**OSKAR Station Simulator:**

**Coding conventions**

This document describes the coding conventions used for the OSKAR station simulator. The following subjects are considered.

* Naming conventions
* Comment styles
* The use of Doxygen

# Naming conventions

The following sections define t he naming conventions used in the OSKAR station simulator code. Note that variables with global scope are prefixed with ‘OS’ (Oskar Simulator) to provide some sort of namespace protection

## File Names

Source code files take the ‘os’ prefix followed by upper camel case a compound name. Each source code file has a corresponding header file of the same name.

Header files: osFileName.h
Source files: osFilename.c

## Variables

### Global constants

Globally defined constants are in full uppercase notation. An ‘OS’ prefix is followed by a underscore separated compound variable name.

#define OS\_VARIABLE\_NAME

const type OS\_VARIABLE\_NAME

### Global variables

Global variables are constructed from a lowercase ‘os’ prefix followed by a lower camel case compound variable name.

type os\_variableName

### Local variables

Variables with local scope do not require an OS prefix and use the lower camel case name convention.

type variableName

## Function names

Function names are prefixed with a lowercase os followed by an upper camel case compound function name.

### Declaration

All function declarations reside in header files.

Function declaration can be prefixed with an optional descriptive comment.

In the function declaration, each function argument appears on a separate line followed by an optional comment (the main comment is in the form of a Doxygen comment and resides in the source file.

// optional desciption

returnType osFunctionName(

 type arg1, // optional comment

 type arg2 // optional comment

);

### Definintion

implementation can be from any of the following templates

returnType osFunctionName(

 type arg1, // optional comment.

 type arg2

 )

{

}

***(shorter functions can take the form)***

returnType osFunctionName(type arg1, type arg2)

{

}

# Class names (frontend)

class OS\_ClassName

{

 public:

};

# Comment style

## Header files

Name osFileName.h and take the following form

#ifndef HEADER\_FILE\_NAME\_
#define HEADER\_FILE\_NAME\_

<LICENCE>

/\*\*

 \* @file osFilename.h

 \*

 \* @brief description of what the file is for

 \*

 \* last modified: ... eclipse macro ...

 \*/

... includes ...

#include “common.h”

#include <string.h>

#include <osErrorCodes.h>

... function prototypes etc ...

#endif /\* HEADER\_FILE\_NAME\_ \*/

## Source files

Name osFileName.c(pp) and take the following form.

<LICENCE>

#include “osHeaderFile.h”

... function implementations ...

## Doxygen

Doxygen reads both source code and user special comments to generate documentation. Comments need to be in the form of special documentation blocks which are simply C or C++ style comment blocks with some additional markings.

For each code item there are two (or in some cases three) types of description which together form the documentation

* A brief.
* A detailed description
* (and for methods and functions the in-body description)

As the name suggests, 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 HTML output produced by doxgyen brief descriptions are also used to provide tooltips at the place an item is referenced.

### Detailed description

To mark a comment block as a detailed description, use the following notation.

/\*\*
 \* ... comment text ...
 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
 \* ... comment text ...
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

One line detailed descriptions can also be used in for example.

 /// ... comment text ...

/\*\* ... comment text ... \*/

### Brief description

For the brief description use the @brief command with one of the following above comment blocks. This command ends at the end of a paragraph, with the detailed description following after an empty line.

/\*\*
 \* @brief Brief description ...
 \* Brief description continued.
 \*
 \* Detailed description starts here.
 \*
 \*/

### 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 of these members inside the compound, its sometimes desired to place a documentation block after the member instead of before. For this purpose you have to put an additional “<” marker in the comment block.

int var1; ///< Detailed description after the member

 ///< (*note a detailed description has to be at least*

///< *two lines of comment)*

int var2; ///< Brief description after the member

### Special commands and documentation in other places

Normally the documentation block is located in front of the declaration or definition of a file, class or namespace or in front or after one of its members.

Doxygen however allows you to put your documentation blocks almost anywhere (although should be avoided unless necessary).

In the case that the documentation block is not directly before (or after) an item, you need to put a structural command inside the documentation block.

A structural command starts with an “@” followed by a command name and one or more parameters.

@struct

@union

@enum

@fn (document a function)

@var (document a variable or typedef or enum value)

@def (document a #define)

@typedef

@file

@namespace

### Comment example

Header file

#ifdef COMMENT\_EXAMPLE\_\_

#define COMMENT\_EXAMPLE\_\_

#include <stdio.h>

/\*\*

 \* @brief brief one line description of the function

 \*/

int function(float arg1, int arg2);

#endif /\* COMMENT\_EXAMPLE\_\_ \*/

Source file

#include “commentExample.h”

/\*\*

 \* Detailed description/ extra notes (can put

 \* lists/html/equation etc. here.

 \*

 \* @var arg1 description of arg1

 \* @var

 \* @ref osFunctionName2 you might like to see this as well

 \*/

int function(float arg1, int arg2)

{

 int value;

 .. some operations ..

return value;

}

## Other comments