used exactly like \defgroup, but when the group has been defined already, then it silently merges the existing documentation with the new one. The title of the group is optional for this command, so you can use

```
/** \addtogroup <label> */
/*\@{*/
/*\@}*/
```

to add additional members to a group that is defined in more detail elsewhere.

Note that compound entities (like classes, files and namespaces) can be put into multiple groups, but members (like variable, functions, typedefs and enums) can only be a member of one group (this restriction is in place to avoid ambiguous linking targets in case a member is not documented in the context of its class, namespace or file, but only visible as part of a group).

Doxygen will put members into the group whose definition has the highest "priority": e.g. An explicit \ingroup overrides an implicit grouping definition via `@{ @}`. Conflicting grouping definitions with the same priority trigger a warning, unless one definition was for a member without any explicit documentation.

The following example puts VarInA into group A and silently resolves the conflict for IntegerVariable by putting it into group IntVariables, because the second instance of IntegerVariable is undocumented:

```
/**
 * \ingroup A
 */
extern int VarInA;

/**
 * \defgroup IntVariables Global integer variables
 */
/*@{*/

/** an integer variable */
extern int IntegerVariable;

/*@}*/

....

/**
 * \defgroup Variables Global variables
 */
/*@{*/

/** a variable in group A */
int VarInA;

int IntegerVariable;

/*@}*/
```

The \ref command can be used to refer to a group. The first argument of the \ref command should be group's label. To use a custom link name, you can put the name of the links in double quotes after the label, as shown by the following example

```
*   - sub item 3
* - list item 2
* .
* More text in the same paragraph.
*
* More text in a new paragraph.
*/
```

Using HTML commands

If you like you can also use HTML commands inside the documentation blocks. Using these commands has the advantage that it is more natural for list items that consist of multiple paragraphs.

Here is the above example with HTML commands:

```
/*!
 *  A list of events:
 *  <ul>
 *  <li> mouse events
 *     <ol>
 *     <li>mouse move event
 *     <li>mouse click event\n
 *         More info about the click event.
 *     <li>mouse double click event
 *     </ol>
 *  <li> keyboard events
 *     <ol>
 *     <li>key down event
 *     <li>key up event
 *     </ol>
 *  </ul>
 *  More text here.
 */
```

Note:
In this case the indentation is not important.

Using \arg or @li

For compatibility with the Troll Tech's internal documentation tool and with KDoc, doxygen has two commands that can be used to create simple unnested lists.

See \arg and \li for more info.

GroupingDoxygen has three mechanisms to group things together. One mechanism works at a global level, creating a new page for each group. These groups are called 'modules' in the documentation. The second mechanism works within a member list of some compound entity, and is refered to as a 'member groups'. For pages there is a third grouping mechanism referred to as subpaging.

15 Modules

Modules are a way to group things together on a separate page. You can document a group as a whole, as well as all individual members. Members of a group can be files, namespaces, classes, functions, variables, enums, typedefs, and defines, but also other groups.

To define a group, you should put the \defgroup command in a special comment block. The first argument of the command is a label that should uniquely identify the group. The second argument is the name or title of the group as it should appear in the documentation.

You can make an entity a member of a specific group by putting a \ingroup command inside its documentation block.

10 Special documentation blocks

A special documentation block is a C or C++ style comment block with some additional markings, so doxygen knows it is a piece of documentation that needs to end up in the generated documentation. For Python and VHDL code there are a different comment conventions, which can be found in section Special documentation blocks in Python and Special documentation blocks in VHDL respectively.

For each code item there are two (or in some cases three) types of descriptions, which together form the documentation: a *brief* description and *detailed* description, both are optional. For methods and functions there is also a third type of description, the so called "in body" description, which consists of the concatenation of all comment blocks found within the body of the method or function.

Having more than one brief or detailed description is allowed (but not recommended, as the order in which the descriptions will appear is not specified).

As the name suggest, a brief description is a short one-liner, whereas the detailed description provides longer, more detailed documentation. An "in body" description can also act as a detailed description or can describe a collection of implementation details. For the HTML output brief descriptions are also use to provide tooltips at places where an item is referenced.

