µOS++
A POSIX inspired open source framework, written in C++.

"Perfekt ist nicht gut genug"

News

- µOS++ v7.0.0 released
- µOS++ v6.3.17 released
- µOS++ v6.3.16 released
- First µOS++ scriptable build

Home

µOS++

- Overview
- Releases

APIs

- Overview
- RTOS API

xPacks

- Overview

Documentation

- User's **manual**
  - Getting started
  - Basic concepts
  - Features
  - Threads
  - Thread event flags
  - Semaphores
  - Event flags
  - Mutexes
  - Condition variables
  - Message queues
  - Memory pools
  - Software timers
  - Clocks
- µOS++ **reference**

Developer

- Overview
- Change log
- C++ coding style
- Naming conventions
- Links & references

Support

- Overview
- Known issues
- FAQ
- Forum

- Report µOS++ issues

Latest Articles

- CMSIS++ RTOS: fully functional reference implementation
- CMSIS++: a proposal for a future CMSIS, written in C++

Project

- About
- History
- License (MIT)

This site uses the GitHub Wiki-like theme by Liviu Ionescu.

Doxygen style guide

Last modified on Sat Jul 22 11:14:10 2023 UTC. Improve this page

Doxygen has many, many features, and accepts various syntaxes for its commands.

In the previous versions the recommended syntax was the traditional C++ one, with `//` for comments, and `\` for escape characters and commands.

However, experience proved that back slashes and triple slashes are more difficult to read; given these subjective reasons, the syntax moved to the original Java documentation style, considered to have a better readability.

`/**` Comments instead of `///`

Use C style blocks of `/** */` instead of C++ `///` lines.

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

Note: Uninterrupted sequences of lines refer to the same object. To be sure the commands are not grouped together when this is not required, use separate comment blocks.

`@` Commands instead of `\`

Use `@` to prefix commands, instead of `\`, which is considered too visually hurtful.

Explicit `@brief`

Explicitly use the `@brief` command before the brief description. End the text with a dot. For visibility reasons, add an empty line inside the same comment block.

```
/**
 * @brief Base for all architecture implementation classes.
 */
```

Explicit `@details`

Explicitly use the `@details` command, on a separate line, before the main documentation text.

For class member functions, the details should be placed before the implementation; although allowed by Doxygen, avoid placing details both at the declaration and definitions, since it is difficult to keep them consistent (see below).

```
/**
 * @details
 * The @ref trace_streambuf_base class implements an @ref ostream class
 * to be used by the @ref trace class.
 */
```

Use `@brief` with declarations and `@details` with definitions

Contrary to Java, the C++ sources are usually split between a .h file with declarations and a .cpp file with the method/functions definitions.

Always add the `@brief`, `@param` and `@return`/`@retval` commands in the header file before the member declaration, and the detailed part of the documentation before the member definition (be it in .h for inline definitions or in .cpp for regular definitions).

Back apostrophes for references to code

Always use `something` instead of `@c` something, or `@p` something, or `<code>something`</code>.

Underscores for italics

Use underscores to mark *italics* texts.

Double asterisks for bold

Use asterisks to mark **bold** texts.

Use `@code` for sequences of source lines

When including lines of code, surround them by `@code`, `@endcode` and add the language. For visibility reasons, add empty lines inside the comment block.

```
/**
 * @code{.cpp}
 * set_class_name("os::infra::test_suite");
 * @endcode
 */
```

Use `@verbatim` for other pre-formatted lines

When including other lines, like shell commands, surround them by `@verbatim`, `@endverbatim`:

```
/**
 * @verbatim
 * /bin/bash micro-os-plus-se.git/scripts/runTests.sh
 * @endverbatim
 */
```

Lists

Use `*` to enter bulleted lists, and `#` to enter numbered lists.

For multiple levels use additional indentations.

```
* first level 1
  * first level 2
  * second level 2
* second level 1
```

Tables

Use the below syntax to enter tables. Columns can be left/right aligned.

```
| Right | Center | Left  |
| ----: | :----: | :---- |
| 10    | 10 | 10    |
| 1000  | 1000   | 1000  |
```

External links

Links to other pages can be expressed with the following markup:

```
[The link text](http://example.net/)
```

Use `@tparam` for template parameters

Use `@tparam` for template parameters. Start the explanation with upper case and end it with dot.

```
/**
 * @tparam T  Type of the implementation class.
 */
```

Use `@param` for function parameters

Use `@param [in]` for usual input parameters, and occasionally `[out]` for output parameters. Start the explanation with upper case and end the line with a dot.

