Manual for version 1.3.5

Written by Dimitri van Heesch

Contents

Introduction

Doxygen is a documentation system for C++, C, Java, IDL (Corba and Microsoft flavors) and to some extent PHP and C#.

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, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows 9x/NT and Mac OS X 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 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 Output Formats shows how to generate the various output formats supported by doxygen.
- Section Automatic link generation shows how to put links to files, classes, and members in the documentation.
- 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.

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

Projects using doxygen

I have compiled a 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.

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 :-)
- 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.
- Petr Prikryl for coordinating the internationalisation support. All language maintainers for providing translations into many languages.
- Erik Jan Lingen of Habanera, Mark Roddy, Paul Schwartz, Charles Duffy, Vadym Voznyuk, Philip Walton, Dwight Browne, Andreas Fredriksson, Karel Lindveld, Ivan Lee, Albert Vernon, Adam McKee, Vijapurapu Anatharac, Ben Hunsberger and Walter Wartenweiler, Jeff Garbers, David Harris, Terry Brown and Nicolas Reimen for donating money.
- The Comms group of Symbian for donating an ultra cool Revo plus organizer!
- Steve Upstill of Weta Digital for sending me some Lord of the Rings goodies.
- 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

1 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

1.1 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 make
- In order to generate a Makefile for your platform, you need perl (see http://www.perl.com/).

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 2 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.research.att.com/sw/tools/graphviz/). 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.
- The ghostscript interpreter. To be found at www.ghostscript.com.

Compilation is now done by performing the following steps:

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-2.1.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.

Note:
You will need the stream editor `sed` for this, but this should be available on any Unix platform.

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).

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.

1.2 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` 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 fileutils 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.

1.3 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).

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__) \
+#if (!defined (__STDC__) && defined (sparc)) || defined (__sparc__) \
     || defined (__sparc) || defined (__sgi)
 #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

I tried compiling doxygen only with Sun's C++ WorkShop Compiler version 5.0 (I used `./configure --platform solaris-cc`)

Qt-2.x.y is required for this compiler (Qt-1.44 has problems with the bool type).

Compiling the `doxygen` binary went ok, but while linking `doxytag` I got a lot of link errors, like these:

```
QList<PageInfo>::__vtbl   /home/dimitri/doxygen/
objects/SunWS_cache/CC_obj_6/6c3eO4IogMT2vrlGCQUQ.o
[Hint: try checking whether the first non-inlined, non-pure
virtual function of class QList<PageInfo> is defined]
```

These are generated because the compiler is confused about the object sharing between `doxygen` and `doxytag`. To compile `doxytag` anyway do:

```
rm -rf objects
mkdir objects
cd src
gmake -f Makefile.doxytag
```

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.

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.

1.4 Compiling from source on Windows

Currently, I have only compiled doxygen for Windows using Microsoft's Visual C++ (version 6.0). For other compilers you may need to edit the perl script in `wintools/make.pl` a bit. Let me know what you had to change if you got Doxygen working with another compiler.

If you have Visual C++ 6.0, and the source distribution, you can easily build doxygen using the project files in the `wintools` directory. If you want to build the CVS sources, or want to build from the command line, or with another compiler, you have to follow the steps below.

Thomas Baust reported that if you have Visual Studio.NET (2003) then you should be aware that there is a problem with the _popen() and _pclose() implementation, which currently leaks handles, so if you build doxygen with it and use the INPUT_FILTER, you will run to risk of crashing Windows! The problem is reported to and confirmed by Microsoft so maybe it will fixed in the next service pack.

Since Windows comes without all the nice tools that Unix users are used to, you'll need to install a number of these tools before you can compile doxygen for Windows from the command-line.

Here is what is required:

- An unzip/untar tool like WinZip to unpack the tar source distribution. This can be found at `http://www.winzip.com/`

  The good, tested, and free alternative is the `tar` utility supplied with `cygwin tools`. Anyway, the cygwin's flex, bison, and sed are also recommended below.

- Microsoft Visual C++ (I only tested with version 6.0). Use the `vcvars32.bat` batch file to set the environment variables (if you did not select to do this automatically during installation).

  Borland C++ or MINGW (see `http://www.mingw.org/`) are also supported.

- Perl 5.0 or higher for Windows. This can be downloaded from: `http://www.ActiveState.com/Products/ActivePerl/`

- The GNU tools flex, bison, and sed. To get these working on Windows you should install the `cygwin tools` (see `http://sources.redhat.com/cygwin/`)

  Alternatively, you can also choose to download only a `small subset` (see `http://www.doxygen.org/dl/cygwin_tools.zip`)of the cygwin tools that I put together just to compile doxygen.

  As a third alternative one could use the GNUWin32 tools that can be found at `http://gnuwin32.sourceforge.net/`

  Make sure the `BISONLIB` environment variable points to the location where the files `bison.simple` and `bison.hairy` are located. For instance if these files are in `c:\tools\cygwin\share` then BISONLIB should be set to `//c/tools/cygwin/share/`

  Also make sure the tools are available from a dos box, by adding the directory they are in to the search path.

  For those of you who are very new to cygwin (if you are going to install it from scratch), you should notice that there is an archive file `bootstrap.zip` which also contains the `tar` utility (`tar.exe`), `gzip` utilities, and the `cygwin1.dll` core. This also means that you have the `tar` in hands from the start. It can be used to unpack the tar source distribution instead of using WinZip – as mentioned at the beginning of this list of steps.

- From Doxygen-1.2.2-20001015 onwards, the distribution includes the part of Qt-2.x.y that is needed for to compile doxygen and doxytag, The Windows specific part were also created. As a result doxygen can be compiled on systems without X11 or the commerical version of Qt.

  For doxywizard, a complete Qt library is still a requirement however. You can download the non-commercial version from Troll-Tech web-site. See doxygen download page for a link.

- To generate LaTeX documentation or formulas in HTML you need the tools: `latex`, `dvips` and `gswin32`. To get these working under Windows install the fpTeX distribution. You can find more info at: `http://www.fptex.org/` and download it from CTAN or one of its mirrors. In the Netherlands for example this would be: `ftp://ftp.easynet.nl/mirror/CTAN/systems/win32/fptex/`

Make sure the tools are available from a dos box, by adding the directory they are in to the search path.

For your information, the LaTeX is freely available set of so called macros and styles on the top of the famous TeX program (by famous Donald Knuth) and the accompanied utilities (all available for free). It is used for high quality typesetting. The result – in the form of so called `DVI` (DeVice Independent) file – can be printed or displayed on various devices preserving exactly the same look up to the capability of the device. The `dvips` allows you to convert the `dvi` to the high quality PostScript (i.e. PostScript that can be processed by utilities like `psnup`, `psbook`, `psselect`, and others). The derived version of TeX (the pdfTeX) can be used to produce PDF output instead of DVI, or the PDF can be produced from PostScript using the utility `ps2pdf`.

If you want to use MikTeX then you need to select at least the medium size installation. For really old versions of MikTex or minimal installations, you may need to download the fancyhdr package separately. You can find it at: `ftp://ftp.tex.ac.uk/tex-archive/macros/latex/contrib/supported/fancyhdr/`

- If you want to generate compressed HTML help (see GENERATE_HTMLHELP) in the config file, then you need the Microsoft HTML help workshop. You can download it at: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/vs` `HH1Start.asp`
- If you used WinZip to extract the tar archive it will (apparently) not create empty folders, so you have to add the folders `objects` and `bin` manually in the root of the distribution before compiling.
- `the Graph visualization toolkit version 1.8.10`

  (see `http://www.research.att.com/sw/tools/graphviz/`). Needed for the include dependency graphs, the graphical inheritance graphs, and the collaboration graphs.

Compilation is now done by performing the following steps:

1. Open a dos box. Make sure all tools (i.e. `nmake`, `latex`, `gswin32`, `dvips`, `sed`, `flex`, `bison`, `cl`, `rm`, and `perl`), are accessible from the command-line (add them to the PATH environment variable if needed).

   Notice: The use of LaTeX is optional and only needed for compilation of the documentation into PostScript or PDF. It is *not* needed for compiling the doxygen's binaries.

2. Go to the doxygen root dir and type:

   ```
   make.bat msvc
   ```

   This should build the executables `doxygen.exe` and `doxytag.exe` using Microsoft's Visual C++ compiler (The compiler should not produce any serious warnings or errors).

   You can use also the `bcc` argument to build executables using the Borland C++ compiler, or `mingw` argument to compile using GNU gcc.

3. To build the examples, go to the `examples` subdirectory and type:

   ```
   nmake
   ```

4. To generate the doxygen documentation, go to the `doc` subdirectory and type:

   ```
   nmake
   ```

   The generated HTML docs are located in the `..\html` subdirectory.

   The sources for LaTeX documentation are located in the `..\latex` subdirectory. From those sources, the DVI, PostScript, and PDF documentation can be generated.

1.5 Installing the binaries on Windows

There is no fancy installation procedure at the moment (if anyone can add it in a location independent way please let me know).

To install doxygen, just copy the binaries from the `bin` directory to a location somewhere in the path. Alternatively, you can include the `bin` directory of the distribution to the path.

1.6 Tools used to develop doxygen

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

- GCC version 2.95.3
- GNU flex version 2.5.4
- GNU bison version 1.35
- GNU make version 3.79.1
- Perl version 5.005_03
- VIM version 5.8
- Mozilla 1.0
- Troll Tech's tmake version 1.3 (included in the distribution)
- teTeX version 1.0
- CVS 1.10.7

2 Getting started

The 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.

The following figure shows the relation between the tools and the flow of information between them:

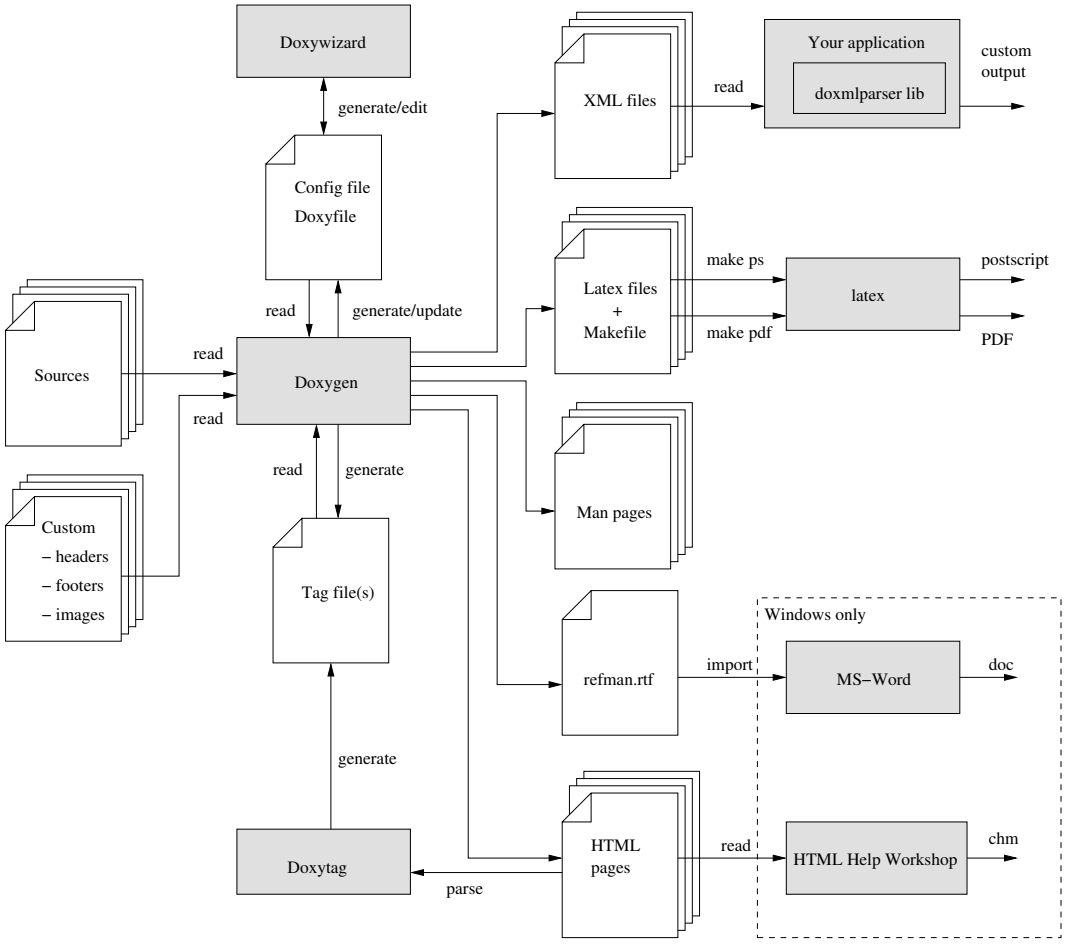

Figure 1: Doxygen information flow

2.1 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`).

The configuration file has a format that is similar to that of a (simple) Makefile. It contains 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 put the root directory or directories after 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 normally parses files if they are C or C++ sources. If a file has a `.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. Finally, files with the extensions `.php`, `.php4`, `.inc` or `.phtml` are treated as PHP sources.

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 documented result would be. 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).

2.2 Step 2: Running doxygen

To generate the documentation you can now enter:

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

Doxygen will create a `html`, `rtf`, `latex` and/or `man` directory inside the output directory. As the names suggest these directories contain the generated documentation in HTML, RTF, LaTeX and Unix-Man page format.

The default output directory is the directory in which `doxygen` is started. The directory to which the output is written can be changed using the OUTPUT_DIRECTORY, HTML_OUTPUT, RTF_OUTPUT, LATEX_OUTPUT, and MAN_OUTPUT tags of the configuration file. If the output directory does not exist, `doxygen` will try to create it for you.

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 currently using Netscape 4.61 to test the generated output).

The generated LaTeX documentation must first be compiled by a LaTeX compiler (I use teTeX distribution version 0.9 that contains TeX version 3.14159). To simplify the process of compiling the generated documentation, `doxygen` writes a `Makefile` into the `latex` directory. By typing `make` in the `latex`

directory the dvi file `refman.dvi` will be generated (provided that you have a make tool called `make` of course). 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 tag to `YES`.

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.

2.3 Step 3: Documenting the sources

Although documenting the source 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.
- 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.

3 Documenting the code

3.1 Special documentation blocks

A special documentation block is a C or C++ 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 each code item there are two types of descriptions, which together form the documentation: a *brief* description and *detailed* description, both are optional. Having more than one brief or detailed description however, is not allowed.

As the name suggest, a brief description is a short one-liner, whereas the detailed description provides longer, more detailed documentation.

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 ...
   //////////////////////////////////////////////
   ```

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

```
//! 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.
       */
     ~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);
};
```

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.

3.2 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 should put an additional $<$ marker in the comment block.

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.

3.3 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 before (or after) an item is the need to put a structural command inside the documentation block, which leads to some duplication of information.

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.
- \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:

```
/*! \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.

4 Lists

Doxygen has a number of ways to create lists of items.

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.

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 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
 *   - 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 consists 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:
The indentation here 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.

5 Grouping

Doxygen has two 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 other mechanism works within a member list of some compound entity, and is refered to as a "member group".

5.1 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. You can make an entity a member of a specific group by putting a \ingroup command inside its documentation block. The second argument is the title of the group.

To avoid putting \ingroup commands in the documentation of 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 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 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 to avoid ambiguous linking targets).

Doxygen will put members into that group where the grouping definition had the highest priority: f.i. \ingroup overrides any automatic 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 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
 */

/** @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
```

5.2 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 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).

6 Including formulas

Doxygen 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 0.9 distribution.
- `dvips`: a tool to convert DVI files to PostScript files I have used version 5.86 from Radical Eye software for testing.
- `gs`: the GhostScript interpreter for converting PostScript files to bitmaps. I have used Aladdin GhostScript 5.10 for testing.

There are two 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:

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

Formulas should be valid commands in LaTeX's math-mode.

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

7 Graphs and diagrams

Doxygen 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 from AT&T and Lucent Bell Labs and can be found at http://www.research.att.com/sw/tools/graphviz/

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).

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 **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
```

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
```

8 Preprocessing

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

As an 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
{
  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)= " \
"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).

9 Linking to external documentation

If 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.

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
```

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.

10 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 Introduction
    *
    * This is the introduction.
    *
    * \section install 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?**

   You can 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?**

   You can document your class like

   ```
   /*! \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::
   ```

   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 lexical scanner has a rule that matches more than 16K of input characters in one go. I've seen this happening on a very large generated file (>16K lines), where the built-in preprocessor converted it into an empty file (with >16K of newlines). Another case where this might happen is if you have lines in your code with more than 16K 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, so it does not know that class A relates to class B in the following example

```
#include <vector>

using namespace std;

class B {};

class A
{
  public:
    vector<B> m_bvec;
};
```

    To overcome this problem you could provide the definition of the vector class to doxygen (by including the file that defines it at the INPUT tag in the config file). Since STL header files are often messy, a (possibly) better approach is to include a dummy definition of a vector class to the input. Here is an example of a dummy STL file for the vector class:

```
namespace std {
  /*! STL vector class */
  template<class T> class vector { public: T element; };
}
```

    I’m still looking for someone who can provide me with definitions for all (relevant) STL classes.

16. **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”).

17. **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...

11 Troubleshooting

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.
- Since it 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 everthing 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.

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.

How to report a bug

Bugs are tracked in GNOME's bugzilla database. Before submitting a new bug, first check if the same bug has already been submitted by others. If you believe you have found a new bug, please file 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 1.2.4, 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 send me a small example demonstrating the problem you have, 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. When reporting a new bug you'll get a chance to attach a file to it immediately *after* opening the bug.

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

12 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, IDL, and to some extent C# and PHP sources.
- Supports documentation of files, namespaces, classes, structs, unions, templates, variables, functions, typedefs, enums and defines.
- JavaDoc (1.1), Qt-Doc, and KDOC compatible.

- Automatically generates class diagrams in HTML (as clickable image maps) and LaTeX (as Encapsulated PostScript images).
- Uses the dot tool of the Graphviz tool kit to generate include dependency graphs, collaboration diagrams, and graphical class hierarchy graphs.
- Allows you to put documentation in the header file (before the declaration of an entity), source file (before the definition of an entity) or in a separate file.
- Can generate a list of all members of a class (including any inherited members) along with their protection level.
- Outputs documentation in on-line format (HTML and UNIX man page) and off-line format (LaTeX and RTF) simultaneously (any of these can be disabled if desired). All formats are optimized for ease of reading.

