Rocaloid Code Standard

Language: C

Version 1, Revision 0

---

1. Preliminaries

1.1 About

This standard instructs collaborators to contribute high-quality codes to Rocaloid, and helps them better understand others' works within Rocaloid Project.

The scope of this standard applies to all **codes written in C** in Rocaloid Project, excluding those in the front end.

1.1.1 About this Revision

- 1.2: Broaden definition of user.
- 2.2.1: Add newline for comments.
- 2.4: Add specification to string literals.
- 4: Add restriction to the use of Function Declarators.
- 5.1: Add class declaration standards.
- 5.2: Additions to Naming.
- 5.3: Add specification to macros.
- 5.4: Add specification to RTemplates.
- I.1: Add RTemplate examples.
- Add bibliography.

1.2 Definitions

- `Component`: refers to the source code of a library or executable.
- `User`: refers to anyone who utilize certain source codes(can be components or parts of codes) under Rocaloid Project.
- `Contributor`: refers to anyone who contribute codes to Rocaloid Project.
- `Class`: refers to a structure declared by `RClass` of RUtil2.
- `Interface`: refers to any **standard formed by codes that contributors and users share** within the published codes.
- `Word`: a chain of characters that follow the Upper Camel Case naming method.
- `Object`: region of data storage in the execution environment, the contents of which can represent values. (WG14/n1256)

2. Format

2.1 File Extension

- Headers should have `.h` file extension.
- RTemplate headers should have `.h` file extension and `_` prefix.
- Sources, except RTemplate sources, should have `.c` file extension.
- RTemplate sources should have `.rc` file extension `_` prefix.
- Natural docs documentations should have `.txt` file extension.
- Other markdown-written documents should have `.md` file extension.

2.2 Line

- Tabs should be replaced by space. Tab size should be 4 characters.
- Maximum length for a line is 80 characters.
- When the length of a line exceeds 80 characters, the line should be separated and continued in the next line, with an indent of 4 more characters.

2.2.1 Newline and brackets

Braces are separated to be in the next line in the following situations:

- After `if`.
- After `else`.
- After `for`.
- After `while`.
- After `do`.
- After `switch`.

And are not separated when:

- Before `else`.
- Before `while` in `do` blocks.

The separated braces **should not** be indented. But the succedent block items **should** be indented.

Comments which start at the beginning of a line should leave one blank line above each of them.

2.2.2 Exceptions

When a block has only one item, its braces should be omitted. But the block items are still indented. The above rules still apply to the items.

2.3 Spacing

Additional spacing is required **before and after** these punctuators:

```
-> *(as multiply) / + - % << >> < > <= >= == != &(as bitwise AND operator) ^ | && ||
? : = *= /= %= += -= <<= >>= &= ^= |=
```

Additional spacing is required **before but not after** these punctuators:

```
++ --
```

Additional spacing is required **after but not before** these punctuators:

```
, &(as address operator) ~ ! *(as abstract declarator)
```

Additional spacing is **not required** before or after these punctuators:

```
. [ ] ( ) { } ; # ## *(as indirection operator)
```

2.3.1 Rule of combination

When **one of** the above rules is satisfied, a space should be inserted before or after the punctuator, except when a space(**including indent**) already exists in the target location.

2.4 Other

The above rules apply to macro definitions.

The above rules apply to string literals.

3. Naming

All identifiers follow an extended **Upper Camel Case** naming method.

3.1 External Identifier

`External Identifier`:

- `{Annoymous Prefix(opt)}{Namespace Abbreviation}_{Library-Scope Identifier}`

`Library-Scope Identifier`:

- `{Class Identifier}_{Annoymous Prefix(opt)}{Method Identifier}`
- `{Annoymous Prefix(opt)}{Function Identifier}`
- `{Annoymous Prefix(opt)}{Object Identifier}`

`Class Identifier`:

- `{Class Name}_{Class Architecture(opt)}_{Class Type(opt)}`

`Method/Function Identifier`:

- `{Function Name}_{Function Architecture(opt)}_{Function Type(opt)}`

`Object Identifier`:

- `{Object Name}_{Object Architecture(opt)}_{Object Type(opt)}`

`Function Name`:

- `{Function Prefix(opt)}{Function Core}{Function Postfix(opt)}`

Namespace Abbreviation should not be used inside RUtil2.

Class/Function Architecture and Class/Function Type can either be generated by RTemplate or be manually specified.