There are several ways to mark a comment block as a detailed description:

1. You can use the JavaDoc style, which consist of a C-style comment block starting with two *'s, like this:

```
/**
 * ... text ...
 */
```

2. or you can use the Qt style and add an exclamation mark (!) after the opening of a C-style comment block, as shown in this example:

```
/*!
 * ... text ...
 */
```

In both cases the intermediate *'s are optional, so

```
/*!
 ... text ...
*/
```

is also valid.

3. A third alternative is to use a block of at least two C++ comment lines, where each line starts with an additional slash or an exclamation mark. Here are examples of the two cases:

```
///
/// ... text ...
///
```

or

```
//!
//!... text ...
//!
```

4. Some people like to make their comment blocks more visible in the documentation. For this purpose you can use the following:

```
/********************************************//**
 *  ... text
 ***********************************************/
```

(note the 2 slashes to end the normal comment block and start a special comment block).

or

```
/////////////////////////////////////////////////
/// ... text ...
/////////////////////////////////////////////////
```

For the brief description there are also several posibilities:

1. One could use the \brief command with one of the above comment blocks. This command ends at the end of a paragraph, so the detailed description follows after an empty line.

   Here is an example:

```
/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
 */
```

2. If JAVADOC_AUTOBRIEF is set to `YES` in the configuration file, then using JavaDoc style comment blocks will automatically start a brief description which ends at the first dot followed by a space or new line. Here is an example:

```
/** Brief description which ends at this dot. Details follow
 *  here.
 */
```

   The option has the same effect for multi-line special C++ comments:

```
/// Brief description which ends at this dot. Details follow
/// here.
```

3. A third option is to use a special C++ style comment which does not span more than one line. Here are two examples:

```
/// Brief description.
/** Detailed description. */
```

   or

```
//! Brief descripion.

//! Detailed description
//! starts here.
```

   Note the blank line in the last example, which is required to separate the brief description from the block containing the detailed description. The JAVADOC_AUTOBRIEF should also be set to `NO` for this case.

As you can see doxygen is quite flexible. The following however is not legal

Using dashes

By putting a number of column-aligned minus signs at the start of a line, a bullet list will automatically be generated. Numbered lists can also be generated by using a minus followed by a hash. Nesting of lists is allowed and is based on indentation of the items.

Here is an example:

```
/*!
 *  A list of events:
 *    - mouse events
 *         -# mouse move event
 *         -# mouse click event\n
 *            More info about the click event.
 *         -# mouse double click event
 *    - keyboard events
 *         -# key down event
 *         -# key up event
 *
 *  More text here.
 */
```

The result will be:

A list of events:

- mouse events
  1. mouse move event
  2. mouse click event
     More info about the click event.
  3. mouse double click event
- keyboard events
  1. key down event
  2. key up event

More text here.

If you use tabs for indentation within lists, please make sure that TAB_SIZE in the configuration file is set to the correct tab size.

You can end a list by starting a new paragraph or by putting a dot (.) on an empty line at the same indent level as the list you would like to end.

Here is an example that speaks for itself:

```
/**
 * Text before the list
 * - list item 1
 *   - sub item 1
 *     - sub sub item 1
 *     - sub sub item 2
 *     .
 *     The dot above ends the sub sub item list.
 *     More text for the first sub item
 *   .
 *   The dot above ends the first sub item.
 *   More text for the first list item
 *   - sub item 2
```

```
    def PyMethod(self):
        pass

    ## A class variable.
    classVar = 0;

    ## @var _memVar
    #  a member variable
```

Since python looks more like Java than like C or C++, you should set OPTMIZE_OUTPUT_JAVA to YES in the config file.

14 Special documentation blocks in VHDL

For VHDL a comment normally start with "–". Doxygen will extract comments starting with "–!". There are only two types of comment blocks in VHDL; a one line –! comment representing a brief description, and a multiline –! comment (where the –! prefix is repeated for each line) representing a detailed description.

