18.3 Documentation Comments

18.3.1 Preliminaries

Documentation comments describe classes, constructors, destructors, methods, members, and functions. The Doxygen documentation system [6] is used to extract the documentation comments and create external documentation in HTML or LaTeX. Although Doxygen supports several documentation formats, we will stick to the Javadoc format as it is widely-accepted and it facilitates visually-pleasing and unobstrusive comments.

Each documentation comment is set inside the comment delimiters `/** ... */`. Within this comment, several keywords are used to flag specific types of information (*e.g.* `@param`, `@see`, and `@return`). We will treat each of these below by way of example.

Documentation comments are placed in front of a declaration or definition. Although Doxygen allows documentation comments to be placed in other places, such as after a declaration, in another location, or in another file, we will stick to the convention that documentation comments are placed directly in front of a declaration or definition.

Note that blank lines are treated as paragraph separators and the resulting documentation will automatically have a new paragraph whenever a blank line is encountered in a documentation comment.

18.3.2 Brief and Detailed Descriptions

For each code item (class, constructor, destructor, method, member, and function) there are two types of descriptions, which together form the documentation: a brief description and detailed description. Both should be provided. Having more than one brief or detailed description is not allowed.

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

As noted above, we use the JavaDoc style so that the brief description is automatically taken from the first line of the comment block and it is terminated by the first dot followed by a space or new line. For example:

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

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 opposite applies: the one before the definition is used and the one before the declaration will be ignored.

In short, brief descriptions before declarations have precedence over brief descriptions before definitions; detailed descriptions before definitions have precedence over detailed descriptions before declarations.

We recommend that you avoid confusion and simply put all documentation comments, i.e. *brief and detailed descriptions, before declarations in the interface (*`.h`*) file.*

Note that to use the JavaDoc style `JAVADOC_AUTOBRIEF` must be set to `YES` in the Doxygen configuration file.

18.3.3 The First Documentation Comment

All source files should begin with a documentation comment that lists the program or class name, version information, and date, as follows.

```
/** @file <filename> <one line to identify the nature of the file>
 *
 * Version information
 *
 * Date
 *
 */
```

18.3.4 Documenting Classes

There should be one documentation comment per class or function. This comment should appear just before the declaration:

```
/**
 * 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 */
   };

   Tenum *enumPtr;  /**< enum pointer.  Details. */
   Tenum  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);
};
```

Doxygen also allows you to put the documentation of members (including global functions) in front of the definition. This way the detailed documentation can be placed in the source file (definition) instead of the header file (declaration). Recall the point we made above about the precedence of definition and declaration regarding brief and detailed descriptions, and the recommendation that you

put all documentation comments in the header (*i.e.* interface) file.

Top-level classes are not indented but their members are. The first line of a documentation comment is not indented but subsequent documentation comment lines each have one space of indentation (to align the asterisks vertically). Members, including constructors and destructors, have three or four spaces for the first documentation comment line (depending on which indentation standard you are using) and five spaces thereafter.

If you need to give information about a class, method, member, or function that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in such an implementation block comment *following* the class statement, not in the class documentation comment.

Documentation comments should not be positioned inside a method or a constructor definition block, because Doxygen associates documentation comments with the first declaration *after* the comment.

Documentation comments should, as a bare minimum, state:

- What the function or method does.
- What arguments it is passed, their types, and their use.
- What arguments it returns, their types, and their use.
- What the return type is, if any, and what it signifies.

18.3.5 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. For example:

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

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.

18.3.6 Documenting Global Code Items

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.* This causes a problem because you can't put a document comment 'in front' of a file. Doxygen allows code items to be documented by putting the document comment somewhere else but you must then identify the code item being documented with a *structural command*.

