

# 1 Introduction

## 1.1 Terms and Definitions

A quick reference with explanations of terms used in this document is provided in Table 1. Note that the definitions are often C programming language specific and might not be true in other contexts.

Term	Definition
allocated storage duration	Object with dynamic storage duration, the storage is created and destroyed upon request.
automatic storage duration	Object with storage duration inside a block only, from the point of declaration to the end of the block.
block	A range of statements enclosed between a pair of braces.
declaration	Statement that introduces an identifier and its type.
identifier	A name given to an entity such as variables, functions, structures, et cetera.
object	A location in memory whose content represent a value.
statement	Instruction to a computer, in C typically a single line expression followed by a semicolon.
static storage duration	Storage duration from start to end of program execution.
storage duration	Determines an objects lifetime. There are three storage durations; allocated, auto, and static.
storage class specifier	Specifies storage duration and linkage of objects. There are five specifiers; auto, extern, register, static, and <code>_Thread_local</code> .
type	Variable classification such as; int, char, double, et cetera, that determines storage size and how the bit pattern stored shall be interpreted.
type qualifier	Adds attributes to types at the point of declaration. There are four type qualifiers; const, restrict, volatile, and <code>_Atomic</code> .

Table 1: Term and definition quick reference

## 2 Rules

### 2.1 Initialize Automatic Variables

Variables with automatic storage duration shall be initialized before use.

#### 2.1.1 Reasoning

The value of an automatic variable that is not initialized is undefined. Explicit initialization shall hence be done before using the variable, either when declaring the variable or just before the variable is used for the first time.

#### Non-Compliant

```
int f(void)
{
    uint32_t i;
    i++;
    return i;
}
```

#### Compliant

```
int foo(void)
{
    uint32_t i = 0;
    i++;
    return i;
}
```

## 2.2 Initialize Constant Variables When Declared

Declaration of variables with const type qualifier shall include initialization.

## 2.3 Strict Enumeration Initialization

Strategy used for initialization of the members in an enum type shall be one of the following: not specifying any values, specifying all values, specifying only the first value.

### 2.3.1 Reasoning

Minimizes the risk that a pair of members is assigned the same value by mistake.

#### Non-Compliant

```
enum nonCompliantEnum
{
    NC_E1 = 1,
    NC_E2,
    NC_E3 = 2
};
```

#### Compliant

```
enum compliantEnum_1
{
    C1_E1,
    C1_E2,
    C1_E3
};

enum compliantEnum_2
{
    C2_E1 = 1,
    C2_E2,
    C2_E3
};

enum compliantEnum_3
{
    C3_E1 = 1,
    C3_E2 = 3,
    C3_E3 = 4
};
```

## 2.4 No Usage of Restrict

Do not use the restrict type qualifier.

### 2.4.1 Reasoning

The keyword `restrict` is type qualifier that can be added in a object pointer declaration. It provides a hint to the compiler that only this pointer will be used access the object. This will in some situations make it possible for the compiler to generate a more optimized result. The behavior of the code will be undefined if this guarantee is not meet. Using `restrict` burdens the design of the code to guarantee that the memory areas do not overlap and adds a risk, the `restrict` type qualifier shall hence not be used.

### 2.4.2 Examples

Non-Compliant

```
void g(int *restrict a,
      int *restrict b,
      int *restrict x)
{
    *a += *x;
    *b += *x;
}
```

Compliant

```
void f(int *a,
      int *b,
      int *x)
{
    *a += *x;
    *b += *x;
}
```

### Rule 1

The declaration of a function that does not take any parameters shall use the `void` type parameter.

#### Examples

Non-Compliant

```
void f();
```

Compliant

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

### Reasoning

A C function declaration with an empty parameter list is not the same as that the function has no parameters, it is an obsolete way to declare a function without needing to explicitly specify the number, and types of parameters. Using `void` states explicitly that a function does not takes any parameters, making it possible for the compiler to check for conflicts in the function usage.

### Rule 2

Use `enum` to define related constants.

## Examples

### Non-Compliant

```
#define UP      0
#define RIGHT   1
#define DOWN    2
#define LEFT    3
```

### Compliant

```
typedef enum eDirection
{
    UP,
    RIGHT,
    DOWN,
    LEFT
} direction;
```

## Reasoning

Defining related constants as enum type, as opposed to a series of preprocessor defines, comes with various benefits. Makes it possible to have automated error checks for when the constant is used in the wrong context. Simplifies debugging due to that there will be a symbol for each enum constant in the debugger symbol table.

## Rule 3

Ensure that pointer cast preserves the type qualifiers of the type addressed by the pointer.

## Examples

TODO fix the other Example

### Non-Compliant

```
void f(char *data_p);

int main(void)
{
    char m[] = "Foo";
    const char *mp = m;
    f((char*) mp);

    return 0;
}
```

### Compliant

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

## Reasoning

Casting away const and volatile memory area qualifications is not illegal but adds some risks. The compiler will not be able to check and detect erroneous handling of the memory area. The compiler might also perform unintended optimization.