3.1.1 Annoymous Prefix

Annoymous Prefix is used to hide an identifier from programmers, to prevent naming collision.

An Annoymous Prefix can be either

- `_`: to hide an identifier from users.
- `__`: to hide an identifier from both users and contributors of the component itself.

3.1.2 Namespace Abbreviation

The abbreviation of a namespace, usually shorter than 7 characters and upper case(but not compulsively). A namespace can be the name of a component.

Examples:

- `CDSP2`: abbr. Cybervoice engine Digital Signal Processing library 2
- `CSVP`: abbr. Cybervoice engine Spectral Voice Processing library
- `RFNL`: abbr. Rocaloid Fast Numerical Library

3.1.3 Class Name

The name of a Class.

Examples:

- `FWindow`: Fast Window generator(see RFNL/src/FWindow/).
- `STFTIterlyzer`: STFT iterative analyzer(see CVESVP/src/Iterator/_STFTIterlyzer.h).

3.1.4 Class Architecture/Function Architecture

The CPU/GPU instruction set that the class/function targets at.

Examples:

- `Gnrc`: Generic architecture support.
- `SSE`: SSE instruction set support.

3.1.5 Class Type/Function Type

The data type(s) that the class/function operates on.

Class Type/Function Types should be words.

When multiple types have to be specified, join them with `_`.

Examples:

- `Float`.

- `Float_Double`.

3.1.6 Function Prefix

When a function/method has variants with different set of parameters or different return types, a Function Prefix is added before the Function Core.

Function Prefix should be an abbreviation that is less than 6 characters and in upper case.

Examples:

- `V`: Vector operation(see RFNL/src/Vec/).
- `VC`: Vector operation with one constant argument.
- `IR`: Inverse Real operation.
- `FW`: Accepts a pointer to FWindow(see CVEDSP2/src/Container/_Wave.h).

3.1.7 Function Core

The essence that describes the behaviour of the function.

Examples:

- `FFT`: Fast Fourier Transform(see RFNL/src/DFT).
- `Mul`: Multiply(see RFNL/src/Vec).

3.1.8 Function Postfix

When a function/method has variants with same set of parameters and return types but different internal mechanisms, a Function Postfix is added after the Function Core.

Function Postfix should be an abbreviation that is less than 6 characters and in upper case, or word(s) joined to the Function Core with `_`.

Examples:

- `_LPrec`: Low Precision(see RFNL/src/FMath/).
- `W`: Windowed(see CVEDSP2/src/Container/_Wave.h).

3.1.9 Object Name

The name of an object.

3.2 Function Parameters

- Always use `Dest` to represent the data destination of an operation.
- Always use `Sorc` to represent the data source of an operation.
- Always use `Size` instead of `Length`.
- When multiple destinations or sources have to be specified, add numbers after them. e.g., `Dest1`, `Dest2` ...
- When conjuate destinations or sources or other parameters have to be specified, add postfixes after them, 4 characters long is preferred. e.g., `SorcReal`, `SorcImag`.

3.3 Other

The above rules apply to macro definitions.

_ in Function Cores, Object Names, and Class Names should be avoided, **if possible**.

4. Language

The C programming language standard for Rocaloid Project is ISO/IEC 9899:1999, with the additional restrictions:

- Type qualifiers(`const`, `restrict`, and `volatile`) should never be used.
- Function declarators should be avoided except when the storage class specifier is `typedef`.

4.1 Scope-specific Standards

- `register` specifiers should never be used, except in RFNL.
- `volatile` qualifiers can be used in RFNL.

5. Interface

5.1 Class

5.1.1 Class Declaration Standards

- `RInherit` should always be placed in the first.
- Comment `//Public` should be placed above the struct declarations which are intended to be visible to the users of the class.
- Comment `//Private` should be placed above the struct declarations which are intended to be invisible to the users of the class.
- `//Public` and `//Private` comments should only appear once in each class declaration.

5.2 Functions

5.2.1 From/To Notations

- For functions that accept, process and return any data, it is recommended to name those functions(Function Core) as `AFromB` or `BToA`. The former is called From Notation; the later is called To Notation. In such case, A is called Function Destination; B is called Function Source.
- Operative Source is the class name on the left side of a From or To Notation; Operative Destination is the one on the right side.
- For methods of such type, _ should be inserted after the Operative Source. e.g., `A_FromB`, `B_ToA`.
- From Notation is more recommended than To Notation, except when:
    - Two-way conversions are needed, but the data of one class(Derived Class) originates from the other(Source Class). In such situation, the Operative Destination of both From Notation and To Notation named functions should be Function Destination; the Operative Source of both should be Function Source. e.g., `SourceToDerived` `SourceFromDerived`.