To document a global code item, such as a C function, you must document the file in which they are defined by putting a document comment with file structural command

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

in that file. Usually this will be a header file. Here is an example of a C header named `structcmd.h`.

```
/** @file structcmd.h  A documented header file ...
 * These are the functions ...
 */


/** Opens a file descriptor.
 * @param pathname The name of the descriptor.
 * @param flags Opening flags.
 */
int open(const char *,int);


/** Closes the file descriptor .
 * @param fd The descriptor to close.
 */
int close(int);


/** 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.
 */
size_t write(int,const char *, size_t);
```

19 Programming Style

19.1 Declarations

19.1.1 Number Per Line

One declaration per line is recommended since it encourages commenting:

```
   int level; // indentation level
   int size;  // size of table
```

is preferable to:

```
   int level, size;
```

Do not put different types on the same line:

```
int foo, fooarray[]; //WRONG!
```

19.1.2 Initialization

Initialize local variables where they are declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

19.2 Placement

Put declarations only at the beginning of blocks. A block is any code surrounded by curly braces { and }. Don't wait to declare variables until their first use. Ideally, declare all variables at the beginning of the method or function block.

```
void myMethod() {
   int int1 = 0; // beginning of method block

   if (condition) {
      int int2 = 0; // beginning of "if" block
      ...
   }
}
```

19.2.1 Class Declarations

The following formatting rules should be followed:

- No space between a method name and the parenthesis (starting its parameter list.
- The open brace { appears at the end of the same line as the declaration statement.
- The closing brace } starts a line by itself indented to match its corresponding opening statement.

```
class Sample {
   ...
}
```

- Methods are separated by a blank line.

19.3 Statements

19.3.1 Simple Statements

Each line should contain at most one statement. For example:

```
argv++;          // Correct
argc++;          // Correct
argv++; argc--; // AVOID!
```

19.3.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces { statements }. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

```
if (condition) {
   a = b;
}
else {
   a = c;
}
```

19.3.3 `return` Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. For example:

```
return;

return myDisk.size();

return TRUE;
```

19.3.4 `if`, `if-else`, `if else-if else` Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
   statements;
}

if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else {
   statements;
}
```

Always use braces `{ }`, with `if` statements. Don't use

```
if (condition) //AVOID!
   statement;
```

19.3.5 `for` Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
   statements;
}
```

19.3.6 `while` Statements

A `while` statement should have the following form:

```
while (condition) {
   statements;
}
```

19.3.7 `do-while` Statements

A `do-while` statement should have the following form:

```
do {
   statements;
} while (condition);
```

19.3.8 `switch` Statements

A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
   statements;
   /* falls through */
case DEF:
   statements;
   break;
case XYZ:
   statements;
   break;
default:
   statements;
   break;
}
```

Every time a case falls through (*i.e.* when it doesn't include a `break` statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

19.4 Naming Conventions

19.4.1 C vs. C++

Naming conventions make programs more understandable by making them easier to read. Since iCub software uses both the C language and the C++ language, sometimes using the imperative programming and object-oriented programming paradigms separately, sometimes using them together, we will adopt two different naming conventions, one for C and the other for C++. The naming conventions for C++ are derived from the JavaDoc standards [4].

19.4.2 C++ Language Conventions

The following are the naming conventions for identifiers when using C++ and the object-oriented paradigm.

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized	`class ImageDisplay` `class MotorController`
Methods	Method names should be verbs, in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int grabImage()` `int setVelocity()`

Variables	variable names should be in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int i;`
		`float f;`
		`double pixelValue;`

Constants	The names of variables declared as constants should be all uppercase with words separated by underscores _	`const int WIDTH = 4;`
Type Names	Typedef names should use the same naming policy as that used for class names	`typedef uint16 ModuleType`

Enum Names	Enum names should use the same naming policy as that used for class names.
	Enum labels should should be all uppercase with words separated by underscores _
		`enum PinState {`
		`PIN_OFF,`
		`PIN_ON`
		`};`

19.4.3 C Language Conventions

The following are the naming conventions for identifies when using C and the imperative programming paradigm.

Identifier Type	Rules for Naming	Examples
Functions	Function names should be all lowercase with words separated by underscores _	`int display_image()` `void set_motor_control()`

Variables	variable names should be all lowercase with words separated by underscores _
	of each internal word capitalized
		`int i;`
		`float f;`
		`double pixel_value;`

Constants	Constants should be all uppercase with words separated by underscores _	`#define WIDTH 4`
`#define` and Macros	`#define` and macro names should all uppercase with words separated by underscores _	`#define SUB(a,b) ( (a) - (b) )`

19.5 And Finally: Where To Put The Opening Brace {

There are two main conventions on where to put the opening brace of a block. In this document, we have adopted the JavaDoc convention and put the brace on the same line as the statement preceding the block. For example:

```
class Sample {
   ...
}


while (condition) {
   statements;
}
```

The second convention is to place the brace on the line below the statement preceding the block and it indent it to the same level. For example:

```
class Sample
{
   ...
}


while (condition)
{
   statements;
}
```

If you really hate the JavaDoc format, use the second format, but be consistent and stick to it throughout your code.

20 Programming Practice

20.1 C++ Language Conventions

20.1.1 Access to Data Members

Don't make any class data member public without good reason.

One example of appropriate public data member is the case where the class is essentially a data structure, with no behaviour. In other words, if you would have used a struct instead of a class, then it's appropriate to make the class's data members public.

20.2 C Language Conventions

Use the Standard C syntax for function definitions:

```
void example_function (int an_integer, long a_long, short a_short)
...
```

If the arguments don't fit on one line, split the line according to the rules in Section 20.2:

```
void example_function (int an_integer, long a_long, short a_short,
                       float a_float, double a_double)
...
```

20.3 General Issues

20.3.1 Conditional Compilation

Avoid the use of conditional compilation. If your code deals with different configuration options, use a conventional `if-else` construct. If the code associated with either clause is long, put it in a separate function. For example, please write:

```
if (HAS_FOO) {
   ...
}
else {
   ...
}
```

instead of:

```
#ifdef HAS_FOO
   ...
#else
   ...
#endif
```

20.3.2 Writing Robust Programs

Avoid arbitrary limits on the size or length of any data structure, including arrays, by allocating all data structures dynamically. Use `malloc` or `new` to create data-structures of the appropriate size. Remember to avoid memory leakage by always using `free` and `delete` to deallocate dynamically-created data-structures.

Check every call to `malloc` or `new` to see if it returned `NULL`.

You must expect `free` to alter the contents of the block that was freed. Never access a data structure after it has been freed.

If `malloc` fails in a non-interactive program, make that a fatal error. In an interactive program, it is better to abort the current command and return to the command reader loop.

When static storage is to be written during program execution, use explicit C or C++ code to initialize it. Reserve C initialize declarations for data that will not be changed. Consider the following two examples.

```
static int two = 2; // two will never alter its value
...
static int flag;
flag = TRUE;        // might also be FALSE
```

20.3.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

20.3.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read.

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

```
if (c++ = d++) { // AVOID!
   ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
   ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler.

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

20.3.5 Parentheses

Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others — you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)     // AVOID!

if ((a == b) && (c == d)) // USE
```

20.3.6 Standards for Graphical Interfaces

When you write a program that provides a graphical user interface (GUI), you should use a cross-platform library. At the very least, it must possible to compile your GUI code for both a Window environment and a Linux environment. The FLTK GUI library [3] satisfies this requirement.

20.3.7 Error Messages

Error messages should look like this:

```
function_name: error message
```

20.3.8 License Messages

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
<Program name and version>

Copyright (C) <year> <name of author>, <author institute>
RobotCub Consortium, European Commission FP6 Project IST-004370
email:   <firstname.secondname>@robotcub.org
website: www.robotcub.org

This program comes with ABSOLUTELY NO WARRANTY.
Permission is granted to copy, distribute, and/or modify this program
under the terms of the GNU General Public License, version 2
or any later version published by the Free Software Foundation;
see http://www.robotcub.org/icub/license/gpl.txt
```