  Furthermore, compressed HTML can be generated from HTML output using Microsoft's HTML Help Workshop (Windows only) and PDF can be generated from the LaTeX output.
- Includes a full C preprocessor to allow proper parsing of conditional code fragments and to allow expansion of all or part of macros definitions.
- Automatically detects public, protected and private sections, as well as the Qt specific signal and slots sections. Extraction of private class members is optional.
- Automatically generates references to documented classes, files, namespaces and members. Documentation of global functions, globals variables, typedefs, defines and enumerations is also supported.
- References to base/super classes and inherited/overridden members are generated automatically.
- Includes a fast, rank based search engine to search for strings or words in the class and member documentation.
- You can type normal HTML tags in your documentation. Doxygen will convert them to their equivalent LaTeX, RTF, and man-page counterparts automatically.
- Allows references to documentation generated for other projects (or another part of the same project) in a location independent way.
- Allows inclusion of source code examples that are automatically cross-referenced with the documentation.
- Inclusion of undocumented classes is also supported, allowing to quickly learn the structure and interfaces of a (large) piece of code without looking into the implementation details.
- Allows automatic cross-referencing of (documented) entities with their definition in the source code.
- All source code fragments are syntax highlighted for ease of reading.
- Allows inclusion of function/member/class definitions in the documentation.
- All options are read from an easy to edit and (optionally) annotated configuration file.
- Documentation and search engine can be transferred to another location or machine without regenerating the documentation.
- Can cope with large projects easily.

Although doxygen can be used in any C or C++ project, it was specifically designed to be used for projects that make use of Troll Tech's `Qt toolkit`. I have tried to make doxygen 'Qt-compatible'. That is: Doxygen can read the documentation contained in the Qt source code and create a class browser that looks very similar to the one that is generated by Troll Tech. Doxygen understands the C++ extensions used by Qt such as signals and slots.

Doxygen can also automatically generate links to existing documentation that was generated with Doxygen or with Qt's non-public class browser generator. For a Qt based project this means that whenever you refer to members or classes belonging to the Qt toolkit, a link will be generated to the Qt documentation. This is done independent of where this documentation is located!

13 Doxygen History

Version 1.2.0

Major new features:

- Support for RTF output.
- Using the dot tool of the AT&T's GraphViz package, doxygen can now generate inheritance diagrams, collaboration diagrams, include dependency graphs, included by graphs and graphical inheritance overviews.
- Function arguments can now be documented with separate comment blocks.
- Initializers and macro definitions are now included in the documentation.
- Variables and typedefs are now put in their own section.
- Old configuration files can be upgraded using the -u option without loosing any changes.
- Using the \if and \endif commands, doxygen can conditionally include documentation blocks.
- Added Doc++ like support for member grouping.
- Doxygen now has a GUI front-end called doxywizard (based on Qt-2.1)
- All info about configuration options is now concentrated in a new tool called configgen. This tool can generate the configuration parser and GUI front-end from source templates.
- Better support for the using keyword.
- New transparent mini logo that is put in the footer of all HTML pages.
- Internationalization support for the Polish, Portuguese and Croatian language.
- Todo list support.
- If the source browser is enabled, for a function, a list of function whose implementation calls that function, is generated.
- All source code fragments are now syntax highlighted in the HTML output. The colors can be changed using cascading style sheets.

Version 1.0.0

Major new features:

- Support for templates and namespaces.
- Internationalization support. Currently supported languages are: English, Czech, German, Spanish, Finnish, French, Italian, Japanese, Dutch, and Swedish.
- Automatic generation of inheritance diagrams for sub and super classes.
- Support for man page, compressed HTML help, and hyperlinked PDF output.
- Cross-referencing documentation with source code and source inlining.
- LaTeX formulas can be included in the documentation.
- Support for parsing Corba and Microsoft IDL.
- Images can be included in the documentation.
- Improved parsing and preprocessing.

Version 0.4

Major new features:

- LaTeX output generation.
- Full JavaDoc support.
- Build-in C-preprocessor for correct conditional parsing of source code that is read by Doxygen.
- Build-in HTML to LaTeX converter. This allows you to use HTML tags in your documentation, while doxygen still generates proper LaTeX output.
- Many new commands (there are now more than 60!) to document more entities, to make the documentation look nicer, and to include examples or pieces of examples.
- Enum types, enum values, typedefs, #defines, and files can now be documented.
- Completely new documentation, that is now generated by Doxygen.
- A lot of small examples are now included.

Version 0.3

Major new features:

- A PHP based search engine that allows you to search through the generated documentation.
- A configuration file instead of command-line options. A default configuration file can be generated by doxygen.
- Added an option to generate output for undocumented classes.
- Added an option to generate output for private members.

- Every page now contains a condensed index page, allowing much faster navigation through the documentation.
- Global and member variables can now be documented.
- A project name can now given, which will be included in the documentation.

Version 0.2

Major new features:

- Blocks of code are now parsed. Function calls and variables are replaced by links to their documentation if possible.
- Special example documentation block added. This can be used to provide cross references between the documentation and some example code.
- Documentation blocks can now be placed inside the body of a class.
- Documentation blocks with line range may now be created using special `//!` C++ line comments.
- Unrelated members can now be documented. A page containing a list of these members is generated.
- Added an `\include` command to insert blocks of source code into the documentation.
- Warnings are generated for members that are undocumented.
- You can now specify your own HTML headers and footers for the generated pages.
- Option added to generated indices containing all external classes instead of only the used ones.

Version 0.1

Initial version.

14 Doxygen usage

Doxygen is a command line based utility. Calling `doxygen` with the `--help` option at the command line will give you a brief description of the usage of the program.

All options consist of a leading character `-`, followed by one character and one or more arguments depending on the option.

To generate a manual for your project you typically need to follow these steps:

1. You document your source code with special documentation blocks (see section Special documentation blocks).
2. You generate a configuration file (see section Configuration) by calling doxygen with the `-g` option:

   ```
   doxygen -g <config_file>
   ```

3. You edit the configuration file so it matches your project. In the configuration file you can specify the input files and a lot of optional information.
4. You let doxygen generate the documentation, based on the settings in the configuration file:

```
doxygen <config_file>
```

If you have a configuration file generated with an older version of doxygen, you can upgrade it to the current version by running doxygen with the -u option.

```
doxygen -u <config_file>
```

All configuration settings in the orginal configuration file will be copied to the new configuration file. Any new options will have their default value. Note that comments that you may have added in the original configuration file will be lost.

If you want to fine-tune the way the output looks, doxygen allows you generate default style sheet, header, and footer files that you can edit afterwards:

- For HTML output, you can generate the default header file (see HTML_HEADER), the default footer (see HTML_FOOTER), and the default style sheet (see HTML_STYLESHEET), using the following command:

  ```
  doxygen -w html header.html footer.html stylesheet.css
  ```

- For LaTeX output, you can generate the first part of `refman.tex` (see LATEX_HEADER) and the style sheet included by that header (normally `doxygen.sty`), using:

  ```
  doxygen -w latex header.tex doxygen.sty
  ```

- For RTF output, you can generate the default style sheet file (see RTF_STYLESHEET_FILE) using:

  ```
  doxygen -w rtf rtfstyle.cfg
  ```

Note:

- If you do not want documentation for each item inside the configuration file then you can use the optional `-s` option. This can use be used in combination with the `-u` option, to add or strip the documentation from an existing configuration file. Please use the `-s` option if you send me a configuration file as part of a bug report!
- To make doxygen read/write to standard input/output instead of from/to a file, use `-` for the file name.

15 Doxytag usage

Doxytag is a small command line based utility. It can generate *tag files*. These tag files can be used with `doxygen` to generate references to external documentation (i.e. documentation not contained in the input files that are used by doxygen).

A tag file contains information about files, classes and members documented in external documentation. Doxytag extracts this information directly from the HTML files. This has the advantage that you do not need to have the sources from which the documentation was extracted.

If you *do* have the sources it is better to let `doxygen` generate the tag file by putting the name of the tag file after GENERATE_TAGFILE in the configuration file.

The input of doxytag consists of a set of HTML files.

Important:

If you use tag files, the links that are generated by doxygen will contain *dummy* links. You have to run the `installdox` script to change these dummy links into real links. See Installdox usage for more information. The use of dummy links may seem redundant, but it is really useful, if you want to move the external documentation to another location. Then the documentation does not need to be regenerated by `doxygen`, only `installdox` has to be run.

Note:

Because the HTML files are expected to have a certain structure, only HTML files generated with doxygen or with Qt's class browser generator can be used. Doxytag only *reads* the HTML files, they are not altered in any way.

Doxytag expects a list of all HTML files that form the documentation or a directory that contains all HTML files. If neither is present doxytag will read all files with a `.html` extension from the current directory. If doxytag is used with the `-t` flag it generates a tag file.

Example 1:

Suppose the file `example.cpp` from the `examples` directory that is listed below is included in some package for which you do not have the sources. Fortunately, the distributor of the packages included the HTML documentation that was generated by doxygen in the package.

```
/** A Test class.
 *  More details about this class.
 */

class Test
{
  public:
    /** An example member function.
     *  More details about this function.
     */
    void example();
};

void Test::example() {}

/** \example example_test.cpp
 * This is an example of how to use the Test class.
 * More details about this example.
 */
```

Now you can create a tag file from the HTML files in the package by typing:

```
doxytag -t example.tag example/html
```

from the examples directory. Finally you can use this tag file with your own piece of code, such as done in the following example:

```
/*! A class that is inherited from the external class Test.
*/

class Tag : public Test
{
  public:
    /*! an overloaded member. */
    void example();
};
```

Doxygen will now include links to the external package in your own documentation. Because the tag file does not specify where the documentation is located, you will have to specify that by running the installdox script that doxygen generates (See section Installdox usage for more information).

Note that this is actually a feature because if you (or someone else) moves the external documentation to a different directory or URL you can simply run the script again and all links in the HTML files will be updated.

Example 2:
To generate a tag file of the Qt documentation you can do the following:

```
doxytag -t qt.tag $QTDIR/doc/html
```

16 Doxywizard usage

Doxywizard is a GUI front-end for creating and editing configuration files that are used by doxygen.

Doxywizard consists of a single executable that, when started, pops up a window.

The main window has a number of tab field, one for each section in the configuration file. Each tab-field contains a number of lines, one for each configuration option in that section.

The kind of input widget depends on the type of the configuration option.

- For each boolean option (those options that are answered with YES or NO in the configuration file) there is a check-box.
- For items taking one of a fixed set of values (like OUTPUT_LANGUAGE) a combo box is used.
- For items taking an integer value from a range, a spinbox is used.
- For free form string-type options there is a one line edit field
- For options taking a lists of strings, a one line edit field is available, with a '+' button to add this string to the list and a '-' button to remove the selected string from the list. There is also a button with a circular "refresh" arrow that, when pressed, replaces the selected item in the list with the string entered in the edit field.
- For file and folder entries, there are special buttons that start a file dialog.

17 Installdox usage

Installdox is a perl script that is generated by doxygen whenever tag files are used (See `TAGFILES` in section External reference options) or the search engine is enabled (See `SEARCHENGINE` in section Search engine options). The script is located in the same directory where the HTML files are located.

Its purpose is to set the location of the external documentation for each tag file and to set the correct links to the search engine at install time.

Calling `installdox` with option **-h** at the command line will give you a brief description of the usage of the program.

The following options are available:

-l <`tagfile`>@<`location`> Each tag file contains information about the files, classes and members documented in a set of HTML files. A user can install these HTML files anywhere on his/her hard disk or web site. Therefore installdox *requires* the location of the documentation for each tag file <`tagfile`> that is used by doxygen. The location <`location`> can be an absolute path or a URL.

Note:
Each <tagfile> must be unique and should only be the name of the file, not including the path.

-q When this option is specified, installdox will generate no output other than fatal errors.

Optionally a list of HTML files may be given. These files are scanned and modified if needed. If this list is omitted all files in the current directory that end with .html are used.

The `installdox` script is unique for each generated class browser in the sense that it 'knows' what tag files are used. It will generate an error if the **-l** option is missing for a tag file or if an invalid tag file is given.

17.1 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:

Compressed HTML (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. In order to get hyperlinks in the PDF file, `PDF_HYPERLINKS` should be set to `YES` in the configuration file.

18 Automatic link generation

Most 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.

18.1 Links to web pages and mail addresses

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

18.2 Links to classes.

All words in the documentation that correspond to a documented class 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.

18.3 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.

18.4 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>"()"`
6. `(<className>"::")`^{n} `<functionName>`

where n>0.

Note 1:
The patterns above should not contain spaces, tabs or newlines.

Note 2:
For JavaDoc compatibility a # may be used instead of a :: in the patterns above.

Note 3:
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 5 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 6) 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.

18.5 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 6 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
{
  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.
 */
```

18.6 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 */
  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)
{
}
```

19 Configuration

19.1 Format

A configuration file is a free-form ASCII text file with a structure that is similar to that of a Makefile, default name `Doxyfile`. It is parsed by `doxygen`. The file may contain tabs and newlines for formatting purposes. The statements in the file are case-sensitive. Comments may be placed anywhere within the file (except within quotes). Comments begin with the # character and end at the end of the line.

The file essentially consists of a list of assignment statements. Each statement consists of a `TAG_NAME` written in capitals, followed by the `=` character and one or more values. If the same tag is assigned more than once, the last assignment overwrites any earlier assignment. For options that take a list as their argument, the `+=` operator can be used instead of `=` to append new values to the list. Values are sequences of non-blanks. If the value should contain one or more blanks it must be surrounded by quotes (”...”). Multiple lines can be concatenated by inserting a backslash (\) as the last character of a line. Environment variables can be expanded using the pattern $(ENV_VARIABLE_NAME).

You can also include part of a configuration file from another configuration file using a `@INCLUDE` tag as follows:

```
@INCLUDE = config_file_name
```

The include file is searched in the current working directory. You can also specify a list of directories that should be searched before looking in the current working directory. Do this by putting a `@INCLUDE_PATH` tag with these paths before the `@INCLUDE` tag, e.g:

```
@INCLUDE_PATH = my_config_dir
```

The configuration options can be divided into several categories. Below is an alphabetical index of the tags that are recognized followed by the descriptions of the tags grouped by category.

19.2 Project related options

`PROJECT_NAME` The `PROJECT_NAME` tag is a single word (or a sequence of words surrounded by double-quotes) that should identify the project for which the documentation is generated. This name is used in the title of most generated pages and in a few other places.

`PROJECT_NUMBER` The `PROJECT_NUMBER` tag can be used to enter a project or revision number. This could be handy for archiving the generated documentation or if some version control system is used.

OUTPUT_DIRECTORY The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute) path into which the generated documentation will be written. If a relative path is entered, it will be relative to the location where doxygen was started. If left blank the current directory will be used.

OUTPUT_LANGUAGE The OUTPUT_LANGUAGE tag is used to specify the language in which all documentation generated by doxygen is written. Doxygen will use this information to generate all constant output in the proper language. The default language is English, other supported languages are: Brazilian, Chinese, Croatian, Czech, Danish, Dutch, Finnish, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Romanian, Russian, Serbian, Slovak, Slovene, Spanish, Swedish, and Ukrainian.

USE_WINDOWS_ENCODING This tag can be used to specify the encoding used in the generated output. The encoding is not always determined by the language that is chosen, but also whether or not the output is meant for Windows or non-Windows users. In case there is a difference, setting the USE_WINDOWS_ENCODING tag to YES forces the Windows encoding, (this is the default for the Windows binary), whereas setting the tag to NO uses a Unix-style encoding (the default for all platforms other than Windows).

BRIEF_MEMBER_DESC If the BRIEF_MEMBER_DESC tag is set to YES (the default) doxygen will include brief member descriptions after the members that are listed in the file and class documentation (similar to JavaDoc). Set to NO to disable this.

REPEAT_BRIEF If the REPEAT_BRIEF tag is set to YES (the default) doxygen will prepend the brief description of a member or function before the detailed description

Note:
If both HIDE_UNDOC_MEMBERS and BRIEF_MEMBER_DESC are set to NO, the brief descriptions will be completely suppressed.

ABBREVIATE_BRIEF This tag implements a quasi-intelligent brief description abbreviator that is used to form the text in various listings. Each string in this list, if found as the leading text of the brief description, will be stripped from the text and the result after processing the whole list, is used as the annotated text. Otherwise, the brief description is used as-is. If left blank, the following values are used (”\$name” is automatically replaced with the name of the entity): ”The $name class” ”The $name widget” ”The $name file” ”is” ”provides” ”specifies” ”contains” ”represents” ”a” ”an” ”the”.

ALWAYS_DETAILED_SEC If the ALWAYS_DETAILED_SEC and REPEAT_BRIEF tags are both set to YES then doxygen will generate a detailed section even if there is only a brief description.

INLINE_INHERITED_MEMB If the INLINE_INHERITED_MEMB tag is set to YES, doxygen will show all inherited members of a class in the documentation of that class as if those members were ordinary class members. Constructors, destructors and assignment operators of the base classes will not be shown.

FULL_PATH_NAMES If the FULL_PATH_NAMES tag is set to YES doxygen will prepend the full path before files name in the file list and in the header files. If set to NO the shortest path that makes the file name unique will be used

STRIP_FROM_PATH If the FULL_PATH_NAMES tag is set to YES then the STRIP_FROM_PATH tag can be used to strip a user-defined part of the path. Stripping is only done if one of the specified strings matches the left-hand part of the path.

CASE_SENSE_NAMES If the CASE_SENSE_NAMES tag is set to NO (the default) then doxygen will only generate file names in lower-case letters. If set to YES upper-case letters are also allowed. This is useful if you have classes or files whose names only differ in case and if your file system supports case sensitive file names.

SHORT_NAMES If the SHORT_NAMES tag is set to YES, doxygen will generate much shorter (but less readable) file names. This can be useful is your file systems doesn't support long names like on DOS, Mac, or CD-ROM.

VERBATIM_HEADERS If the VERBATIM_HEADERS tag is set the YES (the default) then doxygen will generate a verbatim copy of the header file for each class for which an include is specified. Set to NO to disable this.

See also:
Section \class.

JAVADOC_AUTOBRIEF If the JAVADOC_AUTOBRIEF is set to YES then doxygen will interpret the first line (until the first dot) of a JavaDoc-style comment as the brief description. If set to NO (the default), the Javadoc-style will behave just like the Qt-style comments.

MULTILINE_CPP_IS_BRIEF The MULTILINE_CPP_IS_BRIEF tag can be set to YES to make Doxygen treat a multi-line C++ special comment block (i.e. a block of //! or /// comments) as a brief description. This used to be the default behaviour. The new default is to treat a multi-line C++ comment block as a detailed description. Set this tag to YES if you prefer the old behaviour instead. Note that setting this tag to YES also means that rational rose comments are not recognized any more.

DETAILS_AT_TOP If the DETAILS_AT_TOP tag is set to YES then Doxygen will output the detailed description near the top, like JavaDoc. If set to NO, the detailed description appears after the member documentation.

INHERIT_DOCS If the INHERIT_DOCS tag is set to YES (the default) then an undocumented member inherits the documentation from any documented member that it re-implements.

DISTRIBUTE_GROUP_DOC If member grouping is used in the documentation and the DISTRIBUTE_GROUP_DOC tag is set to YES, then doxygen will reuse the documentation of the first member in the group (if any) for the other members of the group. By default all members of a group must be documented explicitly.

TAB_SIZE the TAB_SIZE tag can be used to set the number of spaces in a tab. Doxygen uses this value to replace tabs by spaces in code fragments.

ALIASES This tag can be used to specify a number of aliases that acts as commands in the documentation. An alias has the form

```
name=value
```

For example adding

```
"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:". You can put \n's in the value part of an alias to insert newlines.

OPTIMIZE_OUTPUT_FOR_C Set the OPTIMIZE_OUTPUT_FOR_C tag to YES if your project consists of C sources only. Doxygen will then generate output that is more tailored for C. For instance, some of the names that are used will be different. The list of all members will be omitted, etc.

OPTIMIZE_OUTPUT_JAVA Set the OPTIMIZE_OUTPUT_JAVA tag to YES if your project consists of Java sources only. Doxygen will then generate output that is more tailored for Java. For instance, namespaces will be presented as packages, qualified scopes will look different, etc.

SUBGROUPING Set the SUBGROUPING tag to YES (the default) to allow class member groups of the same type (for instance a group of public functions) to be put as a subgroup of that type (e.g. under the Public Functions section). Set it to NO to prevent subgrouping. Alternatively, this can be done per class using the \nosubgrouping command.

19.3 Build related options

EXTRACT_ALL If the EXTRACT_ALL tag is set to YES doxygen will assume all entities in documentation are documented, even if no documentation was available. Private class members and static file members will be hidden unless the EXTRACT_PRIVATE and EXTRACT_STATIC tags are set to YES

Note:
This will also disable the warnings about undocumented members that are normally produced when WARNINGS is set to YES

EXTRACT_PRIVATE If the EXTRACT_PRIVATE tag is set to YES all private members of a class will be included in the documentation.

EXTRACT_STATIC If the EXTRACT_STATIC tag is set to YES all static members of a file will be included in the documentation.

EXTRACT_LOCAL_CLASSES If the EXTRACT_LOCAL_CLASSES tag is set to YES classes (and structs) defined locally in source files will be included in the documentation. If set to NO only classes defined in header files are included. Does not have any effect for Java sources.

HIDE_UNDOC_MEMBERS If the HIDE_UNDOC_MEMBERS tag is set to YES, doxygen will hide all undocumented members inside documented classes or files. If set to NO (the default) these members will be included in the various overviews, but no documentation section is generated. This option has no effect if EXTRACT_ALL is enabled.

HIDE_UNDOC_CLASSES If the HIDE_UNDOC_CLASSESS tag is set to YES, doxygen will hide all undocumented classes. If set to NO (the default) these classes will be included in the various overviews. This option has no effect if EXTRACT_ALL is enabled.

HIDE_FRIEND_COMPOUNDS If the HIDE_FRIEND_COMPOUNDS tag is set to YES, Doxygen will hide all friend (class|struct|union) declarations. If set to NO (the default) these declarations will be included in the documentation.

HIDE_IN_BODY_DOCS If the HIDE_IN_BODY_DOCS tag is set to YES, Doxygen will hide any documentation blocks found inside the body of a function. If set to NO (the default) these blocks will be appended to the function's detailed documentation block.

INTERNAL_DOCS The INTERNAL_DOCS tag determines if documentation that is typed after a \internal command is included. If the tag is set to NO (the default) then the documentation will be excluded. Set it to YES to include the internal documentation.

HIDE_SCOPE_NAMES If the HIDE_SCOPE_NAMES tag is set to NO (the default) then doxygen will show members with their full class and namespace scopes in the documentation. If set to YES the scope will be hidden.

SHOW_INCLUDE_FILES If the SHOW_INCLUDE_FILES tag is set to YES (the default) then doxygen will put a list of the files that are included by a file in the documentation of that file.

INLINE_INFO If the INLINE_INFO tag is set to YES (the default) then a tag [inline] is inserted in the documentation for inline members.

SORT_MEMBER_DOCS If the SORT_MEMBER_DOCS tag is set to YES (the default) then doxygen will sort the (detailed) documentation of file and class members alphabetically by member name. If set to NO the members will appear in declaration order.

GENERATE_DEPRECATEDLIST The GENERATE_DEPRECATEDLIST tag can be used to enable (YES) or disable (NO) the deprecated list. This list is created by putting \deprecated commands in the documentation.

GENERATE_TODOLIST The GENERATE_TODOLIST tag can be used to enable (YES) or disable (NO) the todo list. This list is created by putting \todo commands in the documentation.

GENERATE_TESTLIST The GENERATE_TESTLIST tag can be used to enable (YES) or disable (NO) the test list. This list is created by putting \test commands in the documentation.

GENERATE_BUGLIST The GENERATE_BUGLIST tag can be used to enable (YES) or disable (NO) the bug list. This list is created by putting \bug commands in the documentation.

ENABLED_SECTIONS The ENABLED_SECTIONS tag can be used to enable conditional documentation sections, marked by \if <section-label> ... \endif blocks.

MAX_INITIALIZER_LINES The MAX_INITIALIZER_LINES tag determines the maximum number of lines that the initial value of a variable or define can be. If the initializer consists of more lines than specified here it will be hidden. Use a value of 0 to hide initializers completely. The appearance of the value of individual variables and defines can be controlled using \showinitializer or \hideinitializer command in the documentation.

SHOW_USED_FILES Set the SHOW_USED_FILES tag to NO to disable the list of files generated at the bottom of the documentation of classes and structs. If set to YES the list will mention the files that were used to generate the documentation.

19.4 Options related to warning and progress messages

QUIET The QUIET tag can be used to turn on/off the messages that are generated to standard output by doxygen. Possible values are YES and NO, where YES implies that the messages are off. If left blank NO is used.

WARNINGS The WARNINGS tag can be used to turn on/off the warning messages that are generated to standard error by doxygen. Possible values are YES and NO, where YES implies that the warnings are on. If left blank NO is used.

Tip: Turn warnings on while writing the documentation.

WARN_IF_UNDOCUMENTED If WARN_IF_UNDOCUMENTED is set to YES, then doxygen will generate warnings for undocumented members. If EXTRACT_ALL is set to YES then this flag will automatically be disabled.

WARN_IF_DOC_ERROR If WARN_IF_DOC_ERROR is set to YES, doxygen will generate warnings for potential errors in the documentation, such as not documenting some parameters in a documented function, or documenting parameters that don't exist or using markup commands wrongly.

WARN_FORMAT The WARN_FORMAT tag determines the format of the warning messages that doxygen can produce. The string should contain the $file, $line, and $text tags, which will be replaced by the file and line number from which the warning originated and the warning text.

WARN_LOGFILE The WARN_LOGFILE tag can be used to specify a file to which warning and error messages should be written. If left blank the output is written to stderr.

19.5 Input related options

INPUT The INPUT tag is used to specify the files and/or directories that contain documented source files. You may enter file names like myfile.cpp or directories like /usr/src/myproject. Separate the files or directories with spaces.

Note: If this tag is empty the current directory is searched.

FILE_PATTERNS If the value of the INPUT tag contains directories, you can use the FILE_PATTERNS tag to specify one or more wildcard patterns (like ∗.cpp and ∗.h) to filter out the source-files in the directories. If left blank the following patterns are tested: .c ∗.cc ∗.cxx ∗.cpp ∗.c++ ∗.java ∗.ii ∗.ixx ∗.ipp ∗.i++ ∗.inl ∗.h ∗.hh ∗.hxx ∗.hpp .h++ ∗.idl ∗.odl ∗.cs

RECURSIVE The RECURSIVE tag can be used to specify whether or not subdirectories should be searched for input files as well. Possible values are YES and NO. If left blank NO is used.

EXCLUDE The EXCLUDE tag can be used to specify files and/or directories that should excluded from the INPUT source files. This way you can easily exclude a subdirectory from a directory tree whose root is specified with the INPUT tag.

EXCLUDE_SYMLINKS The EXCLUDE_SYMLINKS tag can be used select whether or not files or directories that are symbolic links (a Unix filesystem feature) are excluded from the input.

EXCLUDE_PATTERNS If the value of the INPUT tag contains directories, you can use the EXCLUDE_PATTERNS tag to specify one or more wildcard patterns to exclude certain files from those directories.

EXAMPLE_PATH The EXAMPLE_PATH tag can be used to specify one or more files or directories that contain example code fragments that are included (see the \include command in section \include).

EXAMPLE_RECURSIVE If the EXAMPLE_RECURSIVE tag is set to YES then subdirectories will be searched for input files to be used with the \include or \dontinclude commands irrespective of the value of the RECURSIVE tag. Possible values are YES and NO. If left blank NO is used.

EXAMPLE_PATTERNS If the value of the EXAMPLE_PATH tag contains directories, you can use the EXAMPLE_PATTERNS tag to specify one or more wildcard pattern (like ∗.cpp and ∗.h) to filter out the source-files in the directories. If left blank all files are included.

IMAGE_PATH The IMAGE_PATH tag can be used to specify one or more files or directories that contain images that are to be included in the documentation (see the \image command).

INPUT_FILTER The INPUT_FILTER tag can be used to specify a program that doxygen should invoke to filter for each input file. Doxygen will invoke the filter program by executing (via popen()) the command:

```
<filter> <input-file>
```

where <filter> is the value of the INPUT_FILTER tag, and <input-file> is the name of an input file. Doxygen will then use the output that the filter program writes to standard output.

FILTER_SOURCE_FILES If the FILTER_SOURCE_FILES tag is set to YES, the input filter (if set using INPUT_FILTER) will be used to filter the input files when producing source files to browse.

19.6 Source browsing related options

SOURCE_BROWSER If the SOURCE_BROWSER tag is set to YES then a list of source files will be generated. Documented entities will be cross-referenced with these sources.

INLINE_SOURCES Setting the INLINE_SOURCES tag to YES will include the body of functions, classes and enums directly into the documentation.

STRIP_CODE_COMMENTS Setting the STRIP_CODE_COMMENTS tag to YES (the default) will instruct doxygen to hide any special comment blocks from generated source code fragments. Normal C and C++ comments will always remain visible.

REFERENCED_BY_RELATION If the REFERENCED_BY_RELATION tag is set to YES (the default) then for each documented function all documented functions referencing it will be listed.

REFERENCES_RELATION If the REFERENCES_RELATION tag is set to YES (the default) then for each documented function all documented entities called/used by that function will be listed.

19.7 Alphabetical index options

ALPHABETICAL_INDEX If the ALPHABETICAL_INDEX tag is set to YES, an alphabetical index of all compounds will be generated. Enable this if the project contains a lot of classes, structs, unions or interfaces.

COLS_IN_ALPHA_INDEX If the alphabetical index is enabled (see ALPHABETICAL_INDEX) then the COLS_IN_ALPHA_INDEX tag can be used to specify the number of columns in which this list will be split (can be a number in the range [1..20])

IGNORE_PREFIX In case all classes in a project start with a common prefix, all classes will be put under the same header in the alphabetical index. The IGNORE_PREFIX tag can be used to specify a prefix (or a list of prefixes) that should be ignored while generating the index headers.

19.8 HTML related options

GENERATE_HTML If the GENERATE_HTML tag is set to YES (the default) doxygen will generate HTML output

HTML_OUTPUT The HTML_OUTPUT tag is used to specify where the HTML docs will be put. If a relative path is entered the value of OUTPUT_DIRECTORY will be put in front of it. If left blank 'html' will be used as the default path.

HTML_FILE_EXTENSION The HTML_FILE_EXTENSION tag can be used to specify the file extension for each generated HTML page (for example: .htm, .php, .asp). If it is left blank doxygen will generate files with .html extension.

HTML_HEADER The HTML_HEADER tag can be used to specify a user-defined HTML header file for each generated HTML page. To get valid HTML the header file should contain at least a <HTML> and a <BODY> tag, but it is good idea to include the style sheet that is generated by doxygen as well. Minimal example:

```
<HTML>
  <HEAD>
    <TITLE>My title</TITLE>
    <LINK HREF="doxygen.css" REL="stylesheet" TYPE="text/css">
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
```

If the tag is left blank doxygen will generate a standard header.

The following commands have a special meaning inside the header: $title, $datetime, $date, $doxygenversion, $projectname, $projectnumber. Doxygen will replace them by respectively the title of the page, the current date and time, only the current date, the version number of doxygen, the project name (see PROJECT_NAME), or the project number (see PROJECT_NUMBER).

See also section Doxygen usage for information on how to generate the default header that doxygen normally uses.

HTML_FOOTER The HTML_FOOTER tag can be used to specify a user-defined HTML footer for each generated HTML page. To get valid HTML the header file should contain at least a </BODY> and a </HTML> tag. A minimal example:

```
  </BODY>
</HTML>
```

If the tag is left blank doxygen will generate a standard footer.

The following commands have a special meaning inside the header: $title, $datetime, $date, $doxygenversion, $projectname, $projectnumber. Doxygen will replace them by respectively the title of the page, the current date and time, only the current date, the version number of doxygen, the project name (see PROJECT_NAME), or the project number (see PROJECT_NUMBER).

See also section Doxygen usage for information on how to generate the default footer that doxygen normally uses.

HTML_STYLESHEET The HTML_STYLESHEET tag can be used to specify a user-defined cascading style sheet that is used by each HTML page. It can be used to fine-tune the look of the HTML output. If the tag is left blank doxygen will generate a default style sheet.

See also section Doxygen usage for information on how to generate the style sheet that doxygen normally uses.

HTML_ALIGN_MEMBERS If the HTML_ALIGN_MEMBERS tag is set to YES, the members of classes, files or namespaces will be aligned in HTML using tables. If set to NO a bullet list will be used.

Note: Setting this tag to NO will become obsolete in the future, since I only intent to support and test the aligned representation.

GENERATE_HTMLHELP If the GENERATE_HTMLHELP tag is set to YES then doxygen generates three additional HTML index files: index.hhp, index.hhc, and index.hhk. The index.hhp is a project file that can be read by Microsoft's HTML Help Workshop on Windows.

The HTML Help Workshop contains a compiler that can convert all HTML output generated by doxygen into a single compressed HTML file (.chm). Compressed HTML files are now used as the Windows 98 help format, and will replace the old Windows help format (.hlp) on all Windows platforms in the future. Compressed HTML files also contain an index, a table of contents, and you can search for words in the documentation. The HTML workshop also contains a viewer for compressed HTML files.

CHM_FILE If the GENERATE_HTMLHELP tag is set to YES, the CHM_FILE tag can be used to specify the file name of the resulting .chm file. You can add a path in front of the file if the result should not be written to the html output directory.

HHC_LOCATION If the GENERATE_HTMLHELP tag is set to YES, the HHC_LOCATION tag can be used to specify the location (absolute path including file name) of the HTML help compiler (hhc.exe). If non empty doxygen will try to run the HTML help compiler on the generated index.hhp.

GENERATE_CHI If the GENERATE_HTMLHELP tag is set to YES, the GENERATE_CHI flag controls if a separate .chi index file is generated (YES) or that it should be included in the master .chm file (NO).

BINARY_TOC If the GENERATE_HTMLHELP tag is set to YES, the BINARY_TOC flag controls whether a binary table of contents is generated (YES) or a normal table of contents (NO) in the .chm file.

TOC_EXPAND The TOC_EXPAND flag can be set to YES to add extra items for group members to the table of contents of the HTML help documentation and to the tree view.

DISABLE_INDEX If you want full control over the layout of the generated HTML pages it might be necessary to disable the index and replace it with your own. The DISABLE_INDEX tag can be used to turn on/off the condensed index at top of each page. A value of NO (the default) enables the index and the value YES disables it.

ENUM_VALUES_PER_LINE This tag can be used to set the number of enum values (range [1..20]) that doxygen will group on one line in the generated HTML documentation.

GENERATE_TREEVIEW If the GENERATE_TREEVIEW tag is set to YES, a side panel will be generated containing a tree-like index structure (just like the one that is generated for HTML Help). For this to work a browser that supports JavaScript and frames is required (for instance Mozilla 1.0+, Netscape 6.0+ or Internet explorer 5.0+ or Konqueror).

TREEVIEW_WIDTH If the treeview is enabled (see GENERATE_TREEVIEW) then this tag can be used to set the initial width (in pixels) of the frame in which the tree is shown.

19.9 LaTeX related options

GENERATE_LATEX If the GENERATE_LATEX tag is set to YES (the default) doxygen will generate LaTeX output.

LATEX_OUTPUT The LATEX_OUTPUT tag is used to specify where the LaTeX docs will be put. If a relative path is entered the value of OUTPUT_DIRECTORY will be put in front of it. If left blank 'latex' will be used as the default path.

LATEX_CMD_NAME The LATEX_CMD_NAME tag can be used to specify the LaTeX command name to be invoked. If left blank 'latex' will be used as the default command name.

MAKEINDEX_CMD_NAME The MAKEINDEX_CMD_NAME tag can be used to specify the command name to generate index for LaTeX. If left blank 'makeindex' will be used as the default command name.

COMPACT_LATEX If the COMPACT_LATEX tag is set to YES doxygen generates more compact LaTeX documents. This may be useful for small projects and may help to save some trees in general.

PAPER_TYPE The PAPER_TYPE tag can be used to set the paper type that is used by the printer. Possible values are:

- a4 (210 x 297 mm).
- a4wide (same as a4, but including the a4wide package).
- letter (8.5 x 11 inches).
- legal (8.5 x 14 inches).
- executive (7.25 x 10.5 inches)

If left blank a4wide will be used.

EXTRA_PACKAGES The EXTRA_PACKAGES tag can be used to specify one or more LaTeX package names that should be included in the LaTeX output. To get the times font for instance you can specify