Comments are always located in front of the item that is being documented with one exception: for ports the comment can also be after the item and is then treated as a brief description for the port.

Here is an example VHDL file with doxygen comments:

```
-------------------------------------------------------
--! @file
--! @brief 2:1 Mux using with-select
-------------------------------------------------------

--! Use standard library
library ieee;
--! Use logic elements
    use ieee.std_logic_1164.all;

--! Mux entity brief description

--! Detailed description of this
--! mux design element.
entity mux_using_with is
    port (
        din_0   : in  std_logic; --! Mux first input
        din_1   : in  std_logic; --! Mux Second input
        sel     : in  std_logic; --! Select input
        mux_out : out std_logic  --! Mux output
    );
end entity;

--! @brief Architure definition of the MUX
--! @details More details about this mux element.
architecture behavior of mux_using_with is
begin
    with (sel) select
    mux_out <= din_0 when '0',
               din_1 when others;
end architecture;
```

To get proper looking output you need to set OPTIMIZE_OUTPUT_VHDL to YES in the config file. This will also affect a number of other settings. When they were not already set correctly doxygen will produce a warning telling which settings where overruled. ListsDoxygen provides a number of ways to create lists of items.


```
//! Brief description, which is
//! really a detailed description since it spans multiple lines.
/*! Oops, another detailed description!
 */
```

because doxygen only allows one brief and one detailed description.

Furthermore, if there is one brief description before a declaration and one before a definition of a code item, only the one before the *declaration* will be used. If the same situation occurs for a detailed description, the one before the *definition* is preferred and the one before the declaration will be ignored.

Here is an example of a documented piece of C++ code using the Qt style:

```
//!  A test class.
/*!
  A more elaborate class description.
*/

class Test
{
  public:

    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
                 TVal1, /*!< Enum value TVal1. */
                 TVal2, /*!< Enum value TVal2. */
                 TVal3  /*!< Enum value TVal3. */
               }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    //! A constructor.
    /*!
      A more elaborate description of the constructor.
    */
    Test();

    //! A destructor.
    /*!
      A more elaborate description of the destructor.
    */
   ~Test();

    //! A normal member taking two arguments and returning an integer value.
    /*!
      \param a an integer argument.
      \param s a constant character pointer.
      \return The test results
      \sa Test(), ~Test(), testMeToo() and publicVar()
    */
    int testMe(int a,const char *s);

    //! A pure virtual member.
    /*!
      \sa testMe()
      \param c1 the first argument.
      \param c2 the second argument.
    */
    virtual void testMeToo(char c1,char c2) = 0;

    //! A public variable.
```

```
    /*!
      Details.
    */
    int publicVar;

    //! A function variable.
    /*!
      Details.
    */
    int (*handler)(int a,int b);
};
```

The one-line comments contain a brief description, whereas the multi-line comment blocks contain a more detailed description.

The brief descriptions are included in the member overview of a class, namespace or file and are printed using a small italic font (this description can be hidden by setting BRIEF_MEMBER_DESC to `NO` in the config file). By default the brief descriptions become the first sentence of the detailed descriptions (but this can be changed by setting the REPEAT_BRIEF tag to `NO`). Both the brief and the detailed descriptions are optional for the Qt style.

By default a JavaDoc style documentation block behaves the same way as a Qt style documentation block. This is not according the JavaDoc specification however, where the first sentence of the documentation block is automatically treated as a brief description. To enable this behaviour you should set JAVADOC_AUTOBRIEF to YES in the configuration file. If you enable this option and want to put a dot in the middle of a sentence without ending it, you should put a backslash and a space after it. Here is an example:

```
  /** Brief description (e.g.\ using only a few words). Details follow. */
```

Here is the same piece of code as shown above, this time documented using the JavaDoc style and JAVADOC_AUTOBRIEF set to YES:

```
/**
 *  A test class. A more elaborate class description.
 */

class Test
{
  public:

    /**
     * An enum.
     * More detailed enum description.
     */

    enum TEnum {
          TVal1, /**< enum value TVal1. */
          TVal2, /**< enum value TVal2. */
          TVal3  /**< enum value TVal3. */
         }
       *enumPtr, /**< enum pointer. Details. */
       enumVar;  /**< enum variable. Details. */

      /**
       * A constructor.
       * A more elaborate description of the constructor.
       */
      Test();

      /**
       * A destructor.
       * A more elaborate description of the destructor.
```