## Rule 4

The intentional null statement i.e. a statement that have no effect shall be annotated by an empty macro named DO\_NOTHING. This macro shall be defined in a single place, suitably in a project specific header file.

## Examples

### Non-Compliant

```
void temperatureControlTask(void)
{
    int temperature = 0;

    temperature = getTemperature();
    if (temperature < TEMP_MIN)
    {
        turnOnHeater();
    }
    else if (temperature > TEMP_MAX)
    {
        turnOffHeater();
    }
    else
    {
    }
}
```

### Compliant

```
/*
 * For documenting intentional null
 * statements that do nothing.
 */
#define DO_NOTHING

void temperatureControlTask(void)
{
    int temperature = 0;

    temperature = getTemperature();

    if (temperature < TEMP_MIN)
    {
        turnOnHeater();
    }
    else if (temperature > TEMP_MAX)
    {
        turnOffHeater();
    }
    else
    {
        DO_NOTHING;
    }
}
```

## Reasoning

Explicitly stating that there is nothing to do makes it clear that the statement have not been left blank by mistake. Having a standardized way to indicate this with a macro improves readability, since every occurrence of this type will look the same throughout all project source files.

## Rule 5

There shall be no unused elements in the code. Meaning that unused functions, variables, constants, parameters, typedefs, tags, labels, macros, etcetera shall be removed.

## Examples

### Non-Compliant

```
int squareArea(int width, int height)
{
    return width * width;
}
```

### Compliant

```
int squareArea(int width)
{
    return width * width;
}
```

## Reasoning

Cluttering the code base with unused elements makes it harder to read and understand the code which impairs maintainability.

# Appendix A Doxygen

## Introduction

Doxygen is a tool that generates documentation from annotated source files. A software module can be documented directly when developing the code for the module. This workflow makes it more likely that the documentation is kept consistent with the source code.

The documentation can be generated into various formats depending on the needs of the project process. Examples of supported formats are PDF, HTML, RTF, and  $\text{\LaTeX}$ .

Doxygen is commonly used to document modules and functions intended usage in a textual format. But is also possible to generate various visual representations of the elements in the form of graphs and diagrams. These can be used learn the structure of a project or to verify that the implementation is consistent with the design.

The annotations added to the source code are mostly straightforward and human readable. It is possible for developers to digest the documentation directly in the raw source code without need to run the generation process.

## Source Code Annotation Rules

The Doxygen specific annotations are created by incorporating special kind of comment blocks into the source files. These comment blocks can be written in several different ways and still be accepted by Doxygen. This section specifies a set of rules for how to write the comment blocks. The purpose of the rules is; to have consistency, not miss out on required documentation, trouble free coexistence with other tools.

## Doxygen Rule 1

Mark Doxygen comment blocks as a C-style comment block, with the difference that there shall be two initial asterisks.

## Examples

Non-Compliant

```
/*!  
 *  
 */
```

Compliant

```
/**  
 *  
 */
```

Non-Compliant

```
///  
///  
///  
///
```

Non-Compliant

```
///  
///  
///  
///
```

## Reasoning

Doxygen supports multiple different formats for the comments block that is the base for the documentation generation. A single format shall be used in a project to improve source code readability. What format to use is somewhat arbitrary.

The format specified by this rule is chosen due to that it closely resembles ordinary C comment blocks. Many editors will understand that this is a Doxygen comment block and highlight the block accordingly, and it is also understood and formatted correctly by many automatic source code formatting tools.

## Doxygen Rule 2

Doxygen annotation commands shall start with a backslash.

## Examples

Non-Compliant

```
/**  
 * @brief ...  
 */
```

Compliant

```
/**  
 * \brief ...  
 */
```

## Reasoning

Doxygen accepts two different styles for how to write the annotation commands. A command can start with either a backslash or the at-symbol. The backslash is chosen to be used because this symbol is part of the basic character set. It is desirable to have the entire source code base written using only characters from the basic character set due that it minimizes the risk of problems with portability and usage of various tools.

## Doxygen Rule 3

A source code file with Doxygen annotations shall hold an initial Doxygen comment block where the first command shall be the `\file` command followed by the name of the file.

### Examples

The examples assumes that name of the file where the comment block resides is `main.c`.

Non-Compliant

```
/*  
 * file name: main.c  
 */
```

Compliant

```
/**  
 * \file main.c  
 */
```

### Reasoning

Doxygen will, assuming default settings, only run the documentation process of global objects in a file if the file itself is marked as to be documented. This means that more or less all source files constructed in a project should include the `\file` command in a Doxygen comment placed near the top of the file.

## Doxygen Rule 4

A doxygen comment block used for documenting a function shall include an initial section annotated by the `\brief` command. The text for this command shall be a short one-line description capturing the core functionality.