5.2.2 Formal Parameters

- For methods, the first parameters should always be "pointer to Class".
- For other functions, the first parameters should always be the destination of operation, except when their results are returned.
- In the following parameters, sources of operation should have the highest priority.
- **Do not** use `Dest` or `Sorc` to represent data with non pointer data types.

5.2.3 Return Type

- Functions are not recommended to return a pointer. If so, their Function Cores or Function Prefixes should contain `Alloc` or `AL`.

5.2.4 RInterface

- The Function Prefix of a RInterface should start with `I`, which represents "Interface".

5.3 Macro

- **Do not** define macros in headers if the scope of macro is not intended to be every file that directly or indirectly includes the header.
- When defining a macro in RTemplate files, the macro should be defined(`#define`) in the start of the file, and undefined(`#undef`) in the end. The identifier after `#undef` should be aligned with that after `#define`. i.e., to insert two spaces after `#undef`.
- When macro is used to abbreviate class names in RTemplate files, the macro should be named as `_{ClassName}`. The example is given:
    - `#define _{ClassName} _C({Namespace Prefix}{{Annoymous Prefix(opt)}{Namespace Abbreviation}_{ClassName}, _, _T1, _, _T2, ...}`
    - `#undef _{ClassName}`
- Use macros to avoid repitition of code snippets if possible.

5.4 RTemplate

- Use RTemplate to avoid repitition of class declarations if possible.
- When dealing with float-point data types, RTemplate should be used to provide both single and double precision float-point support.
- The format of a set of files that defines a template class with RTemplate is given by example. Please refer to Appendix I.1.
- When multiple template classes are declared using the same header or source(**not template header or template source**), `#include <RUtil2.h>` should be inserted before each declaration.

Appendix

I. Examples

I.1 Template Class Declaration

ClassName.h:

```
#ifndef LIBRARY_CLASSNAME_H
#define LIBRARY_CLASSNAME_H

#include <RUtil2.h>

#if 0
    #include "_ClassName.h"
#endif

#ifdef __Library_Install
    #define _RTAddress "LibraryPath/DirectoryPath/_ClassName.h"
#else
    #define _RTAddress "DirectoryPath/_ClassName.h"
#endif

#define _RTClassName _ClassName
#define _Attr 1

#define _T1 Float
#include <RUtil2/Core/RTemplate.h>

#define _T1 Double
#include <RUtil2/Core/RTemplate.h>

...

#endif
```

ClassName.c:

```
#include "ClassName.h"
#include <RUtil2.h>

#define _RTAddress "DirectoryPath/_ClassName.h"

#define _RTClassName _ClassName
#define _Attr 1

#define _T1 Float
#include <RUtil2/Core/RTemplate.h>

#define _T1 Double
#include <RUtil2/Core/RTemplate.h>

...
```

_ClassName.h:

```
#define _Type _C(Library_Type, _, _T1)
```

```
RClass(_RTClassName)
{
    RInherit(RObject);

    //Public
    ...

    //Private
    ...
};

...

#undef  _Type
```

_ClassName.rc:

```
RCtor(_RTClassName)
{
    ...
    RInit(_RTClassName);
}

RDtor(_RTClassName)
{
    ...
}

...
```

II. Revision History

Version 1, Revision 0

- 1.2: Broaden definition of user.
- 2.2.1: Add newline for comments.
- 2.4: Add specification to string literals.
- 4: Add restriction to the use of Function Declarators.
- 5.1: Add class declaration standards.
- 5.2: Additions to Naming.
- 5.3: Add specification to macros.
- 5.4: Add specification to RTemplates.
- I.1: Add RTemplate examples.

Version 0, Revision 1

- 1.2: Add definition of Object.
- 3.1: Add Object Identifier.
- 3.1.9: Add definition of Object Name.
- 3.3: Avoiding underlines.
- Exchange chapter 4 and 5.
- 4: Add additional language standards.

Bibliography

1. WG14. "ISO/IEC 9899:1999 - Programming languages - C". open-std.org.