13 Special documentation blocks in Python

For Python there is a standard way of documenting the code using so called documentation strings. Such strings are stored in `__doc__` and can be retrieved at runtime. Doxygen will extract such comments and assume they have to be represented in a preformatted way.

```
"""@package docstring
Documentation for this module.

More details.
"""

def func():
    """Documentation for a function.

    More details.
    """
    pass

class PyClass:
    """Documentation for a class.

    More details.
    """

    def __init__(self):
        """The constructor."""
        self._memVar = 0;

    def PyMethod(self):
        """Documentation for a method."""
        pass
```

Note that in this case none of doxygen's special commands are supported.

There is also another way to document Python code using comments that start with "##". These type of comment blocks are more in line with the way documentation blocks work for the other languages supported by doxygen and this also allows the use of special commands.

Here is the same example again but now using doxygen style comments:

```
## @package pyexample
#  Documentation for this module.
#
#  More details.

## Documentation for a function.
#
#  More details.
def func():
    pass

## Documentation for a class.
#
#  More details.
class PyClass:

    ## The constructor.
    def __init__(self):
        self._memVar = 0;

    ## Documentation for a method.
    #  @param self The object pointer.
```

```
/*! \file structcmd.h
    \brief A Documented file.

    Details.
*/

/*! \def MAX(a,b)
    \brief A macro that returns the maximum of \a a and \a b.

    Details.
*/

/*! \var typedef unsigned int UINT32
    \brief A type definition for a .

    Details.
*/

/*! \var int errno
    \brief Contains the last error code.

    \warning Not thread safe!
*/

/*! \fn int open(const char *pathname,int flags)
    \brief Opens a file descriptor.

    \param pathname The name of the descriptor.
    \param flags Opening flags.
*/

/*! \fn int close(int fd)
    \brief Closes the file descriptor \a fd.
    \param fd The descriptor to close.
*/

/*! \fn size_t write(int fd,const char *buf, size_t count)
    \brief Writes \a count bytes from \a buf to the filedescriptor \a fd.
    \param fd The descriptor to write to.
    \param buf The data buffer to write.
    \param count The number of bytes to write.
*/

/*! \fn int read(int fd,char *buf,size_t count)
    \brief Read bytes from a file descriptor.
    \param fd The descriptor to read from.
    \param buf The buffer to read into.
    \param count The number of bytes to read.
*/

#define MAX(a,b) (((a)>(b))?(a):(b))
typedef unsigned int UINT32;
int errno;
int open(const char *,int);
int close(int);
size_t write(int,const char *, size_t);
int read(int,char *,size_t);
```

Because each comment block in the example above contains a structural command, all the comment blocks could be moved to another location or input file (the source file for instance), without affecting the generated documentation. The disadvantage of this approach is that prototypes are duplicated, so all changes have to be made twice! Because of this you should first consider if this is really needed, and avoid structural commands if possible. I often receive examples that contain \fn command in comment blocks which are place in front of a function. This is clearly a case where the \fn command is redundant and will only lead to problems.


```
    */
   ~Test();

    /**
     * a normal member taking two arguments and returning an integer value.
     * @param a an integer argument.
     * @param s a constant character pointer.
     * @see Test()
     * @see ~Test()
     * @see testMeToo()
     * @see publicVar()
     * @return The test results
     */
     int testMe(int a,const char *s);

    /**
     * A pure virtual member.
     * @see testMe()
     * @param c1 the first argument.
     * @param c2 the second argument.
     */
     virtual void testMeToo(char c1,char c2) = 0;

    /**
     * a public variable.
     * Details.
     */
     int publicVar;

    /**
     * a function variable.
     * Details.
     */
     int (*handler)(int a,int b);
};
```

Similarly, if one wishes the first sentence of a Qt style documentation block to automatically be treated as a brief description, one may set QT_AUTOBRIEF to YES in the configuration file.