### Examples

Non-Compliant

```
/**  
 * Get voltage level.  
 *  
 * \param Void.  
 *  
 * \return Voltage level.  
 */  
int getVoltage(void);
```

Compliant

```
/**  
 * \brief Get voltage level.  
 *  
 * \param Void.  
 *  
 * \return The voltage level.  
 */  
int getVoltage(void);
```

### Reasoning

Most functions worthy of a Doxygen comment block deserve at least one line of documentation describing functionality and purpose. The `\brief` command is meant to be used for this type of concise documentation. More detailed description can optionally be inserted after the empty line TODO add reference to doxygen rule 003.



## Doxygen Rule 5

A Doxygen comment block used for documenting a function can include a section with more detailed description of the functionality. This section shall be annotated with the `\details` command. The `\details` section shall follow right below the `\brief` section.

### Examples

#### Non-Compliant

```
/**
 * \brief Get voltage level.
 *
 * Returns battery voltage level as an
 * integer value in mV. For example
 * 3.3 V will be returned as 3300.
 *
 * \param Void.
 *
 * \return The voltage level.
 */
int getVoltage(void);
```

#### Compliant

```
/**
 * \brief Get voltage level.
 *
 * \details Returns battery voltage
 *          level as an integer value
 *          in mV. For example 3.3 V
 *          will be returned as 3300.
 *
 * \param Void.
 *
 * \return The voltage level.
 */
int getVoltage(void);
```

### Reasoning

The `\brief` section shall be short so there is sometimes a need to have additional function documentation in a followup section, this is where the `\details` section comes into play.

The `\details` annotation is strictly not needed because Doxygen will implicitly use a section following the `\brief` section as the detailed function documentation. But explicitly stating that this is the detailed documentation makes it more clear what is placed in this section.

## Doxygen Rule 6

Function Doxygen comment blocks shall use the `\param` command for each function parameter and also include the direction attribute as well as an description.

## Examples

### Non-Compliant

```
/**
 * \brief      Copies RAM data.
 *
 * \details    Copies a given number
 *             of bytes from a source
 *             memory area to a
 *             destination memory
 *             area, where areas may
 *             not overlap.
 *
 *             Dest is where a number
 *             of data bytes will be
 *             copied from src.
 *
 * \return     Void.
 */
void memcpy(void *dest,
            const void *src,
            size_t n);
```

### Compliant

```
/**
 * \brief      Copies RAM data.
 *
 * \details    Copies a given number
 *             of bytes from a source
 *             memory area to a
 *             destination memory
 *             area, where areas may
 *             not overlap.
 *
 * \param[out] dest Destination area.
 * \param[in]  src  Source area.
 * \param[in]  n    Number of bytes.
 *
 * \return     Void.
 */
void memcpy(void *dest,
            const void *src,
            size_t n);
```

## Reasoning

The `\param` command is useful due that it ensures that each function parameter will have documentation, can be especially helpful when using an function written by other developers.

The direction attribute makes it clear if a pointer type parameter will be written to by the function or not.

## Doxygen Rule 7

Function Doxygen comment blocks shall use the `\return` command to document what the function returns. Functions that do not return a value shall be documented with the description "Void".

## Examples

### Non-Compliant

```
/**
 * \brief      Determines the largest
 *             of two float numbers.
 *
 * \details    Returns the largest of
 *             the two float numbers.
 *
 * \param[in]  x First float number.
 * \param[in]  y Second float number.
 */
float fmaxf(float x, float y);
```

### Compliant

```
/**
 * \brief      Determines the largest
 *             of two float numbers.
 *
 * \param[in]  x First float number.
 * \param[in]  y Second float number.
 *
 * \return     The largest of the two
 *             float numbers.
 */
float fmaxf(float x, float y);
```

TODO add example that returns void.

## Reasoning

The Doxygen `\return` command is useful due that it ensures that the value returned by a function will have documentation, can be especially helpful when using an function written by other developers.

Using the description "Void." for functions that do not return anything is mostly a convention to explicitly indicate that the return value have not been missed out in the documentation.

## Doxygen Rule 8

Doxygen annotation commands paragraphs shall be separated horizontally by a blank line. With the exception of the situation that the commands is of the same type, for example multiple `\param` commands do not require any spacing.

## Examples

Non-Compliant

```
/**
 * \brief      Calculates difference
 *             between two times.
 * \details    Function returns the
 *             difference between
 *             time1 - time2 as
 *             seconds.
 * \param[in]  time2 A point in time.
 * \param[in]  time1 A point in time.
 * \return     Time difference.
 */
double difftime(time_t time2,
                time_t time1);
```

Compliant

```
/**
 * \brief      Calculates difference
 *             between two times.
 *
 * \details    Function returns the
 *             difference between
 *             time1 - time2 as
 *             seconds.
 *
 * \param[in]  time2 A point in time.
 * \param[in]  time1 A point in time.
 *
 * \return     Time difference.
 */
double difftime(time_t time2,
                time_t time1);
```

## Reasoning

This rule is mainly intended for readability reasons, to avoid having a wall of documentation text, that can be hard on the eyes.