```
EXTRA_PACKAGES = times
```

If left blank no extra packages will be included.

LATEX_HEADER The LATEX_HEADER tag can be used to specify a personal LaTeX header for the generated LaTeX document. The header should contain everything until the first chapter.

If it is left blank doxygen will generate a standard header. See section Doxygen usage for information on how to let doxygen write the default header to a separate file.

Note:
Only use a user-defined header if you know what you are doing!

The following commands have a special meaning inside the header: `$title`, `$datetime`, `$date`, `$doxygenversion`, `$projectname`, `$projectnumber`. Doxygen will replace them by respectively the title of the page, the current date and time, only the current date, the version number of doxygen, the project name (see `PROJECT_NAME`), or the project number (see `PROJECT_NUMBER`).

`PDF_HYPERLINKS` If the `PDF_HYPERLINKS` tag is set to `YES`, the LaTeX that is generated is prepared for conversion to PDF (using ps2pdf). The PDF file will contain links (just like the HTML output) instead of page references. This makes the output suitable for online browsing using a PDF viewer.

`USE_PDFLATEX` If the `LATEX_PDFLATEX` tag is set to `YES`, doxygen will use pdflatex to generate the PDF file directly from the LaTeX files.

`LATEX_BATCHMODE` If the `LATEX_BATCHMODE` tag is set to `YES`, doxygen will add the \batchmode. command to the generated LaTeX files. This will instruct LaTeX to keep running if errors occur, instead of asking the user for help. This option is also used when generating formulas in HTML.

`LATEX_HIDE_INDICES` If `LATEX_HIDE_INDICES` is set to `YES` then doxygen will not include the index chapters (such as File Index, Compound Index, etc.) in the output.

19.10 RTF related options

`GENERATE_RTF` If the `GENERATE_RTF` tag is set to `YES` doxygen will generate RTF output. The RTF output is optimized for Word 97 and may not look too pretty with other readers/editors.

`RTF_OUTPUT` The `RTF_OUTPUT` tag is used to specify where the RTF docs will be put. If a relative path is entered the value of `OUTPUT_DIRECTORY` will be put in front of it. If left blank `rtf` will be used as the default path.

`COMPACT_RTF` If the `COMPACT_RTF` tag is set to `YES` doxygen generates more compact RTF documents. This may be useful for small projects and may help to save some trees in general.

`RTF_HYPERLINKS` If the `RTF_HYPERLINKS` tag is set to `YES`, the RTF that is generated will contain hyperlink fields. The RTF file will contain links (just like the HTML output) instead of page references. This makes the output suitable for online browsing using Word or some other Word compatible reader that support those fields.

note:
WordPad (write) and others do not support links.

`RTF_STYLESHEET_FILE` Load stylesheet definitions from file. Syntax is similar to doxygen's config file, i.e. a series of assignments. You only have to provide replacements, missing definitions are set to their default value.

See also section Doxygen usage for information on how to generate the default style sheet that doxygen normally uses.

`RTF_EXTENSIONS_FILE` Set optional variables used in the generation of an RTF document. Syntax is similar to doxygen's config file. A template extensions file can be generated using `doxygen -e rtf extensionFile`.

19.11 Man page related options

GENERATE_MAN If the GENERATE_MAN tag is set to YES (the default) doxygen will generate man pages for classes and files.

MAN_OUTPUT The MAN_OUTPUT tag is used to specify where the man pages will be put. If a relative path is entered the value of OUTPUT_DIRECTORY will be put in front of it. If left blank 'man' will be used as the default path. A directory man3 will be created inside the directory specified by MAN_OUTPUT.

MAN_EXTENSION The MAN_EXTENSION tag determines the extension that is added to the generated man pages (default is the subroutine's section .3)

MAN_LINKS If the MAN_LINKS tag is set to YES and doxygen generates man output, then it will generate one additional man file for each entity documented in the real man page(s). These additional files only source the real man page, but without them the man command would be unable to find the correct page. The default is NO.

19.12 XML related options

GENERATE_XML If the GENERATE_XML tag is set to YES Doxygen will generate an XML file that captures the structure of the code including all documentation.

XML_OUTPUT The XML_OUTPUT tag is used to specify where the XML pages will be put. If a relative path is entered the value of OUTPUT_DIRECTORY will be put in front of it. If left blank xml will be used as the default path.

XML_SCHEMA The XML_SCHEMA tag can be used to specify an XML schema, which can be used by a validating XML parser to check the syntax of the XML files.

XML_DTD The XML_DTD tag can be used to specify an XML DTD, which can be used by a validating XML parser to check the syntax of the XML files.

XML_PROGRAMLISTING If the XML_PROGRAMLISTING tag is set to YES Doxygen will dump the program listings (including syntax highlighting and cross-referencing information) to the XML output. Note that enabling this will significantly increase the size of the XML output.

19.13 AUTOGEN_DEF related options

GENERATE_AUTOGEN_DEF If the GENERATE_AUTOGEN_DEF tag is set to YES Doxygen will generate an AutoGen Definitions (see http://autogen.sf.net) file that captures the structure of the code including all documentation. Note that this feature is still experimental and incomplete at the moment.

19.14 PERLMOD related options

GENERATE_PERLMOD If the GENERATE_PERLMOD tag is set to YES Doxygen will generate a Perl module file that captures the structure of the code including all documentation. Note that this feature is still experimental and incomplete at the moment.

PERLMOD_LATEX If the PERLMOD_LATEX tag is set to YES Doxygen will generate the necessary Makefile rules, Perl scripts and LaTeX code to be able to generate PDF and DVI output from the Perl module output.

PERLMOD_PRETTY If the PERLMOD_PRETTY tag is set to YES the Perl module output will be nicely formatted so it can be parsed by a human reader. This is useful if you want to understand what is going on. On the other hand, if this tag is set to NO the size of the Perl module output will be much smaller and Perl will parse it just the same.

PERLMOD_MAKEVAR_PREFIX The names of the make variables in the generated doxyrules.make file are prefixed with the string contained in PERLMOD_MAKEVAR_PREFIX. This is useful so different doxyrules.make files included by the same Makefile don't overwrite each other's variables.

19.15 Preprocessor related options

ENABLE_PREPROCESSING If the ENABLE_PREPROCESSING tag is set to YES (the default) doxygen will evaluate all C-preprocessor directives found in the sources and include files.

MACRO_EXPANSION If the MACRO_EXPANSION tag is set to YES doxygen will expand all macro names in the source code. If set to NO (the default) only conditional compilation will be performed. Macro expansion can be done in a controlled way by setting EXPAND_ONLY_PREDEF to YES.

EXPAND_ONLY_PREDEF If the EXPAND_ONLY_PREDEF and MACRO_EXPANSION tags are both set to YES then the macro expansion is limited to the macros specified with the PREDEFINED and EXPAND_AS_DEFINED tags.

SEARCH_INCLUDES If the SEARCH_INCLUDES tag is set to YES (the default) the includes files in the INCLUDE_PATH (see below) will be searched if a #include is found.

INCLUDE_PATH The INCLUDE_PATH tag can be used to specify one or more directories that contain include files that are not input files but should be processed by the preprocessor.

PREDEFINED The PREDEFINED tag can be used to specify one or more macro names that are defined before the preprocessor is started (similar to the -D option of gcc). The argument of the tag is a list of macros of the form: `name` or `name=definition` (no spaces). If the definition and the "=" are omitted, "=1" is assumed.

EXPAND_AS_DEFINED If the MACRO_EXPANSION and EXPAND_ONLY_PREDEF tags are set to YES then this tag can be used to specify a list of macro names that should be expanded. The macro definition that is found in the sources will be used. Use the PREDEFINED tag if you want to use a different macro definition.

SKIP_FUNCTION_MACROS If the SKIP_FUNCTION_MACROS tag is set to YES (the default) then doxygen's preprocessor will remove all function-like macros that are alone on a line, have an all uppercase name, and do not end with a semicolon. Such function macros are typically used for boiler-plate code, and will confuse the parser if not removed.

19.16 External reference options

TAGFILES The TAGFILES tag can be used to specify one or more tagfiles.

See section Doxytag usage for more information about the usage of tag files.

Optionally an initial location of the external documentation can be added for each tagfile. The format of a tag file without this location is as follows:

```
TAGFILES = file1 file2 ...
```

Adding location for the tag files is done as follows:

```
TAGFILES = file1=loc1 "file2 = loc2" ...
```

where loc1 and loc2 can be relative or absolute paths or URLs, If a location is present for each tag, the installdox tool (see section Installdox usage for more information) does not have to be run to correct the links.

Note:
Each tag file must have a unique name (where the name does *not* include the path) If a tag file is not located in the directory in which doxygen is run, you must also specify the path to the tagfile here.

GENERATE_TAGFILE When a file name is specified after GENERATE_TAGFILE, doxygen will create a tag file that is based on the input files it reads. See section Doxytag usage for more information about the usage of tag files.

ALLEXTERNALS If the ALLEXTERNALS tag is set to YES all external class will be listed in the class index. If set to NO only the inherited external classes will be listed.

EXTERNAL_GROUPS If the EXTERNAL_GROUPS tag is set to YES all external groups will be listed in the modules index. If set to NO, only the current project's groups will be listed.

PERL_PATH The PERL_PATH should be the absolute path and name of the perl script interpreter (i.e. the result of 'which perl').

19.17 Dot options

CLASS_DIAGRAMS If the CLASS_DIAGRAMS tag is set to YES (the default) doxygen will generate a class diagram (in HTML and LaTeX) for classes with base or super classes. Setting the tag to NO turns the diagrams off. Note that this option is superseded by the HAVE_DOT option below. This is only a fallback. It is recommended to install and use dot, since it yields more powerful graphs.

HAVE_DOT If you set the HAVE_DOT tag to YES then doxygen will assume the dot tool is available from the path. This tool is part of Graphviz, a graph visualization toolkit from AT&T and Lucent Bell Labs. The other options in this section have no effect if this option is set to NO (the default)

CLASS_GRAPH If the CLASS_GRAPH and HAVE_DOT tags are set to YES then doxygen will generate a graph for each documented class showing the direct and indirect inheritance relations. Setting this tag to YES will force the the CLASS_DIAGRAMS tag to NO.

COLLABORATION_GRAPH If the COLLABORATION_GRAPH and HAVE_DOT tags are set to YES then doxygen will generate a graph for each documented class showing the direct and indirect implementation dependencies (inheritance, containment, and class references variables) of the class with other documented classes.

TEMPLATE_RELATIONS If the TEMPLATE_RELATIONS and HAVE_DOT tags are set to YES then doxygen will show the relations between templates and their instances.

HIDE_UNDOC_RELATIONS If set to YES, the inheritance and collaboration graphs will hide inheritance and usage relations if the target is undocumented or is not a class.

INCLUDE_GRAPH If the ENABLE_PREPROCESSING, SEARCH_INCLUDES, INCLUDE_GRAPH, and HAVE_DOT tags are set to YES then doxygen will generate a graph for each documented file showing the direct and indirect include dependencies of the file with other documented files.

INCLUDED_BY_GRAPH If the ENABLE_PREPROCESSING, SEARCH_INCLUDES, INCLUDED_BY_GRAPH, and HAVE_DOT tags are set to YES then doxygen will generate a graph for each documented header file showing the documented files that directly or indirectly include this file.

CALL_GRAPH If the CALL_GRAPH and HAVE_DOT tags are set to YES then doxygen will generate a call dependency graph for every global function or class method. Note that enabling this option will significantly increase the time of a run. So in most cases it will be better to enable call graphs for selected functions only using the \callgraph command.

GRAPHICAL_HIERARCHY If the GRAPHICAL_HIERARCHY and HAVE_DOT tags are set to YES then doxygen will graphical hierarchy of all classes instead of a textual one.

DOT_IMAGE_FORMAT The DOT_IMAGE_FORMAT tag can be used to set the image format of the images generated by dot. Possible values are gif, jpg, and png. If left blank png will be used.

DOT_PATH This tag can be used to specify the path where the dot tool can be found. If left blank, it is assumed the dot tool can be found on the path.

DOTFILE_DIRS This tag can be used to specify one or more directories that contain dot files that are included in the documentation (see the \dotfile command).

MAX_DOT_GRAPH_HEIGHT The MAX_DOT_GRAPH_HEIGHT tag can be used to set the maximum allows height (in pixels) of the graphs generated by dot. If a graph becomes larger than this value, doxygen will try to truncate the graph, so that it fits within the specified constraint. Beware that most browsers cannot cope with very large images.

MAX_DOT_GRAPH_DEPTH The MAX_DOT_GRAPH_DEPTH tag can be used to set the maximum depth of the graphs generated by dot. A depth value of 3 means that only nodes reachable from the root by following a path via at most 3 edges will be shown. Nodes that lay further from the root node will be omitted. Note that setting this option to 1 or 2 may greatly reduce the computation time needed for large code bases. Also note that a graph may be further truncated if the graph's image dimensions are not sufficient to fit the graph (see MAX_DOT_GRAPH_WIDTH and MAX_DOT_GRAPH_HEIGHT). If 0 is used for the depth value (the default), the graph is not depth-constraint.

MAX_DOT_GRAPH_WIDTH The MAX_DOT_GRAPH_WIDTH tag can be used to set the maximum allowed width (in pixels) of the graphs generated by dot. If a graph becomes larger than this value, doxygen will try to truncate the graph, so that it fits within the specified constraint. Beware that most browsers cannot cope with very large images.

GENERATE_LEGEND If the GENERATE_LEGEND tag is set to YES (the default) doxygen will generate a legend page explaining the meaning of the various boxes and arrows in the dot generated graphs.

DOT_CLEANUP If the DOT_CLEANUP tag is set to YES (the default) doxygen will remove the intermediate dot files that are used to generate the various graphs.

19.18 Search engine options

SEARCHENGINE The SEARCHENGINE tag specifies whether or not the HTML output should contain a search facility. Possible values are YES and NO. If set to YES, doxygen will produce a search index and a PHP script to search through the index. For this to work the documentation should be viewed via a web-server running PHP version 4.1.0 or higher. (See http://www.php.net/manual/en/installation.php for installation instructions).

Examples

Suppose you have a simple project consisting of two files: a source file example.cc and a header file example.h. Then a minimal configuration file is as simple as:

```
INPUT            = example.cc example.h
```

Assuming the example makes use of Qt classes and perl is located in `/usr/bin`, a more realistic configuration file would be:

```
PROJECT_NAME     = Example
INPUT            = example.cc example.h
WARNINGS         = YES
TAGFILES         = qt.tag
PERL_PATH        = /usr/bin/perl
SEARCHENGINE     = NO
```

To generate the documentation for the `QdbtTabular` package I have used the following configuration file:

```
PROJECT_NAME     = QdbtTabular
OUTPUT_DIRECTORY = html
WARNINGS         = YES
INPUT            = examples/examples.doc src
FILE_PATTERNS    = *.cc *.h
INCLUDE_PATH     = examples
TAGFILES         = qt.tag
PERL_PATH        = /usr/local/bin/perl
SEARCHENGINE     = YES
```

To regenerate the Qt-1.44 documentation from the sources, you could use the following config file:

```
PROJECT_NAME         = Qt
OUTPUT_DIRECTORY     = qt_docs
HIDE_UNDOC_MEMBERS   = YES
HIDE_UNDOC_CLASSES   = YES
ENABLE_PREPROCESSING = YES
MACRO_EXPANSION      = YES
EXPAND_ONLY_PREDEF   = YES
SEARCH_INCLUDES      = YES
FULL_PATH_NAMES      = YES
STRIP_FROM_PATH      = $(QTDIR)/
PREDEFINED           = USE_TEMPLATECLASS Q_EXPORT= \
                       QArrayT:=QArray \
                       QListT:=QList \
                       QDictT:=QDict \
                       QQueueT:=QQueue \
                       QVectorT:=QVector \
                       QPtrDictT:=QPtrDict \
                       QIntDictT:=QIntDict \
                       QStackT:=QStack \
                       QDictIteratorT:=QDictIterator \
                       QListIteratorT:=QListIterator \
                       QCacheT:=QCache \
                       QCacheIteratorT:=QCacheIterator \
                       QIntCacheT:=QIntCache \
                       QIntCacheIteratorT:=QIntCacheIterator \
                       QIntDictIteratorT:=QIntDictIterator \
                       QPtrDictIteratorT:=QPtrDictIterator
INPUT                = $(QTDIR)/doc \
                       $(QTDIR)/src/widgets \
                       $(QTDIR)/src/kernel \
                       $(QTDIR)/src/dialogs \
                       $(QTDIR)/src/tools
FILE_PATTERNS        = *.cpp *.h q*.doc
INCLUDE_PATH         = $(QTDIR)/include
RECURSIVE            = YES
```

For the Qt-2.1 sources I recommend to use the following settings:

```
PROJECT_NAME           = Qt
PROJECT_NUMBER         = 2.1
HIDE_UNDOC_MEMBERS     = YES
HIDE_UNDOC_CLASSES     = YES
SOURCE_BROWSER         = YES
INPUT                  = $(QTDIR)/src
FILE_PATTERNS          = *.cpp *.h q*.doc
RECURSIVE              = YES
EXCLUDE_PATTERNS       = *codec.cpp moc_* */compat/* */3rdparty/*
ALPHABETICAL_INDEX     = YES
COLS_IN_ALPHA_INDEX    = 3
IGNORE_PREFIX          = Q
ENABLE_PREPROCESSING   = YES
MACRO_EXPANSION        = YES
INCLUDE_PATH           = $(QTDIR)/include
PREDEFINED             = Q_PROPERTY(x)= \
                         Q_OVERRIDE(x)= \
                         Q_EXPORT= \
                         Q_ENUMS(x)= \
                         "QT_STATIC_CONST=static const " \
                         _WS_X11_ \
                         INCLUDE_MENUITEM_DEF
EXPAND_ONLY_PREDEF     = YES
EXPAND_AS_DEFINED      = Q_OBJECT_FAKE Q_OBJECT ACTIVATE_SIGNAL_WITH_PARAM \
                         Q_VARIANT_AS
```

Here doxygen's preprocessor is used to substitute some macro names that are normally substituted by the C preprocessor, but without doing full macro expansion.

20 Special Commands

20.1 Introduction

All commands in the documentation start with a backslash (\) or an at-sign (**@**). If you prefer you can replace all commands starting with a backslash below, by their counterparts that start with an at-sign.

Some commands have one or more arguments. Each argument has a certain range:

- If <sharp> braces are used the argument is a single word.
- If (round) braces are used the argument extends until the end of the line on which the command was found.
- If {curly} braces are used the argument extends until the next paragraph. Paragraphs are delimited by a blank line or by a section indicator.

If [square] brackets are used the argument is optional.

Here is an alphabetically sorted list of all commands with references to their documentation:

The following subsections provide a list of all commands that are recognized by doxygen. Unrecognized commands are treated as normal text.

Structural indicators

20.2 \addtogroup <name> [(title)]

Defines a group just like \defgroup, but in contrast to that command using the same <name> more than once will not result in a warning, but rather one group with a merged documentation and the first title found in any of the commands.

The title is optional, so this command can also be used to add a number of entities to an existing group using @{ and @} like this:

```
/*! \addtogroup mygrp
 *  Additional documentation for group 'mygrp'
 *  @{
 */

/*!
 *  A function
 */
void func1()
{
}

/*! Another function */
void func2()
{
}

/*! @} */
```

See also:

page Grouping, sections \defgroup, \ingroup and \weakgroup.

20.3 \callgraph

When this command is put in a comment block of a function or method and HAVE_DOT is set to YES, then doxygen will generate a call graph for that function (provided the implementation of the function or method calls other documented functions). The call graph will generated regardless of the value of CALL_GRAPH.

Note:
The completeness (and correctness) of the call graph depends on the doxygen code parser which is not perfect.

20.4 \class <name> [<header-file>] [<header-name>]

Indicates that a comment block contains documentation for a class with name <name>. Optionally a header file and a header name can be specified. If the header-file is specified, a link to a verbatim copy of the header will be included in the HTML documentation. The <header-name> argument can be used to overwrite the name of the link that is used in the class documentation to something other than <header-file>. This can be useful if the include name is not located on the default include path (like <X11/X.h>). With the <header-name> argument you can also specify how the include statement should look like, by adding either quotes or sharp brackets around the name. Sharp brackets are used if just the name is given.

Example:

```
/* A dummy class */

class Test
{
};

/*! \class Test class.h "inc/class.h"
 *  \brief This is a test class.
 *
 * Some details about the Test class
 */
```

20.5 \def <name>

Indicates that a comment block contains documentation for a `#define` macro.

Example:

```
/*! \file define.h
    \brief testing defines

    This is to test the documentation of defines.
*/

/*!
  \def MAX(x,y)
  Computes the maximum of \a x and \a y.
*/

/*!
   Computes the absolute value of its argument \a x.
*/
#define ABS(x) (((x)>0)?(x):-(x))
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)>(y)?(y):(x)) /*!< Computes the minimum of \a x and \a y. */
```

20.6 \defgroup <name> (group title)

Indicates that a comment block contains documentation for a group of classes, files or namespaces. This can be used to categorize classes, files or namespaces, and document those categories. You can also use groups as members of other groups, thus building a hierarchy of groups.

The <name> argument should be a single-word identifier.

See also:
page Grouping, sections \ingroup, \addtogroup, \weakgroup.

20.7 \enum <name>

Indicates that a comment block contains documentation for an enumeration, with name <name>. If the enum is a member of a class and the documentation block is located outside the class definition, the scope of the class should be specified as well. If a comment block is located directly in front of an enum declaration, the \enum comment may be omitted.

Note:
The type of an anonymous enum cannot be documented, but the values of an anonymous enum can.

Example:

```
class Test
{
  public:
    enum TEnum { Val1, Val2 };

    /*! Another enum, with inline docs */
    enum AnotherEnum
    {
      V1, /*!< value 1 */
      V2  /*!< value 2 */
    };
};

/*! \class Test
 * The class description.
 */

/*! \enum Test::TEnum
 * A description of the enum type.
 */

/*! \var Test::TEnum Test::Val1
 * The description of the first enum value.
 */
```

20.8 \example <file-name>

Indicates that a comment block contains documentation for a source code example. The name of the source file is <file-name>. The text of this file will be included in the documentation, just after the documentation contained in the comment block. All examples are placed in a list. The source code is scanned for documented members and classes. If any are found, the names are cross-referenced with the documentation. Source files or directories can be specified using the EXAMPLE_PATH tag of doxygen's configuration file.

If <file-name> itself is not unique for the set of example files specified by the EXAMPLE_PATH tag, you can include part of the absolute path to disambiguate it.

If more that one source file is needed for the example, the \include command can be used.

Example:

```
/** A Test class.
 *  More details about this class.
 */

class Test
{
  public:
    /** An example member function.
     *  More details about this function.
     */
    void example();
};

void Test::example() {}

/** \example example_test.cpp
 * This is an example of how to use the Test class.
 * More details about this example.
 */
```

Where the example file `example_test.cpp` looks as follows:

```
void main()
{
  Test t;
  t.example();
}
```

See also:

section \include.

20.9 \file [<name>]

Indicates that a comment block contains documentation for a source or header file with name <name>. The file name may include (part of) the path if the file-name alone is not unique. If the file name is omitted (i.e. the line after \file is left blank) then the documentation block that contains the \file command will belong to the file it is located in.

Important:

The documentation of global functions, variables, typedefs, and enums will only be included in the output if the file they are in is documented as well.

Example:

```
/** \file file.h
 * A brief file description.
 * A more elaborated file description.
 */

/**
 * A global integer value.
 * More details about this value.
 */
extern int globalValue;
```

20.10 \fn (function declaration)

Indicates that a comment block contains documentation for a function (either global or as a member of a class). This command is *only* needed if a comment block is *not* placed in front (or behind) the function

declaration or definition.

If your comment block *is* in front of the function declaration or definition this command can (and to avoid redundancy should) be omitted.

A full function declaration including arguments should be specified after the \fn command on a *single* line, since the argument ends at the end of the line!

Warning:
Do not use this command if it is not absolutely needed, since it will lead to duplication of information and thus to errors.

Example:

```
class Test
{
  public:
    const char *member(char,int) throw(std::out_of_range);
};

const char *Test::member(char c,int n) throw(std::out_of_range) {}

/*! \class Test
 * \brief Test class.
 *
 * Details about Test.
 */

/*! \fn const char *Test::member(char c,int n)
 *  \brief A member function.
 *  \param c a character.
 *  \param n an integer.
 *  \exception std::out_of_range parameter is out of range.
 *  \return a character pointer.
 */
```

See also:
section \var and \typedef.

20.11 \hideinitializer

By default the value of a define and the initializer of a variable are displayed unless they are longer than 30 lines. By putting this command in a comment block of a define or variable, the initializer is always hidden.

See also:
section \showinitializer.

20.12 \ingroup (<groupname> [<groupname> <groupname>])

If the \ingroup command is placed in a comment block of a class, file or namespace, then it will be added to the group or groups identified by <groupname>.

See also:
page Grouping, sections \defgroup, \addtogroup and \weakgroup

20.13 \interface

Indicates that a comment block contains documentation for an interface with name <name>. The arguments are equal to the \class command.

See also:
section \class.

20.14 \internal

This command writes the message 'For internal use only' to the output and all text *after* an `\internal` command until the end of the comment block or the end of the section (whichever comes first) is marked as "internal".

If the \internal command is put inside a section (see for example \section) all subsection after the command are considered to be internal as well. Only a new section at the same level will be visible again.

You can use INTERNAL_DOCS in the config file to show or hide the internal documentation.

20.15 \mainpage [(title)]

If the \mainpage command is placed in a comment block the block is used to customize the index page (in HTML) or the first chapter (in LaTeX).

The title argument is optional and replaces the default title that doxygen normally generates. If you do not want any title you can specify `notitle` as the argument of \mainpage.

Here is an example:

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

You can refer to the main page using \ref index (if the treeview is disabled, otherwise you should use \ref main).

See also:
section \section, section \subsection and section \page.

20.16 \name (header)

This command turns a comment block into a header definition of a member group. The comment block should be followed by a `//@{ ... //@}` block containing the members of the group.

See section Member Groups for an example.

20.17 \namespace <name>

Indicates that a comment block contains documentation for a namespace with name <name>.

20.18 \nosubgrouping

This command can be put in the documentation of a class. It can be used in combination with member grouping to avoid that doxygen puts a member group as a subgroup of a Public/Protected/Private/... section.

20.19 \overload [(function declaration)]

This command can be used to generate the following standard text for an overloaded member function:

'This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.'

If the documentation for the overloaded member function is not located in front of the function declaration or definition, the optional argument should be used to specify the correct function.

Any other documentation that is inside the documentation block will by appended after the generated message.

Note 1:

You are responsible that there is indeed an earlier documented member that is overloaded by this one. To prevent that document reorders the documentation you should set SORT_MEMBER_DOCS to NO in this case.

Note 2:

The \overload command does not work inside a one-line comment.

Example:

```
class Test
{
  public:
    void drawRect(int,int,int,int);
    void drawRect(const Rect &r);
};

void Test::drawRect(int x,int y,int w,int h) {}
void Test::drawRect(const Rect &r) {}

/*! \class Test
 *  \brief A short description.
 *
 *  More text.
 */

/*! \fn void Test::drawRect(int x,int y,int w,int h)
 * This command draws a rectangle with a left upper corner at ( \a x , \a y ),
 * width \a w and height \a h.
 */

/*!
 * \overload void Test::drawRect(const Rect &r)
 */
```

20.20 \package <name>

Indicates that a comment block contains documentation for a Java package with name <name>.

20.21 \page <name> (title)

Indicates that a comment block contains a piece of documentation that is not directly related to one specific class, file or member. The HTML generator creates a page containing the documentation. The LaTeX generator starts a new section in the chapter 'Page documentation'.

Example:

```
/*! \page page1 A documentation page
  Leading text.
  \section sec An example section
  This page contains the subsections \ref subsection1 and \ref subsection2.
  For more info see page \ref page2.
  \subsection subsection1 The first subsection
  Text.
  \subsection subsection2 The second subsection
  More text.
*/

/*! \page page2 Another page
  Even more info.
*/
```

Note:

The <name> argument consists of a combination of letters and number digits. If you wish to use upper case letters (e.g. `MYPAGE1`), or mixed case letters (e.g. `MyPage1`) in the <name> argument, you should set `CASE_SENSE_NAMES` to `YES`. However, this is advisable only if your file system is case sensitive. Otherwise (and for better portability) you should use all lower case letters (e.g. `mypage1`) for <name> in all references to the page.

See also:

section \section, section \subsection, and section \ref.

20.22 \relates <name>

This command can be used in the documentation of a non-member function <name>. It puts the function inside the 'related function' section of the class documentation. This command is useful for documenting non-friend functions that are nevertheless strongly coupled to a certain class. It prevents the need of having to document a file, but only works for functions.

Example:

```
/*!
 * A String class.
 */

class String
{
  friend int strcmp(const String &,const String &);
};

/*!
 * Compares two strings.
 */

int strcmp(const String &s1,const String &s2)
{
}

/*! \relates String
 * A string debug function.
```

```
 */

void stringDebug()
{
}
```

20.23 \relatesalso <name>

This command can be used in the documentation of a non-member function <name>. It puts the function both inside the 'related function' section of the class documentation as well as leaving its normal file documentation location. This command is useful for documenting non-friend functions that are nevertheless strongly coupled to a certain class. It only works for functions.

Example:

```
/*!
 * A String class.
 */

class String
{
  friend int strcmp(const String &,const String &);
};

/*!
 * Compares two strings.
 */

int strcmp(const String &s1,const String &s2)
{
}

/*! \relates String
 * A string debug function.
 */

void stringDebug()
{
}
```

20.24 \showinitializer

By default the value of a define and the initializer of a variable are only displayed if they are less than 30 lines long. By putting this command in a comment block of a define or variable, the initializer is shown unconditionally.

See also:
section \hideinitializer.

20.25 \struct <name> [<header-file>] [<header-name>]

Indicates that a comment block contains documentation for a struct with name <name>. The arguments are equal to the \class command.

See also:
section \class.

20.26 \typedef (typedef declaration)

Indicates that a comment block contains documentation for a typedef (either global or as a member of a class). This command is equivalent to \var and \fn.

See also:
section \fn and \var.

20.27 \union <name> [<header-file>] [<header-name>]

Indicates that a comment block contains documentation for a union with name <name>. The arguments are equal to the \class command.

See also:
section \class.

20.28 \var (variable declaration)

Indicates that a comment block contains documentation for a variable or enum value (either global or as a member of a class). This command is equivalent to \typedef and \fn.

See also:
section \fn and \typedef.

20.29 \weakgroup <name> [(title)]

Can be used exactly like \addtogroup, but has a lower priority when it comes to resolving conflicting grouping definitions.

See also:
page Grouping and \addtogroup.

Section indicators

20.30 \attention { attention text }

Starts a paragraph where a message that needs attention may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \attention commands will be joined into a single paragraph. The \attention command ends when a blank line or some other sectioning command is encountered.

20.31 \author { list of authors }

Starts a paragraph where one or more author names may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \author commands will be joined into a single paragraph and separated by commas. Alternatively, one \author command may mention several authors. The \author command ends when a blank line or some other sectioning command is encountered.

Example:

```
/*! \class WindowsNT
 *  \brief Windows Nice Try.
 *  \author Bill Gates
 *  \author Several species of small furry animals gathered together
 *          in a cave and grooving with a pict.
 *  \version 4.0
 *  \date    1996-1998
 *  \bug It crashes a lot and requires huge amounts of memory.
 *  \bug The class introduces the more bugs, the longer it is used.
 *  \warning This class may explode in your face.
 *  \warning If you inherit anything from this class, you're doomed.
 */

class WindowsNT {};
```

20.32 \brief {brief description}

Starts a paragraph that serves as a brief description. For classes and files the brief description will be used in lists and at the start of the documentation page. For class and file members, the brief description will be placed at the declaration of the member and prepended to the detailed description. A brief description may span several lines (although it is advised to keep it brief!). A brief description ends when a blank line or another sectioning command is encountered. If multiple \brief commands are present they will be joined. See section \author for an example.

Synonymous to \short.

20.33 \bug { bug description }

Starts a paragraph where one or more bugs may be reported. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \bug commands will be joined into a single paragraph. Each bug description will start on a new line. Alternatively, one \bug command may mention several bugs. The \bug command ends when a blank line or some other sectioning command is encountered. See section \author for an example.

20.34 \date { date description }

Starts a paragraph where one or more dates may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \date commands will be joined into a single paragraph. Each date description will start on a new line. Alternatively, one \date command may mention several dates. The \date command ends when a blank line or some other sectioning command is encountered. See section \author for an example.

20.35 \deprecated { description }

Starts a paragraph indicating that this documentation block belongs to a deprecated entity. Can be used to describe alternatives, expected life span, etc.

20.36 \else

Starts a conditional section if the previous conditional section was not enabled. The previous section should have been started with a \if, \ifnot, or \elseif command.

See also:
\if, \ifnot, \elseif, \endif.

20.37 \elseif <section-label>

Starts a conditional documentation section if the previous section was not enabled. A conditional section is disabled by default. To enable it you must put the section-label after the ENABLED_SECTIONS tag in the configuration file. Conditional blocks can be nested. A nested section is only enabled if all enclosing sections are enabled as well.

See also:
sections \endif, \ifnot, \else, and \elseif.

20.38 \endif

Ends a conditional section that was started with \if or \ifnot For each \if or \ifnot one and only one matching \endif must follow.

See also:
\if, and \ifnot.

20.39 \exception <exception-object> { exception description }

Starts an exception description for an exception object with name <exception-object>. Followed by a description of the exception. The existence of the exception object is not checked. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \exception commands will be joined into a single paragraph. Each parameter description will start on a new line. The \exception description ends when a blank line or some other sectioning command is encountered. See section \fn for an example.

Note:
the tag \exceptions is a synonym for this tag.

20.40 \if <section-label>

Starts a conditional documentation section. The section ends with a matching \endif command. A conditional section is disabled by default. To enable it you must put the section-label after the ENABLED_SECTIONS tag in the configuration file. Conditional blocks can be nested. A nested section is only enabled if all enclosing sections are enabled as well.

Example:

```
/*! Unconditionally shown documentation.
 *  \if Cond1
 *    Only included if Cond1 is set.
 *  \endif
 *  \if Cond2
```

```
*    Only included if Cond2 is set.
*    \if Cond3
*      Only included if Cond2 and Cond3 are set.
*    \endif
*    More text.
*  \endif
*  Unconditional text.
*/
```

You can also use conditional commands inside aliases. To document a class in two languages you could for instance use:

Example 2:

```
/*! \english
 *  This is English.
 *  \endenglish
 *  \dutch
 *  Dit is Nederlands.
 *  \enddutch
 */
class Example
{
};
```

Where the following aliases are defined in the configuration file:

```
ALIASES  = "english=\if english" \
           "endenglish=\endif" \
           "dutch=\if dutch" \
           "enddutch=\endif"
```

and ENABLED_SECTIONS can be used to enable either english or dutch.

See also:
sections \endif, \ifnot, \else, and \elseif.

20.41 \ifnot <section-label>

Starts a conditional documentation section. The section ends with a matching \endif command. This conditional section is enabled by default. To disable it you must put the section-label after the ENABLED_SECTIONS tag in the configuration file.

See also:
sections \endif, \if, \else, and \elseif.

20.42 \invariant { description of invariant }

Starts a paragraph where the invariant of an entity can be described. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \invariant commands will be joined into a single paragraph. Each invariant description will start on a new line. Alternatively, one \invariant command may mention several invariants. The \invariant command ends when a blank line or some other sectioning command is encountered.

20.43 \note { text }

Starts a paragraph where a note can be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \note commands will be joined into a single paragraph. Each note description will start on a new line. Alternatively, one \note command may mention several notes. The \note command ends when a blank line or some other sectioning command is encountered. See section \par for an example.

20.44 \par [(paragraph title)] { paragraph }

If a paragraph title is given this command starts a paragraph with a user defined heading. The heading extends until the end of the line. The paragraph following the command will be indented.

If no paragraph title is given this command will start a new paragraph. This will also work inside other paragraph commands (like \param or \warning) without ending the that command.

The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. The \par command ends when a blank line or some other sectioning command is encountered.

Example:

```
/*! \class Test
 * Normal text.
 *
 * \par User defined paragraph:
 * Contents of the paragraph.
 *
 * \par
 * New paragraph under the same heading.
 *
 * \note
 * This note consists of two paragraphs.
 * This is the first paragraph.
 *
 * \par
 * And this is the second paragraph.
 *
 * More normal text.
 */

class Test {};
```

20.45 \param <parameter-name> { parameter description }

Starts a parameter description for a function parameter with name <parameter-name>. Followed by a description of the parameter. The existence of the parameter is not checked. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \param commands will be joined into a single paragraph. Each parameter description will start on a new line. The \param description ends when a blank line or some other sectioning command is encountered. See section \fn for an example.

20.46 \post { description of the postcondition }

Starts a paragraph where the postcondition of an entity can be described. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used

inside the paragraph. Multiple adjacent \post commands will be joined into a single paragraph. Each post-condition will start on a new line. Alternatively, one \post command may mention several postconditions. The \post command ends when a blank line or some other sectioning command is encountered.

20.47 \pre { description of the precondition }

Starts a paragraph where the precondition of an entity can be described. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \pre commands will be joined into a single paragraph. Each precondition will start on a new line. Alternatively, one \pre command may mention several preconditions. The \pre command ends when a blank line or some other sectioning command is encountered.

20.48 \remarks { remark text }

Starts a paragraph where one or more remarks may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \remark commands will be joined into a single paragraph. Each remark will start on a new line. Alternatively, one \remark command may mention several remarks. The \remark command ends when a blank line or some other sectioning command is encountered.

20.49 \return { description of the return value }

Starts a return value description for a function. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \return commands will be joined into a single paragraph. The \return description ends when a blank line or some other sectioning command is encountered. See section \fn for an example.

20.50 \retval <return value> { description }

Starts a return value description for a function with name <return value>. Followed by a description of the return value. The text of the paragraph that forms the description has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \retval commands will be joined into a single paragraph. Each return value description will start on a new line. The \retval description ends when a blank line or some other sectioning command is encountered.

20.51 \sa { references }

Starts a paragraph where one or more cross-references to classes, functions, methods, variables, files or URL may be specified. Two names joined by either `::` or # are understood as referring to a class and one of its members. One of several overloaded methods or constructors may be selected by including a parenthesized list of argument types after the method name.

Synonymous to \see.

See also:

section autolink for information on how to create links to objects.

20.52 \since { text }

This tag can be used to specify since when (version or time) an entity is available. The paragraph that follows \since does not have any special internal structure. All visual enhancement commands may be used inside the paragraph. The \since description ends when a blank line or some other sectioning command is encountered.

20.53 \test { paragraph describing a test case }

Starts a paragraph where a test case can be described. The description will also add the test case to a separate test list. The two instances of the description will be cross-referenced. Each test case in the test list will be preceded by a header that indicates the origin of the test case.

20.54 \throw <exception-object> { exception description }

Synonymous to \exception (see section \exception).

Note:
the tag \throws is a synonym for this tag.

20.55 \todo { paragraph describing what is to be done }

Starts a paragraph where a TODO item is described. The description will also add an item to a separate TODO list. The two instances of the description will be cross-referenced. Each item in the TODO list will be preceded by a header that indicates the origin of the item.

20.56 \version { version number }

Starts a paragraph where one or more version strings may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \version commands will be joined into a single paragraph. Each version description will start on a new line. Alternatively, one \version command may mention several version strings. The \version command ends when a blank line or some other sectioning command is encountered. See section \author for an example.

20.57 \warning { warning message }

Starts a paragraph where one or more warning messages may be entered. The paragraph will be indented. The text of the paragraph has no special internal structure. All visual enhancement commands may be used inside the paragraph. Multiple adjacent \warning commands will be joined into a single paragraph. Each warning description will start on a new line. Alternatively, one \warning command may mention several warnings. The \warning command ends when a blank line or some other sectioning command is encountered. See section \author for an example.

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

This command is a generalization of commands such as \todo and \bug. It can be used to create user-defined text sections which are automatically cross-referenced between the place of occurrence and a related page, which will be generated. On the related page all sections of the same type will be collected.

The first argument <key> is a identifier uniquely representing the type of the section. The second argument is a quoted string representing the heading of the section under which text passed as the forth argument is put. The third argument (list title) is used as the title for the related page containing all items with the same key. The keys ”todo”, ”test”, ”bug”, and ”deprecated” are predefined.

To get an idea on how to use the \xrefitem command and what its effect is, consider the todo list, which (for English output) can be seen an alias for the command

```
\xrefitem todo "Todo" "Todo List"
```

Since it is very tedious and error-prone to repeat the first three parameters of the command for each section, the command is meant to be used in combination with the ALIASES option in the configuration file. To define a new command \reminder, for instance, one should add the following line to the configuration file:

```
ALIASES += "reminder=\xrefitem reminders \"Reminder\" \"Reminders\""
```

Note the use of escaped quotes for the second and third argument of the \xrefitem command.

Commands to create links

20.59 \addindex (text)

This command adds (text) to the LaTeX index.

20.60 \anchor <word>

This command places an invisible, named anchor into the documentation to which you can refer with the \ref command.

Note:
Anchors can currently only be put into a comment block that is marked as a page (using \page) or mainpage (\mainpage).

See also:
section \ref.

20.61 \endlink

This command ends a link that is started with the \link command.

See also:
section \link.

20.62 \link <link-object>

The links that are automatically generated by doxygen always have the name of the object they point to as link-text.

The \link command can be used to create a link to an object (a file, class, or member) with a user specified link-text. The link command should end with an \endlink command. All text between the \link and \endlink commands serves as text for a link to the <link-object> specified as the first argument of \link.

See section autolink for more information on automatically generated links and valid link-objects.

20.63 \ref <name> ["(text)"]

Creates a reference to a named section, subsection, page or anchor. For HTML documentation the reference command will generate a link to the section. For a sections or subsections the title of the section will be used as the text of the link. For anchor the optional text between quotes will be used or <name> if no text is specified. For LaTeX documentation the reference command will generate a section number for sections or the text followed by a page number if <name> refers to an anchor.

See also:
Section \page for an example of the \ref command.

20.64 \section <section-name> (section title)

Creates a section with name <section-name>. The title of the section should be specified as the second argument of the \section command.

Warning:
This command only works inside related page documentation and *not* in other documentation blocks!

20.65 \subsection <subsection-name> (subsection title)

Creates a subsection with name <subsection-name>. The title of the subsection should be specified as the second argument of the \subsection command.

Warning:
This command only works inside a section of a related page documentation block and *not* in other documentation blocks!

See also:
Section \page for an example of the \subsection command.

20.66 \subsubsection <subsubsection-name> (subsubsection title)

Creates a subsubsection with name <subsubsection-name>. The title of the subsubsection should be specified as the second argument of the \subsubsection command.

Warning:
This command only works inside a subsection of a related page documentation block and *not* in other documentation blocks!

See also:
Section \page for an example of the \subsubsection command.

20.67 \paragraph <paragraph-name> (paragraph title)

Creates a named paragraph with name <paragraph-name>. The title of the paragraph should be specified as the second argument of the \paragraph command.

Warning:
This command only works inside a subsubsection of a related page documentation block and *not* in other documentation blocks!

See also:
Section \page for an example of the \paragraph command.

Commands for displaying examples

20.68 \dontinclude <file-name>

This command can be used to parse a source file without actually verbatim including it in the documentation (as the \include command does). This is useful if you want to divide the source file into smaller pieces and add documentation between the pieces. Source files or directories can be specified using the EXAMPLE_PATH tag of doxygen's configuration file.

The class and member declarations and definitions inside the code fragment are 'remembered' during the parsing of the comment block that contained the \dontinclude command.

For line by line descriptions of source files, one or more lines of the example can be displayed using the \line, \skip, \skipline, and \until commands. An internal pointer is used for these commands. The \dontinclude command sets the pointer to the first line of the example.

Example:

```
/*! A test class. */

class Test
{
  public:
    /// a member function
    void example();
};

/*! \page example
 *  \dontinclude example_test.cpp
 *  Our main function starts like this:
 *  \skip main
 *  \until {
 *  First we create a object \c t of the Test class.
 *  \skipline Test
 *  Then we call the example member function
 *  \line example
 *  After that our little test routine ends.
 *  \line }
 */
```

Where the example file `example_test.cpp` looks as follows:

```
void main()
{
  Test t;
  t.example();
}
```

See also:
sections \line, \skip, \skipline, and \until.

20.69 \include <file-name>

This command can be used to include a source file as a block of code. The command takes the name of an include file as an argument. Source files or directories can be specified using the EXAMPLE_PATH tag of doxygen's configuration file.

If <file-name> itself is not unique for the set of example files specified by the EXAMPLE_PATH tag, you can include part of the absolute path to disambiguate it.

Using the \include command is equivalent to inserting the file into the documentation block and surrounding it with \code and \endcode commands.

The main purpose of the \include command is to avoid code duplication in case of example blocks that consist of multiple source and header files.

For a line by line description of a source files use the \dontinclude command in combination with the \line, \skip, \skipline, and \until commands.

See also:
section \example and \dontinclude.

20.70 \line (pattern)

This command searches line by line through the example that was last included using \include or \dontinclude until it finds a non-blank line. If that line contains the specified pattern, it is written to the output.

The internal pointer that is used to keep track of the current line in the example, is set to the start of the line following the non-blank line that was found (or to the end of the example if no such line could be found).

See section \dontinclude for an example.

20.71 \skip (pattern)

This command searches line by line through the example that was last included using \include or \dontinclude until it finds a line that contains the specified pattern.

The internal pointer that is used to keep track of the current line in the example, is set to the start of the line that contains the specified pattern (or to the end of the example if the pattern could not be found).

See section \dontinclude for an example.

20.72 \skipline (pattern)

This command searches line by line through the example that was last included using \include or \dontinclude until it finds a line that contains the specified pattern. It then writes the line to the output.

The internal pointer that is used to keep track of the current line in the example, is set to the start of the line following the line that is written (or to the end of the example if the pattern could not be found).

Note:
The command:

```
\skipline pattern
```

is equivalent to:

```
\skip pattern
\line pattern
```

See section \dontinclude for an example.

20.73 \until (pattern)

This command writes all lines of the example that was last included using \include or \dontinclude to the output, until it finds a line containing the specified pattern. The line containing the pattern will be written as well.

The internal pointer that is used to keep track of the current line in the example, is set to the start of the line following last written line (or to the end of the example if the pattern could not be found).

See section \dontinclude for an example.

20.74 \verbinclude <file-name>

This command includes the file <file-name> verbatim in the documentation. The command is equivalent to pasting the file in the documentation and placing \verbatim and \endverbatim commands around it.

Files or directories that doxygen should look for can be specified using the EXAMPLE_PATH tag of doxygen's configuration file.

20.75 \htmlinclude <file-name>

This command includes the file <file-name> as is in the HTML documentation. The command is equivalent to pasting the file in the documentation and placing \htmlonly and \endhtmlonly commands around it.

Files or directories that doxygen should look for can be specified using the EXAMPLE_PATH tag of doxygen's configuration file.

Commands for visual enhancements

20.76 \a <word>

Displays the argument <word> using a special font. Use this command to refer to member arguments in the running text.

Example:

```
... the \a x and \a y coordinates are used to ...
```

This will result in the following text:
... the *x* and *y* coordinates are used to ...

20.77 \arg { item-description }

This command has one argument that continues until the first blank line or until another \arg is encountered. The command can be used to generate a simple, not nested list of arguments. Each argument should start with a \arg command.

Example:

Typing:

```
\arg \c AlignLeft left alignment.
\arg \c AlignCenter center alignment.
\arg \c AlignRight right alignment
```

```
No other types of alignment are supported.
```

will result in the following text:

- `AlignLeft` left alignment.
- `AlignCenter` center alignment.
- `AlignRight` right alignment

No other types of alignment are supported.

Note:
For nested lists, HTML commands should be used.

Equivalent to \li

20.78 \b <word>

Displays the argument <word> using a bold font. Equivalent to word. To put multiple words in bold use multiple words.

20.79 \c <word>

Displays the argument <word> using a typewriter font. Use this to refer to a word of code. Equivalent to <tt>word</tt>.

Example:
Typing:

```
... This function returns \c void and not \c int ...
```

will result in the following text:
... This function returns `void` and not `int` ...

Equivalent to \p To have multiple words in typewriter font use <tt>multiple words</tt>.

20.80 \code

Starts a block of code. A code block is treated differently from ordinary text. It is interpreted as C/C++ code. The names of the classes and members that are documented are automatically replaced by links to the documentation.

See also:
section \endcode, section \verbatim

20.81 \copydoc <link-object>

Copies a documentation block from the object specified by <link-object> and pastes it at the location of the command. This command can be useful to avoid cases where a documentation block would otherwise have to be duplicated or it can be used to extend the documentation of an inherited member.

The link object can point to a member (of a class, file or group), a class, a namespace, a group, a page, or a file (checked in that order). Note that if the object pointed to is a member (function, variable, typedef, etc), the compound (class, file, or group) containing it should also be documented for the copying to work.

To copy the documentation for a member of a class for instance one can put the following in the documentation

```
/*! @copydoc MyClass::myfunction()
 *  More documentation.
 */
```

if the member is overloaded, you should specify the argument types explicitly (without spaces!), like in the following:

```
/*! @copydoc MyClass::myfunction(type1,type2) */
```

Qualified names are only needed if the context in which the documentation block is found requires them.

The copydoc command can be used recursively, but cycles in the copydoc relation will be broken and flagged as an error.

20.82 \dot

Starts a text fragment which should contain a valid description of a dot graph. The text fragment ends with \enddot. Doxygen will pass the text on to dot and include the resulting image (and image map) into the output. The nodes of a graph can be made clickable by using the URL attribute. By using the command \ref inside the URL value you can conveniently link to an item inside doxygen. Here is an example:

```
/*! class B */
class B {};

/*! class C */
class C {};

/*! \mainpage

  Class relations expressed via an inline dot graph:
  \dot
  digraph example {
      node [shape=record, fontname=Helvetica, fontsize=10];
      b [ label="class B" URL="\ref B"];
      c [ label="class C" URL="\ref C"];
      b -> c [ arrowhead="open", style="dashed" ];
  }
  \enddot
  Note that the classes in the above graph are clickable
  (in the HTML output).
 */
```

20.83 \dotfile <file> ["caption"]

Inserts an image generated by dot from <file> into the documentation.

The first argument specifies the file name of the image. doxygen will look for files in the paths (or files) that you specified after the DOTFILE_DIRS tag. If the dot file is found it will be used as an input file to the dot tool. The resulting image will be put into the correct output directory. If the dot file name contains spaces you'll have to put quotes (") around it.

The second argument is optional and can be used to specify the caption that is displayed below the image. This argument has to be specified between quotes even if it does not contain any spaces. The quotes are stripped before the caption is displayed.

20.84 \e <word>

Displays the argument <word> in italics. Use this command to emphasize words.

Example:

Typing:

```
... this is a \e really good example ...
```

will result in the following text:
... this is a *really* good example ...

Equivalent to \em. To emphasis multiple words use multiple words.

20.85 \em <word>

Displays the argument <word> in italics. Use this command to emphasize words.

Example:

Typing:

```
... this is a \em really good example ...
```

will result in the following text:
... this is a *really* good example ...

Equivalent to \e

20.86 \endcode

Ends a block of code.

See also:

section \code

20.87 \enddot

Ends a blocks that was started with \dot.

20.88 \endhtmlonly

Ends a block of text that was started with a \htmlonly command.

See also:

section \htmlonly.

20.89 \endlatexonly

Ends a block of text that was started with a \latexonly command.

See also:
section \latexonly.

20.90 \endverbatim

Ends a block of text that was started with a \verbatim command.

See also:
section \verbatim.

20.91 \endxmlonly

Ends a block of text that was started with a \xmlonly command.

See also:
section \xmlonly.

20.92 \f$

Marks the start and end of an in-text formula.

See also:
section formulas for an example.

20.93 \f[

Marks the start of a long formula that is displayed centered on a separate line.

See also:
section \f] and section formulas.

20.94 \f]

Marks the end of a long formula that is displayed centered on a separate line.

See also:
section \f[and section formulas.

20.95 \htmlonly

Starts a block of text that will be verbatim included in the generated HTML documentation only. The block ends with a endhtmlonly command.

This command can be used to include HTML code that is too complex for doxygen (i.e. applets, javascripts, and HTML tags that require attributes). You can use the \latexonly and \endlatexonly pair to provide a proper LaTeX alternative.

Note: environment variables (like $(HOME)) are resolved inside a HTML-only block.

See also:
section \htmlonly and section \latexonly.

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

Inserts an image into the documentation. This command is format specific, so if you want to insert an image for more than one format you'll have to repeat this command for each format.

The first argument specifies the output format. Currently, the following values are supported: `html` and `latex`.

The second argument specifies the file name of the image. doxygen will look for files in the paths (or files) that you specified after the IMAGE_PATH tag. If the image is found it will be copied to the correct output directory. If the image name contains spaces you'll have to put quotes (") around it. You can also specify an absolute URL instead of a file name, but then doxygen does not copy the image nor check its existance.

The third argument is optional and can be used to specify the caption that is displayed below the image. This argument has to be specified between quotes even if it does not contain any spaces. The quotes are stripped before the caption is displayed.

The fourth argument is also optional and can be used to specify the width or height of the image. This is only useful for LaTeX output (i.e. format=`latex`). The `sizeindication` can be either `width` or `height`. The size should be a valid size specifier in LaTeX (for example `10cm` or `6in` or a symbolic width like \`textwidth`).

Here is example of a comment block:

```
/*! Here is a snapshot of my new application:
 *  \image html application.jpg
 *  \image latex application.eps "My application" width=10cm
 */
```

And this is an example of how the relevant part of the configuration file may look:

```
IMAGE_PATH     = my_image_dir
```

Warning:
The image format for HTML is limited to what your browser supports. For LaTeX, the image format must be Encapsulated PostScript (eps).
Doxygen does not check if the image is in the correct format. So *you* have to make sure this is the case!

20.97 \latexonly

Starts a block of text that will be verbatim included in the generated LaTeX documentation only. The block ends with a endlatexonly command.

This command can be used to include LaTeX code that is too complex for doxygen (i.e. images, formulas, special characters). You can use the \htmlonly and \endhtmlonly pair to provide a proper HTML alternative.

Note: environment variables (like $(HOME)) are resolved inside a LaTeX-only block.

See also:
section \latexonly and section \htmlonly.

20.98 \li { item-description }

This command has one argument that continues until the first blank line or until another \li is encountered. The command can be used to generate a simple, not nested list of arguments. Each argument should start with a \li command.

Example:

Typing:

```
\li \c AlignLeft left alignment.
\li \c AlignCenter center alignment.
\li \c AlignRight right alignment

No other types of alignment are supported.
```

will result in the following text:

- `AlignLeft` left alignment.
- `AlignCenter` center alignment.
- `AlignRight` right alignment

No other types of alignment are supported.

Note:

For nested lists, HTML commands should be used.

Equivalent to \arg

20.99 \n

Forces a new line. Equivalent to <br> and inspired by the printf function.

20.100 \p <word>

Displays the parameter <word> using a typewriter font. You can use this command to refer to member function parameters in the running text.

Example:

```
... the \p x and \p y coordinates are used to ...
```

This will result in the following text:
... the `x` and `y` coordinates are used to ...

Equivalent to \c

20.101 \verbatim

Starts a block of text that will be verbatim included in both the HTML and the LaTeX documentation. The block should end with a \endverbatim block. All commands are disabled in a verbatim block.

Warning:

Make sure you include a \endverbatim command for each \verbatim command or the parser will get confused!

20.102 \xmlonly

Starts a block of text that will be verbatim included in the generated XML output only. The block ends with a endxmlonly command.

This command can be used to include custom XML tags.

See also:
section \htmlonly and section \latexonly.

20.103 \\

This command writes a backslash character (\) to the HTML and LaTeX output. The backslash has to be escaped in some cases because doxygen uses it to detect commands.

20.104 \@

This command writes an at-sign (@) to the HTML and LaTeX output. The at-sign has to be escaped in some cases because doxygen uses it to detect JavaDoc commands.

20.105 \~[LanguageId]

This command enables/disables a language specific filter. This can be used to put documentation for different language into one comment block and use the OUTPUT_LANGUAGE tag to filter out only a specific language. Use \~language_id to enable output for a specific language only and \~ to enable output for all languages (this is also the default mode).

Example:

```
/*! \~english This is english \~dutch Dit is Nederlands \~german Dieses ist
    deutsch. \~ output for all languages.
 */
```

20.106 \&

This command writes the & character to the HTML and LaTeX output. This character has to be escaped because it has a special meaning in HTML.

20.107 \$

This command writes the $ character to the HTML and LaTeX output. This character has to be escaped in some cases, because it is used to expand environment variables.

20.108 \#

This command writes the # character to the HTML and LaTeX output. This character has to be escaped in some cases, because it is used to refer to documented entities.

20.109 <

This command writes the < character to the HTML and LaTeX output. This character has to be escaped because it has a special meaning in HTML.

20.110 >

This command writes the > character to the HTML and LaTeX output. This character has to be escaped because it has a special meaning in HTML.

Commands included for Qt compatibility

The following commands are supported to remain compatible to the Qt class browser generator. Do *not* use these commands in your own documentation.

- \annotatedclasslist
- \classhierarchy
- \define
- \functionindex
- \header
- \headerfilelist
- \inherit
- \l
- \postheader

For PHP files there are a number of additional commands, that can be used inside classes to make members public, private, or protected even though the language itself doesn't support this notion.

To mark a single item use one of \private, \protected, \public. For starting a section with a certain protection level use one of: \privatesection, \protectedsection, \publicsection. The latter commands are similar to "private:", "protected:", and "public:" in C++.

21 HTML Commands

Here is a list of all HTML commands that may be used inside the documentation. Note that all attributes of a HTML tag are passed on to the HTML output only (the HREF and NAME attributes for the A tag are the only exception).

- <A HREF="..."> Starts a HTML hyper-link (HTML only).
- <A NAME="..."> Starts an named anchor (HTML only).
- </A> Ends a link or anchor (HTML only).
- <B> Starts a piece of text displayed in a bold font.
- </B> Ends a <B> section.

- <BODY> Does not generate any output.
- </BODY> Does not generate any output.
-
 Forces a line break.
- <CENTER> starts a section of centered text.
- </CENTER> ends a section of centered text.
- <CAPTION> Starts a caption. Use within a table only.
- </CAPTION> Ends a caption. Use within a table only.
- <CODE> Starts a piece of text displayed in a typewriter font.
- </CODE> End a <CODE> section.
- <DD> Starts an item description.
- <DFN> Starts a piece of text displayed in a typewriter font.
- </DFN> Ends a <DFN> section.
- <DL> Starts a description list.
- </DL> Ends a description list.
- <DT> Starts an item title.
- </DT> Ends an item title.
- <EM> Starts a piece of text displayed in an italic font.
- </EM> Ends a <EM> section.
- <FORM> Does not generate any output.
- </FORM> Does not generate any output.
- <HR> Writes a horizontal ruler.
- <H1> Starts an unnumbered section.
- </H1> Ends an unnumberd section.
- <H2> Starts an unnumbered subsection.
- </H2> Ends an unnumbered subsection.
- <H3> Starts an unnumbered subsubsection.
- </H3> Ends an unnumbered subsubsection.
- <I> Starts a piece of text displayed in an italic font.
- <INPUT> Does not generate any output.
- </I> Ends a <I> section.
- <IMG> This command is written with attributes to the HTML output only.
- <LI> Starts a new list item.
- </LI> Ends a list item.

- <META> Does not generate any output.
- <MULTICOL> ignored by doxygen.
- </MUTLICOL> ignored by doxygen.
- <OL> Starts a numbered item list.
- </OL> Ends a numbered item list.
- <P> Starts a new paragraph.
- </P> Ends a paragraph.
- <PRE> Starts a preformatted fragment.
- </PRE> Ends a preformatted fragment.
- <SMALL> Starts a section of text displayed in a smaller font.
- </SMALL> Ends a <SMALL> section.
- <STRONG> Starts a section of bold text.
- </STRONG> Ends a section of bold text.
- <SUB> Starts a piece of text displayed in subscript.
- </SUB> Ends a <SUB> section.
- <SUP> Starts a piece of text displayed in superscript.
- </SUP> Ends a </SUP> section.
- <TABLE> starts a table.
- </TABLE> ends a table.
- <TD> Starts a new table data element.
- </TD> Ends a table data element.
- <TR> Starts a new table row.
- </TR> Ends a table row.
- <TT> Starts a piece of text displayed in a typewriter font.
- </TT> Ends a <TT> section.
- <KBD> Starts a piece of text displayed in a typewriter font.
- </KBD> Ends a <KBD> section.
- <UL> Starts an unnumbered item list.
- </UL> Ends an unnumbered item list.
- <VAR> Starts a piece of text displayed in an italic font.
- </VAR> Ends a </VAR> section.

The special HTML character entities that are recognized by Doxygen:

- `&copy;` the copyright symbol
- `&quot;` a double quote
- `&?uml;` where ? is one of {A,E,I,O,U,Y,a,e,i,o,u,y}, writes a character with a diaeresis accent (like ä).
- `&?acute;` where ? is one of {A,E,I,O,U,Y,a,e,i,o,u,y}, writes a character with a acute accent (like á).
- `&?grave;` where ? is one of {A,E,I,O,U,a,e,i,o,u,y}, writes a character with a grave accent (like à).
- `&?circ;` where ? is one of {A,E,I,O,U,a,e,i,o,u,y}, writes a character with a circumflex accent (like â).
- `&?tilde;` where ? is one of {A,N,O,a,n,o}, writes a character with a tilde accent (like ã).
- `&szlig;` write a sharp s (i.e. ”s) to the output.
- `&?cedil;` where ? is one of {c,C}, writes a c-cedille (like ç).
- `&?ring;` where ? is one of {a,A}, writes an `a` with a ring (like å).
- ` ` a non breakable space.

Finally, to put invisible comments inside comment blocks, HTML style comments can be used:

```
/*! <!-- This is a comment with a comment block --> Visible text */
```

Part III

Developers Manual

22 Doxygen’s Internals

Doxygen’s internals Note that this section is still under construction!

The following picture shows how source files are processed by doxygen.

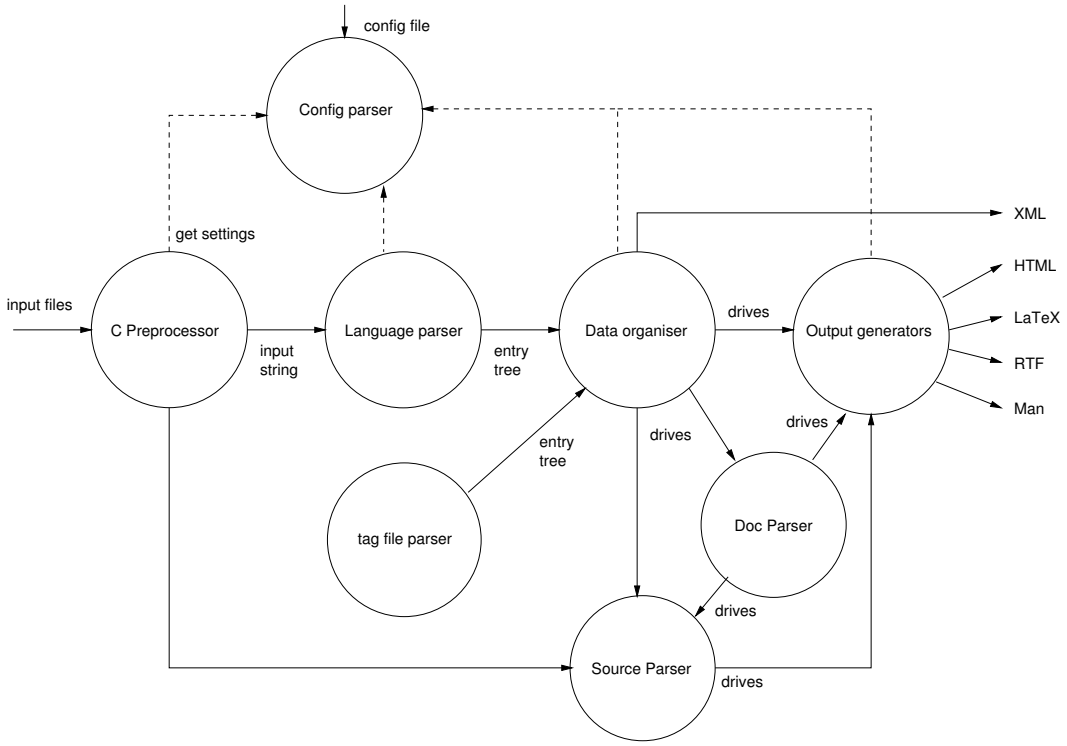

Figure 2: Data flow overview

The following sections explain the steps above in more detail.

Config parser The configuration file that controls the settings of a project is parsed and the settings are stored in the singleton class `Config` in `src/config`.h. The parser itself is written using `flex` and can be found in `src/config`.l. This parser is also used directly by `doxywizard`, so it is put in a separate library.

Each configuration option has one of 5 possible types: `String`, `List`, `Enum`, `Int`, or `Bool`. The values of these options are available through the global functions `Config_getXXX()`, where `XXX` is the type of the option. The argument of these function is a string naming the option as it appears in the configuration file. For instance: `Config_getBool`("GENERATE_TESTLIST") returns a reference to a boolean value that is `TRUE` if the test list was enabled in the config file.

The function `readConfiguration()` in `src/doxygen`.cpp reads the command line options and then calls the configuration parser.

C Preprocessor The input files mentioned in the config file are (by default) fed to the C Preprocessor (after being piped through a user defined filter if available).

The way the preprocessor works differs somewhat from a standard C Preprocessor. By default it does not do macro expansion, although it can be configured to expand all macros. Typical usage is to only expand a user specified set of macros. This is to allow macro names to appear in the type of function parameters for instance.

Another difference is that the preprocessor parses, but not actually includes code when it encounters a #include (with the exception of #include found inside { ... } blocks). The reasons behind this deviation from the standard is to prevent feeding multiple definitions of the same functions/classes to doxygen's

parser. If all source files would include a common header file for instance, the class and type definitions (and their documentation) would be present in each translation unit.

The preprocessor is written using `flex` and can be found in `src/pre`.l. For condition blocks (#if) evaluation of constant expressions is needed. For this a `yacc` based parser is used, which can be found in `src/constexp`.y and `src/constexp`.l.

The preprocessor is invoked for each file using the `preprocessFile()` function declared in `src/pre`.h, and will append the preprocessed result to a character buffer. The format of the character buffer is

```
0x06 file name 1
0x06 preprocessed contents of file 1
...
0x06 file name n
0x06 preprocessed contents of file n
```

Language parser The preprocessed input buffer is fed to the language parser, which is implemented as a big state machine using `flex`. It can be found in the file `src/scanner`.l. There is one parser for all languages (C/C++/Java/IDL). The state variables `insideIDL` and `insideJava` are uses at some places for language specific choices.

The task of the parser is to convert the input buffer into a tree of entries (basically an abstract syntax tree). An entry is defined in `src/entry`.h and is a blob of loosely structured information. The most important field is `section` which specifies the kind of information contained in the entry.

Possible improvements for future versions:

- Use one scanner/parser per language instead of one big scanner.
- Move the first pass parsing of documentation blocks to a separate module.
- Parse defines (these are currently gathered by the preprocessor, and ignored by the language parser).

Data organizer This step consists of many smaller steps, that build dictionaries of the extracted classes, files, namespaces, variables, functions, packages, pages, and groups. Besides building dictionaries, during this step relations (such as inheritance relations), between the extracted entities are computed.

Each step has a function defined in `src/doxygen`.cpp, which operates on the tree of entries, built during language parsing. Look at the ”Gathering information” part of `parseInput()` for details.

The result of this step is a number of dictionaries, which can be found in the Doxygen ”namespace” defined in `src/doxygen`.h. Most elements of these dictionaries are derived from the class `Definition`; The class `MemberDef`, for instance, holds all information for a member. An instance of such a class can be part of a file (class `FileDef`), a class (class `ClassDef`), a namespace (class `NamespaceDef`), a group (class `GroupDef`), or a Java package (class `PackageDef`).

Tag file parser If tag files are specified in the configuration file, these are parsed by a SAX based XML parser, which can be found in `src/tagreader`.cpp. The result of parsing a tag file is the insertion of `Entry` objects in the entry tree. The field `Entry::tagInfo` is used to mark the entry as external, and holds information about the tag file.

Documentation parser Special comment blocks are stored as strings in the entities that they document. There is a string for the brief description and a string for the detailed description. The documentation parser reads these strings and executes the commands it finds in it (this is the second pass in parsing the documentation). It writes the result directly to the output generators.

The parser is written in C++ and can be found in src/docparser.cpp. The tokens that are eaten by the parser come from src/doctokenizer.l. Code fragments found in the comment blocks are passed on to the source parser.

The main entry point for the documentation parser is `validatingParseDoc()` declared in `src/docparser`.h. For simple texts with special commands `validatingParseText()` is used.

Source parser If source browsing is enabled or if code fragments are encountered in the documentation, the source parser is invoked.

The code parser tries to cross-reference to source code it parses with documented entities. It also does syntax highlighting of the sources. The output is directly written to the output generators.

The main entry point for the code parser is `parseCode()` declared in `src/code`.h.

Output generators After data is gathered and cross-referenced, doxygen generates output in various formats. For this it uses the methods provided by the abstract class `OutputGenerator`. In order to generate output for multiple formats at once, the methods of `OutputList` are called instead. This class maintains a list of concrete output generators, where each method called is delegated to all generators in the list.

To allow small deviations in what is written to the output for each concrete output generator, it is possible to temporarily disable certain generators. The OutputList class contains various `disable()` and `enable()` methods for this. The methods `OutputList::pushGeneratorState()` and `OutputList::popGeneratorState()` are used to temporarily save the set of enabled/disabled output generators on a stack.

The XML is generated directly from the gathered data structures. In the future XML will be used as an intermediate language (IL). The output generators will then use this IL as a starting point to generate the specific output formats. The advantage of having an IL is that various independently developed tools written in various languages, could extract information from the XML output. Possible tools could be:

- an interactive source browser
- a class diagram generator
- computing code metrics.

Debugging Since doxygen uses a lot of `flex` code it is important to understand how `flex` works (for this one should read the man page) and to understand what it is doing when `flex` is parsing some input. Fortunately, when flex is used with the -d option it outputs what rules matched. This makes it quite easy to follow what is going on for a particular input fragment.

To make it easier to toggle debug information for a given flex file I wrote the following perl script, which automatically adds or removes -d from the correct line in the Makefile:

```
#!/usr/local/bin/perl

$file = shift @ARGV;
print "Toggle debugging mode for $file\n";

# add or remove the -d flex flag in the makefile
unless (rename "Makefile.libdoxygen","Makefile.libdoxygen.old") {
  print STDERR "Error: cannot rename Makefile.libdoxygen!\n";
  exit 1;
}
if (open(F,"<Makefile.libdoxygen.old")) {
  unless (open(G,">Makefile.libdoxygen")) {
```

```
    print STDERR "Error: opening file Makefile.libdoxygen for writing\n";
    exit 1;
  }
  print "Processing Makefile.libdoxygen...\n";
  while (<F>) {
    if ( s/\(LEX\) -P([a-z]+)YY -t $file/(LEX) -d -P\1YY -t $file/g ) {
      print "Enabling debug info for $file\n";
    }
    elsif ( s/\(LEX\) -d -P([a-z]+)YY -t $file/(LEX) -P\1YY -t $file/g ) {
      print "Disabling debug info for $file\n";
    }
    print G "$_";
  }
  close F;
  unlink "Makefile.libdoxygen.old";
}
else {
  print STDERR "Warning file Makefile.libdoxygen.old does not exist!\n";
}

# touch the file
$now = time;
utime $now, $now, $file
```

23 Perl Module output format documentation

Since version 1.2.18, Doxygen can generate a new output format we have called the ”Perl Module output format”. It has been designed as an intermediate format that can be used to generate new and customized output without having to modify the Doxygen source. Therefore, its purpose is similar to the XML output format that can be also generated by Doxygen. The XML output format is more standard, but the Perl Module output format is possibly simpler and easier to use.

The Perl Module output format is still experimental at the moment and could be changed in incompatible ways in future versions, although this should not be very probable. It is also lacking some features of other Doxygen backends. However, it can be already used to generate useful output, as shown by the Perl Module-based LaTeX generator.

Please report any bugs or problems you find in the Perl Module backend or the Perl Module-based LaTeX generator to the doxygen-develop mailing list. Suggestions are welcome as well.

23.1 Using the Perl Module output format.

When the **GENERATE_PERLMOD** tag is enabled in the Doxyfile, running Doxygen generates a number of files in the **perlmod/** subdirectory of your output directory. These files are the following:

- **DoxyDocs.pm**. This is the Perl module that actually contains the documentation, in the Perl Module format described below.
- **DoxyModel.pm**. This Perl module describes the structure of **DoxyDocs.pm**, independently of the actual documentation. See below for details.
- **doxyrules.make**. This file contains the make rules to build and clean the files that are generated from the Doxyfile. Also contains the paths to those files and other relevant information. This file is intended to be included by your own Makefile.
- **Makefile**. This is a simple Makefile including **doxyrules.make**.

To make use of the documentation stored in DoxyDocs.pm you can use one of the default Perl Module-based generators provided by Doxygen (at the moment this includes the Perl Module-based LaTeX generator, see below) or write your own customized generator. This should not be too hard if you have some knowledge of Perl and it's the main purpose of including the Perl Module backend in Doxygen. See below for details on how to do this.

23.2 Using the Perl Module-based LaTeX generator.

The Perl Module-based LaTeX generator is pretty experimental and incomplete at the moment, but you could find it useful nevertheless. It can generate documentation for functions, typedefs and variables within files and classes and can be customized quite a lot by redefining TeX macros. However, there is still no documentation on how to do this.

Setting the **PERLMOD_LATEX** tag to **YES** in the Doxyfile enables the creation of some additional files in the **perlmod/** subdirectory of your output directory. These files contain the Perl scripts and LaTeX code necessary to generate PDF and DVI output from the Perl Module output, using PDFLaTeX and LaTeX respectively. Rules to automate the use of these files are also added to **doxyrules.make** and the **Makefile**.

The additional generated files are the following:

- **doxylatex.pl**. This Perl script uses DoxyDocs.pm and DoxyModel.pm to generate **doxydocs.tex**, a TeX file containing the documentation in a format that can be accessed by LaTeX code. This file is not directly LaTeXable.
- **doxyformat.tex**. This file contains the LaTeX code that transforms the documentation from doxydocs.tex into LaTeX text suitable to be LaTeX'ed and presented to the user.
- **doxylatex-template.pl**. This Perl script uses DoxyModel.pm to generate **doxytemplate.tex**, a TeX file defining default values for some macros. doxytemplate.tex is included by doxyformat.tex to avoid the need of explicitly defining some macros.
- **doxylatex.tex**. This is a very simple LaTeX document that loads some packages and includes doxyformat.tex and doxydocs.tex. This document is LaTeX'ed to produce the PDF and DVI documentation by the rules added to **doxyrules.make**.

23.2.1 Simple creation of PDF and DVI output using the Perl Module-based LaTeX generator.

To try this you need to have installed LaTeX, PDFLaTeX and the packages used by **doxylatex.tex**.

1. Update your Doxyfile to the latest version using:

   ```
   doxygen -u Doxyfile
   ```

2. Set both **GENERATE_PERLMOD** and **PERLMOD_LATEX** tags to YES in your Doxyfile.
3. Run Doxygen on your Doxyfile:

   ```
   doxygen Doxyfile
   ```

4. A **perlmod/** subdirectory should have appeared in your output directory. Enter the **perlmod/** subdirectory and run:

   ```
   make pdf
   ```

This should generate a **doxylatex.pdf** with the documentation in PDF format.

5. Run:

```
make dvi
```

This should generate a **doxylatex.dvi** with the documentation in DVI format.

23.3 Perl Module documentation format.

The Perl Module documentation generated by Doxygen is stored in **DoxyDocs.pm**. This is a very simple Perl module that contains only two statements: an assigment to the variable **$doxydocs** and the customary **1;** statement which usually ends Perl modules. The documentation is stored in the variable **$doxydocs**, which can then be accessed by a Perl script using **DoxyDocs.pm**.

$doxydocs contains a tree-like structure composed of three types of nodes: strings, hashes and lists.

- **Strings**. These are normal Perl strings. They can be of any length can contain any character. Their semantics depends on their location within the tree. This type of node has no children.
- **Hashes**. These are references to anonymous Perl hashes. A hash can have multiple fields, each with a different key. The value of a hash field can be a string, a hash or a list, and its semantics depends on the key of the hash field and the location of the hash within the tree. The values of the hash fields are the children of the node.
- **Lists**. These are references to anonymous Perl lists. A list has an undefined number of elements, which are the children of the node. Each element has the same type (string, hash or list) and the same semantics, depending on the location of the list within the tree.

As you can see, the documentation contained in **$doxydocs** does not present any special impediment to be processed by a simple Perl script. To be able to generate meaningful output using the documentation contained in **$doxydocs** you'll probably need to know the semantics of the nodes of the documentation tree, which we present in this page.

23.4 Data structure describing the Perl Module documentation tree.

You might be interested in processing the documentation contained in **DoxyDocs.pm** without needing to take into account the semantics of each node of the documentation tree. For this purpose, Doxygen generates a **DoxyModel.pm** file which contains a data structure describing the type and children of each node in the documentation tree.

The rest of this section is to be written yet, but in the meantime you can look at the Perl scripts generated by Doxygen (such as **doxylatex.pl** or **doxytemplate-latex.pl**) to get an idea on how to use **DoxyModel.pm**.

24 Internationalization

Support for multiple languages Doxygen has built-in support for multiple languages. This means that the text fragments that doxygen generates can be produced in languages other than English (the default) at configuration time.

Currently (version 1.3.5), 29 languages are supported (sorted alphabetically): Brazilian Portuguese, Catalan, Chinese, Chinese Traditional, Croatian, Czech, Danish, Dutch, English, Finnish, French, German,

Greek, Hungarian, Italian, Japanese, JapaneseEn, Korean, Norwegian, Polish, Portuguese, Romanian, Russian, Serbian, Slovak, Slovene, Spanish, Swedish, and Ukrainian.

The table of information related to the supported languages follows. It is sorted by language alphabetically. The **Status** column was generated from sources and shows approximately the last version when the translator was updated.

Language	Maintainer	Contact address	Status
Brazilian Portuguese	Fabio ”FJTC” Jun Takada Chino	chino@icmc.sc.usp.br	up-to-date
Catalan	Albert Mora	amora@iua.upf.es	1.2.17
Chinese	Wei Liu Wang Weihan	liuwei@asiainfo.com wangweihan@capinfo.com.cn	1.2.13
Chinese Traditional	Daniel YC Lin Gary Lee	daniel@twpda.com garylee@ecosine.com.tw	up-to-date
Croatian	Boris Bralo	boris.bralo@zg.tel.hr	up-to-date
Czech	Petr Přikryl	prikrylp@skil.cz	up-to-date
Danish	Erik Søe Sørensen	erik@mail.nu	up-to-date
Dutch	Dimitri van Heesch	dimitri@stack.nl	up-to-date
English	Dimitri van Heesch	dimitri@stack.nl	up-to-date
Finnish	Olli Korhonen	Olli.Korhonen@ccc.fi	obsolete
French	Xavier Outhier	xouthier@yahoo.fr	up-to-date
German	Jens Seidel	jensseidel@users.sf.net	up-to-date
Greek	Harry Kalogirou	harkal@rainbow.cs.unipi.gr	1.2.11
Hungarian	Földvári György	foldvari@diatronltd.com	1.2.1
Italian	Alessandro Falappa Ahmed Aldo Faisal	alessandro@falappa.net aaf23@cam.ac.uk	up-to-date
Japanese	Ryunosuke Satoh Kenji Nagamatsu	sun594@hotmail.com naga@joyful.club.ne.jp	1.3.3
JapaneseEn	unknown	unknown	obsolete
Korean	Richard Kim	ryk@dspwiz.com	up-to-date
Norwegian	Lars Erik Jordet	lej@circuitry.no	1.2.2
Polish	Piotr Kaminski Grzegorz Kowal	Piotr.Kaminski@ctm.gdynia.pl g_kowal@poczta.onet.pl	strange
Portuguese	Rui Godinho Lopes	ruiglopes@yahoo.com	1.3.3
Romanian	Alexandru Iosup	aiosup@yahoo.com	1.2.16
Russian	Alexandr Chelpanov	cav@cryptopro.ru	up-to-date
Serbian	Dejan Milosavljevic	dmilos@email.com	1.3.3
Slovak	Stanislav Kudláč	skudlac@pobox.sk	1.2.18
Slovene	Matjaz Ostroversnik	matjaz.ostroversnik@zrs-tk.si	1.2.16
Spanish	Francisco Oltra Thennet	foltra@puc.cl	1.3.3
Swedish	Mikael Hallin	mikaelhallin@yahoo.se	1.3.3
Ukrainian	Olexij Tkatchenko	olexij.tkatchenko@gmx.de	1.2.11

Most people on the list have indicated that they were also busy doing other things, so if you want to help to speed things up please let them (or me) know.

If you want to add support for a language that is not yet listed please read the next section.

Adding a new language to doxygen This short HOWTO explains how to add support for a new language to Doxygen:

Just follow these steps:

1. Tell me for which language you want to add support. If no one else is already working on support for that language, you will be assigned as the maintainer for the language.

2. Create a copy of translator_en.h and name it translator_<your_2_letter_country_code>.h I'll use xx in the rest of this document.
3. Add definition of the symbol for your language into lang_cfg.h:

   ```
   #define LANG_xx
   ```

   Use capital letters for your `xx` (to be consistent). The `lang_cfg.h` defines which language translators will be compiled into doxygen executable. It is a kind of configuration file. If you are sure that you do not need some of the languages, you can remove (comment out) definitions of symbols for the languages, or you can say `#undef` instead of `#define` for them.
4. Edit language.cpp: Add a

   ```
   #ifdef LANG_xx
   #include<translator_xx.h>
   #endif
   ```

   Remember to use the same symbol LANG_xx that you added to `lang_cfg.h`. I.e., the `xx` should be capital letters that identify your language. On the other hand, the `xx` inside your `translator_xx.h` should be lower case.

   Now, in `setTranslator()` add

   ```
   #ifdef LANG_xx
       else if (L_EQUAL("your_language_name"))
       {
         theTranslator = new TranslatorYourLanguage;
       }
   #endif
   ```

   after the `if { ... }`. I.e., it must be placed after the code for creating the English translator at the beginning, and before the `else { ... }` part that creates the translator for the default language (English again).
5. Edit libdoxygen.pro.in and add `translator_xx.h` to the `HEADERS` line.
6. Edit `translator_xx.h`:
   - Rename `TRANSLATOR_EN_H` to `TRANSLATOR_XX_H` twice (i.e. in the `#ifndef` and `#define` preprocessor commands at the beginning of the file).
   - Rename TranslatorEnglish to TranslatorYourLanguage
   - In the member `idLanguage()` change "english" into the name of your language (use lower case characters only). Depending on the language you may also wish to change the member functions latexLanguageSupportCommand(), idLanguageCharset() and others (you will recognize them when you start the work).
   - Edit all the strings that are returned by the member functions that start with tr. Try to match punctuation and capitals! To enter special characters (with accents) you can:
     - Enter them directly if your keyboard supports that and you are using a Latin-1 font. Doxygen will translate the characters to proper LaTeX and leave the HTML and man output for what it is (which is fine, if idLanguageCharset() is set correctly).
     - Use html codes like ä for an a with an umlaut (i.e. ä). See the HTML specification for the codes.
7. Run configure and make again from the root of the distribution, in order to regenerated the Makefiles.
8. Now you can use `OUTPUT_LANGUAGE = your_language_name` in the config file to generate output in your language.
9. Send `translator_xx.h` to me so I can add it to doxygen. Send also your name and e-mail address to be included in the `maintainers.txt` list.

Maintaining a language New versions of doxygen may use new translated sentences. In such situation, the `Translator` class requires implementation of new methods – its interface changes. Of course, the English sentences need to be translated to the other languages. At least, new methods have to be implemented by the language-related translator class; otherwise, doxygen wouldn't even compile. Waiting until all language maintainers have translated the new sentences and sent the results would not be very practical. The following text describes the usage of translator adapters to solve the problem.

The role of Translator Adapters. Whenever the `Translator` class interface changes in the new release, the new class `TranslatorAdapter_x_y_z` is added to the `translator_adapter.h` file (here x, y, and z are numbers that correspond to the current official version of doxygen). All translators that previously derived from the `Translator` class now derive from this adapter class.

The `TranslatorAdapter_x_y_z` class implements the new, required methods. If the new method replaces some similar but obsolete method(s) (e.g. if the number of arguments changed and/or the functionality of the older method was changed or enriched), the `TranslatorAdapter_x_y_z` class may use the obsolete method to get the result which is as close as possible to the older result in the target language. If it is not possible, the result (the default translation) is obtained using the English translator, which is (by definition) always up-to-date.

For example, when the new `trFile()` method with parameters (to determine the capitalization of the first letter and the singular/plural form) was introduced to replace the older method `trFiles()` without arguments, the following code appeared in one of the translator adapter classes:

```
/*! This is the default implementation of the obsolete method
 * used in the documentation of a group before the list of
 * links to documented files.  This is possibly localized.
 */
virtual QCString trFiles()
{ return "Files"; }

/*! This is the localized implementation of newer equivalent
 * using the obsolete method trFiles().
 */
virtual QCString trFile(bool first_capital, bool singular)
{
  if (first_capital && !singular)
    return trFiles();  // possibly localized, obsolete method
  else
    return english.trFile(first_capital, singular);
}
```

The `trFiles()` is not present in the `TranslatorEnglish` class, because it was removed as obsolete. However, it was used until now and its call was replaced by

```
trFile(true, false)
```

in the doxygen source files. Probably, many language translators implemented the obsolete method, so it perfectly makes sense to use the same language dependent result in those cases. The `TranslatorEnglish` does not implement the old method. It derives from the abstract `Translator` class. On the other hand, the old translator for a different language does not implement the new `trFile()` method. Because of that it is derived from another base class – `TranslatorAdapter_x_y_z`. The `TranslatorAdapter_x_y_z` class have to implement the new, required `trFile()` method. However, the translator adapter would not be compiled if the `trFiles()` method was not implemented. This is the reason for implementing the old method in the translator adapter class (using the same code, that was removed from the TranslatorEnglish).

The simplest way would be to pass the arguments to the English translator and to return its result. Instead, the adapter uses the old `trFiles()` in one special case – when the new `trFile(true, false)` is called. This is the mostly used case at the time of introducing the new method – see above. While this

may look too complicated, the technique allows the developers of the core sources to change the Translator interface, while the users may not even notice the change. Of course, when the new `trFile()` is used with different arguments, the English result is returned and it will be noticed by non English users. Here the maintainer of the language translator should implement at least that one particular method.

What says the base class of a language translator? If the language translator class inherits from any adapter class the maintenance is needed. In such case, the language translator is not considered up-to-date. On the other hand, if the language translator derives directly from the abstract class `Translator`, the language translator is up-to-date.

The translator adapter classes are chained so that the older translator adapter class uses the one-step-newer translator adapter as the base class. The newer adapter does less *adapting* work than the older one. The oldest adapter class derives (indirectly) from all of the adapter classes. The name of the adapter class is chosen so that its suffix is derived from the previous official version of doxygen that did not need the adapter. This way, one can say approximately, when the language translator class was last updated – see details below.

The newest translator adapter derives from the abstract `TranslatorAdapterBase` class that derives directly from the abstract `Translator` class. It adds only the private English-translator member for easy implementation of the default translation inside the adapter classes, and it also enforces implementation of one method for noticing the user that the language translation is not up-to-date (because of that some sentences in the generated files may appear in English).

Once the oldest adapter class is not used by any of the language translators, it can be removed from the doxygen project. The maintainers should try to reach the state with the minimal number of translator adapter classes.

To simplify the maintenance of the language translator classes for the supported languages, the `translator.pl` perl script was developed (located in `doxygen/doc` directory). It extracts the important information about obsolete and new methods from the source files for each of the languages. The information is stored in the *translator report* ASCII file (`doxygen/doc/translator_report.txt`).

Looking at the base class of the language translator, the script guesses also the status of the translator – see the last column of the table with languages above. The `translator.pl` is called automatically when the doxygen documentation is generated. You can also run the script manualy whenever you feel that it can help you. Of course, you are not forced to use the results of the script. You can find the same information by looking at the adapter class and its base classes.

How should I update my language translator? Firstly, you should be the language maintainer, or you should let him/her know about the changes. The following text was written for the language maintainers as the primary audience.

There are several approaches to be taken when updating your language. If you are not extremely busy, you should always chose the most radical one. When the update takes much more time than you expected, you can always decide use some suitable translator adapter to finish the changes later and still make your translator working.

The most radical way of updating the language translator is to make your translator class derive directly from the abstract class `Translator` and provide translations for the methods that are required to be implemented – the compiler will tell you if you forgot to implement some of them. If you are in doubt, have a look at the `TranslatorEnglish` class to recognize the purpose of the implemented method. Looking at the previously used adapter class may help you sometimes, but it can also be misleading because the adapter classes do implement also the obsolete methods (see the previous `trFiles()` example).

In other words, the up-to-date language translators do not need the `TranslatorAdapter_x_y_z` classes at all, and you do not need to implement anything else than the methods required by the Translator class (i.e. the pure virtual methods of the `Translator` – they end with `=0;`).

If everything compiles fine, try to run `translator.pl`, and have a look at the translator report (ASCII

file) at the `doxygen/doc` directory. Even if your translator is marked as up-to-date, there still may be some remarks related to your souce code. Namely, the obsolete methods–that are not used at all–may be listed in the section for your language. Simply, remove their code (and run the `translator.pl` again).

If you do not have time to finish all the updates you should still start with *the most radical approach* as described above. You can always change the base class to the translator adapter class that implements all of the not-yet-implemented methods.

If you prefer to update your translator gradually, look at the *translator report* generated by the `translator.pl` script and choose one of the missing method that is implemented by the translator adapter, that is used as your base class. When there is not such a method in your translator adapter base class, you probably can change the translator adapter base to the newer one.

Probably the easiest approach of the gradual update is to look at the translator report to the part where the list of the implemented translator adapters is shown. Then:

- Look how many required methods each adapter implements and guess how many methods you are willing to update (to spend the time with).
- Choose the related oldest translator adapters to be removed (i.e. not used by your translator).
- Change the base class of your translator class to the translator adapter that you want to use.
- Implement the methods that were implemented by the older translator adapters.

Notice: Do not blindly implement all methods that are implemented by your translator adapter base class. The reason is that the adapter classes implement also obsolete methods. Another reason is that some of the methods could become obsolete from some newer adapter on. Focus on the methods listed as *required*.

The really obsolete language translators may lead to too much complicated adapters. Because of that, doxygen developers may decide to derive such translators from the `TranslatorEnglish` class, which is by definition always up-to-date.

When doing so, all the missing methods will be replaced by the English translation. This means that not-implemented methods will always return the English result. Such translators are marked using word `obsolete`. You should read it **really obsolete**. No guess about the last update can be done.

Often, it is possible to construct better result from the obsolete methods. Because of that, the translator adapter classes should be used if possible. On the other hand, implementation of adapters for really obsolete translators brings too much maintenance and run-time overhead.

Index