Documenting the code

This chapter covers two topics:

1. How to put comments in your code such that doxygen incorporates them in the documentation it generates. This is further detailed in the **next section**.
2. Ways to structure the contents of a comment block such that the output looks good, as explained in section **Anatomy of a comment block**.

Table of Contents

Special comment blocks

A special comment block is a C or C++ style comment block with some additional markings, so doxygen knows it is a piece of structured text that needs to end up in the generated documentation. The **next** section presents the various styles supported by doxygen.

For Python, VHDL, Fortran, and Tcl code there are different commenting conventions, which can be found in sections **Comment blocks in Python**, **Comment blocks in VHDL**, **Comment blocks in Fortran**, and **Comment blocks in Tcl** respectively.

Comment blocks for C-like languages (C/C++/C#/Objective-C/PHP/Java)

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

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

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

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

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

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

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

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

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

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

   is also valid.

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

```
///
```

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

or

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

Note that a blank line ends a documentation block in this case.

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

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

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

   or

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

For the brief description there are also several possibilities:

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 description.

   //! 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. If you have multiple detailed descriptions, like in the following example:

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

They will be joined. Note that this is also the case if the descriptions are at different places in the code! In this case the order will depend on the order in which doxygen parses the code.

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.

Putting documentation after members

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

Here are some examples:

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

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

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

or

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

or

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

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

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

or

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

For functions one can use the **@param** command to document the parameters and then use `[in]`, `[out]`, `[in,out]` to document the direction. For inline documentation this is also possible by starting with the direction attribute, e.g.

```
void foo(int v /**< [in] docs for input parameter v. */);
```

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 */
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

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 not allowed inside these comment blocks.

Examples

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);
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

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 behavior 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);
};
```

Click here for the corresponding HTML documentation that is generated by doxygen.

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

Documentation at other places

In the examples in the previous section the comment blocks were always located *in front* of the declaration or definition of a file, class or namespace or *in front* or *after* one of its members. Although this is often comfortable, there may sometimes be reasons to put the documentation somewhere else. For documenting a file this is even required since there is no such thing as "in front of a file".

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

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

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

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

    A more detailed class description.
*/
```

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

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

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

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

Attention

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);
```

Click here for the corresponding HTML documentation that is generated by doxygen.

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.

When you place a comment block in a file with one of the following extensions `.dox`, `.txt`, or `.doc` then doxygen will hide this file from the file list.

If you have a file that doxygen cannot parse but still would like to document it, you can show it as-is using **\verbinclude**, e.g.

```
/*! \file myscript.sh
 *  Look at this nice script:
 *  \verbinclude myscript.sh
 */
```

Make sure that the script is explicitly listed in the **INPUT** or that **FILE_PATTERNS** includes the `.sh` extention and the the script can be found in the path set via **EXAMPLE_PATH**.

Comment blocks in Python

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

```
"""@package docstring
```

```
Documentation for this module.

More details.
"""

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

    More details.
    """
    pass

class PyClass:
    """Documentation for a class.

    More details.
    """

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

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

```

Click here for the corresponding HTML documentation that is generated by doxygen.

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

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

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

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

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

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

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

    ## Documentation for a method.
    #  @param self The object pointer.
    def PyMethod(self):
        pass

    ## A class variable.
    classVar = 0;

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

Click here for the corresponding HTML documentation that is generated by doxygen.

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

Comment blocks in VHDL

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

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

Here is an example VHDL file with doxygen comments:

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

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

--! Mux entity brief description

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

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

```

Click here for the corresponding HTML documentation that is generated by doxygen.

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

Comment blocks in Fortran

When using doxygen for Fortran code you should set **OPTIMIZE_FOR_FORTRAN** to `YES`.

The parser tries to guess if the source code is fixed format Fortran or free format Fortran code. This may not always be correct. If not one should use **EXTENSION_MAPPING** to correct this. By setting `EXTENSION_MAPPING = f=FortranFixed f90=FortranFree` files with extension `f` are interpreted as fixed format Fortran code and files with extension `f90` are interpreted as free format Fortran code.

For Fortran "!>" or "!<" starts a comment and "!!" or "!>" can be used to continue an one line comment into a multi-line comment.

Here is an example of a documented Fortran subroutine:

```
!> Build the restriction matrix for the aggregation
!! method.
!! @param aggr information about the aggregates
!! @todo Handle special case
subroutine intrestbuild(A,aggr,Restrict,A_ghost)
  implicit none
  Type(spmtx), intent(in) :: a !< our fine level matrix
  Type(aggrs), intent(in) :: aggr
  Type(spmtx), intent(out) :: restrict !< Our restriction matrix
  !...
end subroutine
```

As an alternative you can also use comments in fixed format code:

```
C> Function comment
C> another line of comment
      function A(i)
C> input parameter
        integer i
      end function A
```

Comment blocks in Tcl

Doxygen documentation can be included in normal Tcl comments.

