RODOS

Coding directives

Version: 2.2
Date: 2019, 2022
Author: S. Montenegro

1. Purpose

The purposes of the coding directives are a) to achieve a uniform programming style in the whole system and b) to avoid typical programming errors. Having a uniform style helps programmers to understand code from other programmers, and to make reviews and error-searching simpler and more effective.

We formulated these coding directives from our experience with space projects. The first version of these directives was applied for the development of the BIRD Operating System. In the current version of the coding directives we have added selected recommendations from ESA (European Space Agency), ESTEC (European Space Research and Technology Centre) and NASA (National Aeronautics and Space Administration, USA).

This coding directives describe the programming style applicable for the OS + Middleware kernel and application tasks, including the mission dependent and mission independent tasks.

1.1. Automatic Formatting

For automatic formatting please use

clang-format

See the script rodos/scripts/help-scripts/rodos-formatfiles.sh
In the root directory of RODOS you will find the file

.clang-format

Which implements this coding directives.

2. Documentation

Doxygen comments
http://www.doxygen.nl/index.html
are recommended to be used to document code classes, attributes and methods.
These comments must always immediately precede the item they are documenting.
Doxygen comments either use /// for single line comments or /** */ for multiple lines.
Normal comments use // and /* */.
In multi line comments, each line shall start with a *.

(Doxygen) documentation of classes must include :
a brief synopsis
a detailed description
an example code segment using the class
an author list using the @author tag

Any complex code or non-trivial sections should be commented with the meaning of these lines and the initials of the author.

DO NOT contribute to comment inflation. Comment only thing which can not be seeing in the source code.

In RODOS, if you find a comment, read it! It is important.

3. Style

3.1. Types of cases

The following naming patterns are used throughout this document: See https://en.wikipedia.org/wiki/Letter_case#Special_case_styles

- `UpperCamelCase`
- `lowerCamelCase`
- `SCREAMING_SNAKE_CASE`
- `UPPERCASE`
- `snake_case`
- `kebab-case`

The following naming patterns are never used throughout this document:

- `TRAIN-CASE`

3.2. Use of different case styles and general rules

- Don't merge multiple words into one. e.g. not `starttime` but `startTime`)
- **Acronyms** are written in UPPERCASE. eg. `checkCRC()`
- **Functions** are written in lowerCamelCase. e.g. `doSomethingNice()`
- **Variables** are written in lowerCamelCase e.g. `commandCounter`
- **Compile-time constant variables** use SCREAMING_SNAKE_CASE. e.g. `STACK_SIZE`
- **`Folder names`** and **`file names`** use kebab-case. e.g. `command-interpreter/command-reader.cpp`
- **Namespaces** use UpperCamelCase. e.g. `MyNamespace`) or UPPERCASE, for acronyms:. eg `RODOS`
- **Classes** are written in UpperCamelCase. e.g. `TimeManager`
- **Class member variables** use lowerCamelCase. e.g. `timeNow`
- **Methods** use lowerCamelCase (see functions)
- **Enums** use SCREAMING_SNAKE_CASE. e.g. `RESET_COMMAND_COUNTER`
- **Preprocessor definitions** use SCREAMING_SNAKE_CASE. e.g. `NOW()`
- **Boolean** variables shall begin with "is", "shall", "has", or simmilar.
- Variables should be declared just before they are used for the first time.
- Avoid similar names for different variables like UDPmsg and udpMsg
- The name of a function shall be an imperative (do something) or in question form (for bool functions)

3.3. General conventions

- All `#include` must be at the beginning of the file, then namespace, then Doxygen comments, then class
- Maximum line length is 120 characters
- All indents are 4 white spaces
- Use blank lines to structure the code
- Attach curly brackets to the previous block
- No magic numbers and strings, use constants with suitable name and const keyword
- Different classes should have different names, even in different namespaces (except they implement the same functionality)
- Avoid similar names for variables, not `cmdCounter` and `commandCounter` in the same programm
- Variables in macros shall begin with _. eg `SET_TIME(_time)`
- Order in classes: Attributes, constructors, methods
- Function and method names should tell what the method does (e.g. `setTime(int64_t t), getTime(), doSomething(), isEnabled()`
- Functions and methods should be as simple as possible

3.4. Recommendation for clarity and safety

Methods should be specialized to a single action, which can be named clearly. Avoid creating huge functions that do a lot of independent operations sequentially.

User-defined types (classes and structs; types that are bigger than an int) shall be passed by reference. Unless the intent of the method is to modify an argument, the argument in the method signature should be declared as a reference to a const object

Code shall never return a reference or pointer to a local variable

The formal arguments of methods shall have names, and the same names should be used for both the method declaration and the method definition.

When declaring methods, name and parameter shall be in the same line.

Member functions, which do not alter the state of an object, shall be declared “const”

Public member functions may not return non-cost references or pointers to member-variables of an object

Preprocessor macros, which contain parameters or expressions, must be enclosed in parentheses. eg `#define ADD(_a, _b) (_a) + (_b)`

Do not compare Boolean values with true, e.g. if(boolx == true) /* WRONG */

Do not use spaces around the "." and "->" operators or between unary operators and their operands.

The programmer may not assume that different data types have equivalent representations in memory

The programmer may not assume knowledge of how different data types are aligned in memory

When declaring methods, the leading parenthesis and the first argument (if any) should be written on the same line as the method name. eg:

```
void sendMsg(void* msg) {
    ...
}
```

3.5. Embedded Programming

- no c++ exceptions
- no RTTI (Run Time Type Information)
- only very simple templates (only to simplify code, highly recommended for
- type-safe)
- no multiple inheritance
- no dynamic memory allocation (no new, no malloc)

4. Flow control

- Don’t use nesting levels greater than 6
- Don’t nest the ternary operator (?:)
- Methods and functions should only have one exit point, or return on exeption. eg `float sqrt(float x) { if (x <0) return 0; … compute… return val; }`
- Goto should not be used
- Code that follows a case label shall be terminated by a break statement. If several case statements execute the same block of code, i.e. fall-throughs, they must be clearly commented

5. General advise

- Use parentheses to make the intentions clear
- Use parentheses around bitwise operators
- No spaces around . and →
- Use spaces around operators, unless parts shall be grouped
- If in doubt, use parentheses

- #include <header.h> for system headers, #include “header.h” for user headers
- #include may not contain full path
- Macros should have no side effects
- DO NOT redefine the language with macros
- Don’t assume anything about the data representation in the system
- Don’t mix integer and pointer arithmetic

We use clang-format to create nice to read programms. The format directives are in the `rodos-root-directory/.clang-format`

And finally: **Keep it simple**, the more simple the better.

Do not try to optimise it, if it isn't necessary! It is better to keep it simple and clear.

Do not try to save some bytes of memory by clever coding! This is strictly not necessary.

Do not try to save some microseconds by optimisation, if not necessary!

Simply put: if the code is already good then don't try to make it any better!