To make the explanation more readable, use tabs to right align the content.

```
/**
 * @param [in]  c  An additional character to consume.
 */
```

If the function has no parameters, use a custom paragraph containing **None.**, indented with a tab and terminated with a dot.

```
/**
 * @par Parameters
 *    None.
 */
```

Use `@return` or `@retval` for the returned result

If the function returns discrete values, enumerate them with `@retval` and terminate the list with an empty line.

```
/**
 * @retval -1 Error
 * @retval 0 OK
 */
```

If the function returns a scalar value, use a `@return` and explain what the value represents.

```
/**
 * @return The number of bytes actually written.
 */
```

If the function has no return value, use a custom paragraph containing **Nothing.**, indented with a tab and terminated with a dot.

```
/**
 * @par Returns
 *    Nothing.
 */
```

Use `@headerfile` to define the header full path

For each class, structure, enum or other object definition, use `@headerfile` to specify the full header path.

```
/**
 * @headerfile CoreInterruptNumbers.h "hal/architecture/arm/cortexm/include/CoreInterruptNumbers.h"
 * @brief ARM Cortex-M architecture interrupt numbers base
 *
 * @details
 * Interrupt numbers defined by the Cortex-M0 light architecture.
 */
class cortex_m0_interrupt_number ...
```

The first name should be present in the filesystem, so it might need some prefixing. The second name is passed to the output.

The comment block should be continuous to the object comments, otherwise the header file definition is not attached to the object.

Use of `@name` to define custom member grouping

For a better look, it is recommended to group class definitions based on their logic, instead of the default class visibility grouping.

Groups have names and descriptions (be sure the comments are continuous up to the `@{` opening brace).

It is recommended to repeat the name in the closing `@}` brace, as seen below.

For readability, use an empty line after opening braces and before closing braces.

Note: In the current version, `@nosubgrouping` is not working as expected, so it is to be avoided.

Note: Groups do not nest.

```
/**
 * @name Standard template types
 *
 * These types permit a standardised way of
 * referring to names of (or names dependent on) the template
 * parameters, which are specific to the implementation. Except
 * when referring to the template, (in which case the templates
 * parameters are required), use these types everywhere
 * else instead of usual types.
 * @{
 */

typedef T implementation_t;

/**
 * @}
 */

/**
 * @name Constructors/destructor
 * @{
 */

/**
 * @brief  Base constructor.
 * @param [in] implementation Reference to the implementation class.
 */
trace_streambuf_base(implementation_t& implementation);

/**
 * @brief  Base destructor.
 */
virtual
~trace_streambuf_base();

/**
 * @}
 */
```

Template sample

```
/**
 * @headerfile trace.h "portable/diagnostics/include/trace.h"
 * @ingroup diag
 *
 * @brief Trace light base class.
 * @tparam T  Type of the implementation class.
 *
 * @details
 * This class provides no functionality, it is used only as a
 * light alternative to trace_ostream_base.
 */
template<class T>
  class trace_light_base
  {
    ...
  };
```

Grouping

`@ingroup`

To include a definition in one or more modules, use `@ingroup`.

```
/**
 * @brief Stack size type.
 * @ingroup core_thread
 */
```

If multiple definition are from the same group, they can be grouped with `@{` … `@}`:

```
/**
 * @ingroup core_thread
 * @{
 */
    <- mandatory different blocks
/**
 * @brief Stack element type.
 */
typedef hal::arch::stack_element_t element_t;

/**
 * @brief Stack size type.
 */
typedef hal::arch::stack_size_t size_t;

/**
 * @}
 */
```

Please note the mandatory empty lines after `@{` and before `@}`.

Also please note that when using together with `@name`, `@ingroup` with must be inside `@name`.

`@name`

This command allows to define the name of the section within a page. Usually it is used with `@{` … `@}`:

```
/**
 * @name Constructors & Destructor
 * @{
 */
    <- mandatory different blocks
/**
 * @brief Constructor.
 */
main_thread(void);

/**
 * @brief Destructor.
 */
~main_thread();

/**
 * @}
 */
```

Recommended sections name are:

```
@name Types & Constants

@name Constructors & Destructor

@name Operators

@name Public Member Functions

@name Public Static Member Functions

@name Private Member variables
```

Samples

Typedefs or using

Use plural.

```
@brief Type of variables holding mutex recursion counters.
```

Extra line between `@brief` and `@details`

For headers which define `@details` too, add an extra line.

```
@brief Type of variables holding scheduler state codes.

@details
Usually a boolean telling if the scheduler is ...
```

- Home
- News
- Releases
- Support
- About