To start a new documentation block start a line with `##` (two hashes). All following comment lines and continuation lines will be added to this block. The block ends with a line not starting with a `#` (hash sign).

A brief documentation can be added with `;#<` (semicolon, hash and lower then sign). The brief documentation also ends at a line not starting with a `#` (hash sign).

Inside doxygen comment blocks all normal doxygen markings are supported. The only exceptions are described in the following two paragraphs.

If a doxygen comment block ends with a line containing only `#\code` or `#@code` all code until a line only containing `#\endcode` or `#@endcode` is added to the generated documentation as code block.

If a doxygen comment block ends with a line containing only `#\verbatim` or `#@verbatim` all code until a line only containing `#\endverbatim` or `#@endverbatim` is added verbatim to the generated documentation.

To detect namespaces, classes, functions and variables the following Tcl commands are recognized. Documentation blocks can be put on the lines before the command.

- `namespace eval` .. Namespace
- `proc` .. Function
- `variable` .. Variable
- `common` .. Common variable
- `itcl::class` .. Class
- `itcl::body` .. Class method body definition
- `oo::class create` .. Class
- `oo::define` .. OO Class definition
- `method` .. Class method definitions
- `constructor` .. Class constructor
- `destructor` .. Class destructor
- `public` .. Set protection level
- `protected` .. Set protection level
- `private` .. Set protection level

Following is an example using doxygen style comments:

```
## \file tclexample.tcl
# File documentation.
#\verbatim

# Startup code:\
exec tclsh "$0" "$@"
#\endverbatim
## Documented namespace \c ns .
# The code is inserted here:
#\code
namespace eval ns {
  ## Documented proc \c ns_proc .
  # param[in] arg some argument
  proc ns_proc {arg} {}
  ## Documented var \c ns_var .
  # Some documentation.
  variable ns_var
  ## Documented itcl class \c itcl_class .
  itcl::class itcl_class {
    ## Create object.
    constructor {args} {eval $args}
    ## Destroy object.
    destructor {exit}
    ## Documented itcl method \c itcl_method_x .
    # param[in] arg Argument
    private method itcl_method_x {arg}{}
    ## Documented itcl method \c itcl_method_y .
    # param[in] arg Argument
    protected method itcl_method_y {arg} {}
    ## Documented itcl method \c itcl_method_z .
    # param[in] arg Argument
    public method itcl_method_z {arg} {}
    ## Documented common itcl var \c itcl_Var .
    common itcl_Var
    ## \protectedsection

    variable itcl_var1;#< Documented itcl var \c itcl_var1 .
    variable itcl_var2}
  ## Documented oo class \c oo_class .
  oo::class create oo_class {
    ## Create object.
    # Configure with args
    constructor {args} {eval $args}
    ## Destroy object.
    # Exit.
    destructor {exit}
    ## Documented oo var \c oo_var .
    # Defined inside class
    variable oo_var
    ## \private Documented oo method \c oo_method_x .
    # param[in] arg Argument
    method oo_method_x {arg} {}
    ## \protected Documented oo method \c oo_method_y .
    # param[in] arg Argument
    method oo_method_y {arg} {}
    ## \public Documented oo method \c oo_method_z .
    # param[in] arg Argument
    method oo_method_z {arg} {}
  }
}
#\endcode

itcl::body ::ns::itcl_class::itcl_method_x {argx} {
  puts "$argx OK"
}

oo::define ns::oo_class {
  ## \public Outside defined variable \c oo_var_out .
  # Inside oo_class
  variable oo_var_out
}

## Documented global proc \c glob_proc .
# param[in] arg Argument
proc glob_proc {arg} {puts $arg}

variable glob_var;#< Documented global var \c glob_var\
  with newline
#< and continued line

# end of file
```

Click here for the corresponding HTML documentation that is generated by doxygen.

Anatomy of a comment block

The previous section focused on how to make the comments in your code known to doxygen, it explained the difference between a brief and a detailed description, and the use of structural commands.

In this section we look at the contents of the comment block itself.

Doxygen supports various styles of formatting your comments.

The simplest form is to use plain text. This will appear as-is in the output and is ideal for a short description.

For longer descriptions you often will find the need for some more structure, like a block of verbatim text, a list, or a simple table. For this doxygen supports the Markdown syntax, including parts of the Markdown Extra extension.

Markdown is designed to be very easy to read and write. It's formatting is inspired by plain text mail. Markdown works great for simple, generic formatting, like an introduction page for your project. Doxygen also supports reading of markdown files directly. See **here** for more details regards Markdown support.

For programming language specific formatting doxygen has two forms of additional markup on top of Markdown formatting.

1. Javadoc like markup. See **here** for a complete overview of all commands supported by doxygen.
2. XML markup as specified in the C# standard. See **here** for the XML commands supported by doxygen.

If this is still not enough doxygen also supports a **subset** of the HTML markup language.

Go to the next section or return to the index.