Unlike most other documentation systems, doxygen also allows you to put the documentation of members (including global functions) in front of the *definition*. This way the documentation can be placed in the source file instead of the header file. This keeps the header file compact, and allows the implementer of the members more direct access to the documentation. As a compromise the brief description could be placed before the declaration and the detailed description before the member definition.

11 Putting documentation after members

If you want to document the members of a file, struct, union, class, or enum, and you want to put the documentation for these members inside the compound, it is sometimes desired to place the documentation block after the member instead of before. For this purpose you have to put an additional < marker in the comment block. Note that this also works for the parameters of a function.

Here are some examples:

```
int var; /*!< Detailed description after the member */
```

This block can be used to put a Qt style detailed documentation block *after* a member. Other ways to do the same are:

```
int var; /**< Detailed description after the member */
```

or

```
int var; //!< Detailed description after the member
         //!<
```

or

```
int var; ///< Detailed description after the member
         ///<
```

Most often one only wants to put a brief description after a member. This is done as follows:

```
int var; //!< Brief description after the member
```

or

```
int var; ///< Brief description after the member
```

Note that these blocks have the same structure and meaning as the special comment blocks in the previous section only the < indicates that the member is located in front of the block instead of after the block.

Here is an example of the use of these comment blocks:

```
/*! A test class */

class Test
{
  public:
    /** An enum type.
     *  The documentation block cannot be put after the enum!
     */
    enum EnumType
    {
      int EVal1,     /**< enum value 1 */
      int EVal2      /**< enum value 2 */
    };
    void member();   //!< a member function.

  protected:
    int value;       /*!< an integer value */
};
```

Warning:
These blocks can only be used to document *members* and *parameters*. They cannot be used to document files, classes, unions, structs, groups, namespaces and enums themselves. Furthermore, the structural commands mentioned in the next section (like `\class`) are ignored inside these comment blocks.

12 Documentation at other places

So far we have assumed that the documentation blocks are always located *in front* of the declaration or definition of a file, class or namespace or in front or after one of its members. Although this is often comfortable, there may sometimes be reasons to put the documentation somewhere else. For documenting a file this is even required since there is no such thing as "in front of a file".

Doxygen allows you to put your documentation blocks practically anywhere (the exception is inside the body of a function or inside a normal C style comment block).

The price you pay for not putting the documentation block directly before (or after) an item is the need to put a structural command inside the documentation block, which leads to some duplication of information. So in practice you should *avoid* the use of structural commands *unless* other requirements force you to do so.

Structural commands (like all other commands) start with a backslash (\), or an at-sign (@) if you prefer JavaDoc style, followed by a command name and one or more parameters. For instance, if you want to document the class `Test` in the example above, you could have also put the following documentation block somewhere in the input that is read by doxygen:

```
/*! \class Test
    \brief A test class.

    A more detailed class description.
*/
```

Here the special command `\class` is used to indicate that the comment block contains documentation for the class `Test`. Other structural commands are:

- `\struct` to document a C-struct.
- `\union` to document a union.
- `\enum` to document an enumeration type.
- `\fn` to document a function.
- `\var` to document a variable or typedef or enum value.
- `\def` to document a #define.
- `\typedef` to document a type definition.
- `\file` to document a file.
- `\namespace` to document a namespace.
- `\package` to document a Java package.
- `\interface` to document an IDL interface.

See section Special Commands for detailed information about these and many other commands.

To document a member of a C++ class, you must also document the class itself. The same holds for namespaces. To document a global C function, typedef, enum or preprocessor definition you must first document the file that contains it (usually this will be a header file, because that file contains the information that is exported to other source files).

Let's repeat that, because it is often overlooked: to document global objects (functions, typedefs, enum, macros, etc), you *must* document the file in which they are defined. In other words, there *must* at least be a

```
/*! \file */
```

or a

```
/** @file */
```

line in this file.

Here is an example of a C header named `structcmd.h` that is documented using structural commands: