C Source Code Style Guide

Internal Standard OKTL-0000034

Abstract

This document describes the general formatting rules of C programming language constructs, naming conventions and coding style.

Copyright Notice

Audience

This document is intended for OKTET Labs C programmers, reviewers, testers and other persons involved in software development process.

It is assumed that a reader is familiar with the C programming language.

Document status

Draft

Contact Information

Unit 3H, Universitetsky pr., 18/2, Petrodvorets, St.-Petersburg, 198504 Russia
Phone: +7(812) 428-6709
Fax: +7(812) 784-6591

For more information, call the above office or visit our web site at http://www.oktetlabs.ru.

Please, send your comments and notes about found errors, typos, and inconsistencies to support@oktetlabs.ru.

Revision History

Date	Rev	Author	Changes
1999	A	Victor V. Vengerov	Initially written
Feb, 23 2003	B	Dmitry V. Semyonov	Typos fixed. Reformatted according to OKTET Generic Document Template. Added doxygen related information.
Mar, 4 2003	C	Dmitry V. Semyonov	Fixes after review
Mar, 18 2004	D	Elena A. Vengerova	OKTET Labs identifier is assigned
Aug, 2 2004	E	Andrew Rybchenko	Contact information updated. Typos fixed. Victor V. Vengerov notes applied.
Jul, 15 2006	F	Andrew Rybchenko	More strict requirements on function parameters and return values description. Add "Safe programming" chapter.

Issues

- The document should be updated after more experience of Doxygen usage is achieved.

Table of Contents

1 Abbreviations

'ANSI'	American National Standards Institute
'CR'	Carriage Return
'CVS'	Concurrent Version System
'LF'	Line Feed
'WRS'	Wind River Systems

2 Terminology

'Doxygen' A documentation system for C++, C, and several other programming languages. See 'Doxygen manual' for more information.

'Splint' A tool for statically checking C programs.

'Subsystem'
Functionally closed, alienable, possibly reusable, software component with well defined external interfaces. It may be library, tool, device driver, part of OS, etc.

'Subversion'
A free/open-source version control system designed to be the successor to CVS.

3 References

This work is inspired by many existing coding standards and our own experience in C software development and understanding of foreign source code. Most notable documents which have been used to create this standard are

- L.W.Cannon, R.A. Elliot et.al. "Recommended C Style and Coding Standards" (based on Indian Hill C Style and Coding Standard).
- R. Stallman "GNU Coding Standards" (March 1998, (C) Free Software Foundation)
- J.K. Ousterhout Tcl/Tk Engineering Manual
- WRS Coding Conventions
- Doxygen manual

4 Introduction

The main purpose of this standard is to define a framework which can help us to develop source code in a clear and uniform manner. Another goal is to define a base for automatic generation of convenient on-line and printable documentation directly from sources.

This document defines the "low-level" C coding conventions, structure of C source files, naming conventions, commenting style which must be applied to developing of C source code. We have tried not only give a pandect of rules, but also give rationale and reasons for some of them.

Doxygen-specific guidelines concerning resulting documentation formatting and common issues will be described in a separate document.

4.1 Contributors

- Victor V. Vengerov Victor.Vengerov@oktetlabs.ru
- Elena A. Vengerova Elena.Vengerova@oktetlabs.ru
- Konstantin Abramenko Konstantin.Abramenko@oktetlabs.ru
- Dmitry V. Semyonov Dmitry.Semyonov@oktet.ru
- Andrew Rybchenko Andrew.Rybchenko@oktetlabs.ru

5 General

5.1 Style remarks

C language has a pretty simple syntax, nevertheless it is possible to write extremely complicated code. The following example, taken from *"Obfuscated C Contest"* archive, is a correct C program; you can try to compile and execute it:

```
main() { printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60); }
```

How much time you need to understand how it works? :-)

It is good for fun, but absolutely not applicable for development of a big software project, which must be supported by many people during few years. Your code may be read by many people, and it is very important to make your code understandable for them. Every time have in mind that you are writing code not for compiler, but for human. Limit your fantasy to make your code clear.

5.2 Text width

Any line of C source code couldn't be longer than 80 characters.

Remember that during debugging and further development of your program you often insert additional `if` statements moving your original code more and more to the right. Try to keep your original code in 60-70 character limit.

If C source code is generated by other programs, tools, scripts, and produced code is not intended to be human-readable, it is not required to follow 80 character limitation.

5.3 File length

Source code file length is not limited explicitly. More important aspect in division of source code to many files is to reflect logical structure of specific program or project. Nevertheless, each file must have a reasonable size. It is not possible to give hard limitations of source length; it seems that 800-4000 lines of code is a good source size and up to 6000-8000 lines is acceptable.

If C source code is generated by other programs, tools, scripts, the file length is not limited.

5.4 Special Characters

Lines in source file are separated by character '`LF`' (hex code 0x0a). It is not allowed to use MS-DOS standard '`CR-LF`' sequence for line delimiting.

It is permitted (but not required) to insert '`FF`' character (Form Feed, hex code 0x0c) to separate logically independent parts of source code (functions or function groups, variable declaration groups, etc). This character shall not be used inside function body.

It is prohibited to use control characters immediately in string constants or comments. For strings you may use escape sequences instead.

Do not forget to setup your text editor program so that it will insert appropriate number of spaces instead of '`TAB`' (hex code 0x09) character on ⟨TAB⟩ key pressing.

5.5 Trigraphs

Do not ever use the "trigraphs" feature of ANSI C. (If you don't know what does it mean - don't worry, it will be better if you never know about it).

5.6 Indentation

Indentation is used to show structure of algorithms or data types. Usually, each statement in complex statement is started at the same column, and each nested statement is indented relative to its "parent". Precise rules specific to each C language construct provided below.

Indentation level is 4 character. It means that every nested block shifted to 4 character right relatively to the position of its "parent block".

If a nested block is a complex statement surrounded with curly brackets (braces), these brackets must be placed at the position of "parent" block.

The following example demonstrates this:

```
while (p != NULL)
{
    for (i = 0; i < p->tabsize; i++)
    {
        if (p->tab[i] == x)
            return i;
    }
    return 0;
}
```

5.7 Vertical spacing

- Use blank lines to make code more readable. Group logically related sections of code together.
- It is not recommended to put more than one declaration to the line. (It is allowed to put two-three declarations of tightly-related variables in one line to emphasize their relationship).
- It is *highly not recommended* to put more than one statement on a line. (It is allowed to put two-three very simple and tightly-related actions (something like `i++; j++;`) on the one line.) Exception: `for` statement, when initial, condition and loop statement could be written on a single line; see templates below.
- Curvy braces (`{` and `}`) always (except `do...while` loop) get their own line.

5.8 Special compiler features

Usually, each compiler provides some non-standard facilities, which are not supported by other compilers. Examples are assembler insertions, function or parameters attributes which make possible, for example, associate parameters with a specific processor registers, nested functions, and so on. You must understand that usage of such features will complicate porting of your code very much. It is *highly* not recommended to use such features. In some situations, usage of such features may be acceptable. In any case, it may be used only in a limited number of modules, and preferable under conditional preprocessor directives. Compiler-depended features like `inline` function attribute must be used via appropriate preprocessor definition.

6 Naming conventions

Choosing names is one of the most important aspects of programming. Good names clarify the function of a program and reduce the need for other documentation. Poor names result in ambiguity, confusion and errors.

The ideal name is one that instantly conveys as much information as possible about the purpose of an entity (variable, function, field, etc.) it refers to. Try to choose names in such way that they could not be misinterpreted or confused by other programmers.

Here is the check list for names:

- Check that the name is consistent. Use the same name to refer to the same entity everywhere. For example, if you have passed a pointer to some structure descriptor in each function in a module, give the same name to the correspondent parameter.
- Check if the name may be misunderstood out of context. For example, if function which performs swapping of odd and even bytes is called `swap_buffer()`, this name may be understand incorrectly in other context: somebody may think that you have meant swapping buffer to the disk.
- Check if the name may be confused with some other name. For example, if two variables with names `str` and `string` in the same procedure both refer to strings, it is hard to anyone to remember which is which.
- Check if the name is so generic that it doesn't convey any information.

Words and abbreviations from non-English languages cannot be used to form identifiers. Use underscore character '_' to separate the words. Do not fear to use long identifiers. In any case, an identifier *must not contain* pronouns, interjections, private names, nicknames, or place names.

Variable, type, field names must be in lower case only. Use upper case for preprocessor names and enum constants. Do not mix upper and lower case letters in the same name.

If you have chosen some abbreviation, document it and use the same abbreviation everywhere in the program.

You must have in mind possible name conflicts when you assign a name to an entity. A conflict may occur if two global variables or functions are defined in two places, or if some header files contain declaration of types, preprocessor variables, structs, etc. with the same name, and these headers may be included both. The only way how to avoid such conflicts is to give names in a consistent manner.

You must add module- or subsystem-specific prefixes to globally visible names. It is recommended to have prefix not less than 3 character long; 2-character prefix is allowed for some significant wide-used subsystems. A prefix must be the same for all global names in module or subsystem. All globally-visible variables and functions must be prefixed. Also, it is highly recommended to give prefixes to all declarations, type names, preprocessor variables, struct names in header files which may be included from external components. It is not required (but allowed), to prefix local names (names in the header files which may be included only in particular subsystem and not exported to the outer world, static variables and functions). Do not prefix field names in structures and unions.

It is recommended for uniformity to use '`object + action + object/property`' style while giving a name to a function. Besides, this style is convenient when module prefix is used in place of the first object. For example, `fat_get_cluster()`, `pkt_send_raw()`.

The only exception for the above rules is if you are implementing something using externally given (standardized) specification. Therefore, it is not required to rename `printf` to `libc_printf` if you are implementing ANSI C library!

7 Conventional comments

Comments are special remarks intended first of all for human reader, who needs to understand your work. Comments are very important parts of your code, it is a way to make your program simple and well-readable. Lack of comments or inaccurate comments may make life for reader (and may be for you too!) very complex.

This chapter provides general rules for comment writing.

7.1 Comment designators

Comments are started with the sequence `/*`. Comments are finished with the sequence `*/`. It is prohibited to use C++ line comment designator `//` even if your compiler supports it. Nested comments are not allowed.

Don't use `/*` and `*/` sequences for temporary removing pieces of your code. Use preprocessor directives instead.

7.2 Language

All comments in your program shall be written in English, because English is the only language that nearly all programmers in all countries can read.

If it is declared in technical requirement document, that comments must be written in any other language UTF-8 character set should be used.

Multilingual comments within sources are *highly* not recommended. It is proposed to use separate translation files for such cases.

7.3 Formatting conventions

The text of your comment must follow some simple usual conventions. Statement must be started from capital letter (except case when the first word is a name of variable, type etc. You must avoid this, but if you need, write name exactly as it is used). Put one space character between words in sentences. Couple punctuation mark to the left word and insert a space character after punctuation mark. Put an empty line between paragraphs. You must put space character after `/*` and before `*/`.

Two comment styles may be used: one-line comments and multiline comments. One line comment looks like this:

```
/* This is example of one-line comment */
```

Multiline comments look like this:

```
/*
 * This is example of multiline comment. Such comment layout must be used
 * if your text couldn't be fitted in one 80-character line. Please, note
 * that each line is started with the asterisk character.
 */
```

Code comments must be indented as appropriate piece of code.

Some specific comment formatting considerations provided in appropriate chapters of this document.

7.4 Comments style

The purpose of comments is to save time and reduce errors. Comments are typically used for two purposes. First, people will read the comments to find out how to use your code. For example, they will read procedure headers to learn how to call the procedures. Ideally, people should have to learn as little as possible about your code in order to use it correctly. Second, people will read the comments to find out how your code works internally, so they can fix bugs or add new features. More documentation isn't necessarily better: reading pages of comments may be not easier than understanding uncommented code.

It is most important to comment things which affect many different pieces of program. Thus it is required that every procedure interface, every structure declaration, and every global variable must be commented clearly. If you haven't commented one of these things it will be necessary to look at all the uses of the thing to understand how it's supposed to work.

On the other hand, things with only local impact may not need much documentation. If the overall purpose of the function has been explained, and if there isn't much code in the procedure, and if variables have meaningful names, it is not required to give any comments at all. For long functions, comment local variables and logical blocks of function. Also, comment a tricky situations, non-obvious code which depends on information to someone reading it for the first time. Non-*Doxygen* style (`/* ... */`) should be used for comments in function body (it saves two symbols and avoids confusion since it is not included in documentation generated by *Doxygen* in any case).

Document each thing in exactly one place. If something is documented in several places, it will be hard to keep the documentation up to date as the system changes. Try to document each major design decision in exactly one place.

Don't just repeat what is already obvious from the code, like this:

```
i += 1;   /* Increment i */
```

Documentation should provide higher-level information about the overall function of the code, helping readers to understand what a complex collection of statements really means.

It is extremely important to write comments as you write the code. Don't delay documentation writing until you finish to implement and change your code. You will never write good comments in such case, because you will never stop coding: there is always more code you have to write. If you accumulate more undocumented code, it will be harder to you to find energy and time to comment it. So, you just write more undocumented code. Finally, you may forget some important things which must be documented.

If you write comments as you go, it won't add much to your coding time and you won't have to worry about doing it later. Therefore, it is required to write comments as you write the code.

8 File layout

This chapter describes general rules about C source and headers file layout.

8.1 Order of file sections

Every file containing C source code must be started from standard file header. After the header the following sections should be included in the specified recommended order:

- General file documentation. It is a C comment consisting of a complete description of the overall module purpose and function.
- *Includes*. The include section (if exists) consist of one or more C preprocessor `#include` directives. This section groups all header files included in the module in one place. In most

cases, system include files, such as '`stdio.h`', should be included first. Subsystem interface include files (see below) usually must be included before subsystem headers. Header files that declare functions or external variables should be included in the file that defines the function or variable.

- *Defines*. The defines section (if it exists) consists of one or more C preprocessor `#define` directives. This section (if it exists) groups all definitions made in module in one place.
- *Types*. This section (if it exists) contains all struct, typedef, enum definitions and groups it together.
- *Forwards*. The forward declarations section (if it exists) consists of one or more ANSI C function prototypes. This section groups all such declarations together.
- *Globals*. This section (if it exists) consists of all external-visible variables definitions and groups it together.
- *Locals*. This section consists of all variables visible only inside this module (static variables) and groups it together.
- *Functions*. The functions section (if it exists) consists of one or more C function definitions. This section groups all such declarations together. In this section similar functions (functions on a similar level of abstraction) should be placed together.

If you see that it is better to repeat some of these sections two or more times to group similar things together, you may do it, saving provided order in each such group. However, all functions should be implemented after all preceding definitions anyway. Nevertheless, remember that it means that your software is likely to have a poor design. It is recommended to split source or header file to many smaller ones in this case.

8.2 Human-written file headers

Each '`.c`' or '`.h`' files written by human, must have a formal header. This header starts at the first line of source file. A header has the following format:

```
/** @file
 * @brief <subsystem description>
 *
 * <module description>
 *
 * <copyright notice>
 *
 * @author <author name> <<author e-mail>>
 *
 * <license term notice>
 *
 * $Id: $
 */
```

Subsystem description is a short (one-line preferred) description of subsystem to which this file belongs. This line must be same for all files of this subsystem.

Module description is a short description of a source file. It should be started with upper-case letter, and finished with dot.

Copyright notice must provide information about copyright holder of this work. This line must consist of word '`Copyright`', copyright sign '`(C)`', organization name which holds copyright, and its location (city and country). It is allowed multiple '`Copyright`' sections if your source contains fragments copyrighted by different organizations.

`@author` section of the header must provide information about principal developer of this module. The section consists of the author's name (in English transcription) and e-mail address which may be used later to contact the author regarding to these sources. It is allowed to have multiple '`@author`' sections if principal developer has been changed; put a name of the last principal developer first. OKTET Labs employees shall use <Name.Surname@oktetlabs.ru> form of e-mail address.

Optional part '`license term notice`' may be added to provide information about licensing terms for this source file, about distribution permissions or restrictions, or to refer to a particular license text.

The last line of the header is a substitution for source code system (e.g. CVS, *Subversion*) information. '`$Id: $`' sequence will be substituted with actual module identification string while a file is committed into repository.

Note that doxygen-style comment and tags are used here.

```
/** @file
 * @brief ncr875 controller device driver
 *
 * Chip-specific primitive implementation.
 *
 * Copyright (C) 2004 OKTET Labs, St.-Petersburg, Russia
 *
 * @author Victor V. Vengerov <Victor.Vengerov@oktetlabs.ru>
 * @author Andrew Rybchenko <Andrew.Rybchenko@oktetlabs.ru>
 *
 * $Id: $
 */
```

8.3 Generated file headers

Some of source files fed to compiler may be generated by other programs or scripts. Such programs or scripts must put special header at the beginning of such files:

```
/*
 * This file is automatically generated by program <program>.
 * DO NOT EDIT IT!!!
 *
 * File generated at <date> <time>
 */
```

Format YYYY-MM-DD should be used for date specification.

8.4 Header ('.h') files

Header files are started with the same header as '`.c`' files. Header files must include C variable and function declarations, common type definitions, preprocessor definitions, definitions of inline functions which may be used in more than one C module. Each module which requires definitions from headers must include appropriate header using preprocessor directive `#include`. If some header file uses definitions or declarations from another header file, it is recommended to include that header file to the first one. (Usually, header file contains some interface definition, and it is poor if user of this interface must know about internal things needed for this interface declaration.)

There are two kinds of header files: which define interface to subsystem and which define inter-module interfaces inside specific subsystem. Try not to mix these two kinds of header files. Do not include internal definitions and declarations to the interface headers.

Header file must be started with the same header as C source code file. The body of header file must be organized in the following way:

```
#ifndef __<subsystem_prefix>_<filename>_H__
#define __<subsystem_prefix>_<filename>_H__

#include <nesting_header.h>
...

#ifdef __cplusplus
extern "C" {
#endif

...

#ifdef __cplusplus
}
#endif
#endif /* __<subsystem_prefix>_<filename>_H__ */
```

In this fragment '`<subsystem_prefix>`' must be substituted with capitalized subsystem abbreviation, '`<filename>`' must be substituted with capitalized form of header file name. If the filename already contains the subsystem prefix there is no need to add another one.

Conditionals `#ifdef __cplusplus` used to make possible usage of appropriate header in C++ program. These preprocessor conditionals may be omitted in internal headers, but are mandatory for interface headers. It is important to including nesting headers before `#ifdef __cplusplus` block in order not to affect symbols declared there.

9 Function Definition Formatting

9.1 Function header

It is required to put formal function header before each function definition. Function header have the following format:

```
/**
 * <function description>
 *
 * @param <param1>      <param1 description>
 * @param <param2>      <param2 description>
 * ...
 *
 * @return <return value description>
 *
 * @retval <retval1>    <retval1 specification>
 * @retval <retval2>    <retval2 specification>
 * ...
 *
 * @se <side effects description>
 *
 * @sa <closely related functions and documentation references>
 *
```

```
* @alg <algorithm description>
*
* @bug <bug description>
*
* @todo <todo description>
*/
```

Not all parts must be presented in each function header.

The only always required part is '`<function description>`'. It should be started with upper-case letter, and finished with dot. It is recommended to use common style of the form 'Copy bytes' (which is opposite to 'Copies bytes') for documentation similarity. If the function complies with some prototype, it should be stated in description.

At the same time it is *mandatory* to describe all function parameters and returned value(s). If a function does not return a value, '`@return`' part may be omitted, or <return value specification> may be '`N/A`' (not applicable). All parameter descriptions in a particular function must start from the same case: either upper or lower. If the description of the parameter or returned value is complete English sentence or is followed by additional statements, it must be started from upper-case letter and finished with dot. Description of return value after '`@return`' tag must be started from upper-case letter.

If a value directly returned by a function lies within a limited set of constants, then each constant should be described using '`@retval`' tag. Description of returned value after '`@retval`' tag must start from upper-case letter.

If a parameter is used to deliver values out from a function, it is allowed to add `[inout]` or `[out]` (depending on whether the parameter is used or not as input at the same time accordingly) hint just after `@param` without any space. If such hints are used, input parameters have to be marked using '`[in]`'. Thus, in/out hints should be used for either all or none parameters.

'`@se`' (side effects) part must be presented only when a function has non-obvious side effects. For example, if this function essentially modifies common data structures, and it is not clear from its name, or a function prints something to standard output, changes state of some hardware, and so on.

Use optional '`@sa`' (see also) part to give references to closely related functions and documentation blocks.

Optional '`@bug`' and '`@todo`' sections may be used to identify found bugs or things that need to be done accordingly.

'`@alg`' (algorithm) part must be provided only when it is not enough to define *what* function does; it is necessary to explain *how* this achieved.

9.1.1 Indentation inside function header

The function description as well as any of the sections beginning tags must be started from the 3rd column position (beginning with 0) just after the '` * `' sequence of characters.

Use exactly one '`SPACE`' character between any of section beginning tags and start of section body, if the body begins on the same line with a tag.

<paramX description>, <retvalX specification>, as well as any line (except first) of multiline section must be indented by multiples of 8 spaces. This simplifies text formatting with TAB key. The whole indented description must form a single column. In other words, it is not allowed to have a line indented by 8 characters, and other one indented by 16.

9.2 Function Formatting

All functions must be defined in ANSI C format. Function definition must look like this:

```
/**
 * Create object instance of class 'objclass' which designates to be
 * referred as a null object of this class. If 'name' is given
 * (not equal to NULL), register object instance with given name.
 *
 * @param  objclass     Class object identifier.
 * @param  name         Null object name, or NULL.
 *
 * @return
 *      Object identifier for null object of class 'objclass', or
 *      OBJID_NULL, if error occurred.
 */
static objid
object_create_null(objid objclass, char *name)
{
    objid obj;

    obj = class_instantiate(objclass);
    if (obj == OBJID_NULL)
    {
        return OBJID_NULL;
    }
    if (name != NULL)
    {
        int rc;

        rc = object_register_with_name(obj, name);
        if (rc != OBJ_SUCCESSFUL)
        {
            object_delete_instance(obj);
            return OBJID_NULL;
        }
    }
    return obj;
}
```

Please, pay your attention to the following topics:

- Open and close curly brackets must be placed at the column zero.
- Function name is started at column zero.

Such formatting will simplify use of some tools. For example, `grep` command can be used to find location of a specific function definition:

```
$ grep ^object_create_null *
```

If argument list can not fit in one line, this line can be split in the following way:

```
static objid
datafile_encode(int in_file_desc, int out_file_desc, char *encoding_key,
```

```
                    int encoding_key_len, int control)
{
    ...
}
```

9.3 Considerations

It is *highly not recommended* to pass or return structures to or from functions. Pass pointers instead and make a copy inside the function if it is really necessary.

If the function has no parameters, `void` must be used as the only item in the parameters list.

It is required to define a function with `static` specifier if the function must be visible only inside a current module (file).

Function return type must be declared. Do not default it to `int`; if a function does not return a value then it must be given return type `void`.

9.4 Function size

It is not possible to give hard limits for function code size. In any case, remember that it is better if function is pretty simple, pretty short and pretty clear. If it is possible, try to limit function size by 25-27 lines of code - such functions usually may fit on one terminal screen; this simplifies understanding what function does. If it is not possible, size up to 70-140 lines is good, and up to 400-500 is acceptable. If your function is overgrown, try to redesign it to isolate logically separable parts in different functions.

9.5 Parameters and local variables conventions

Function parameters may be divided into three categories.

- *In* parameters only pass information into the function (either directly or by pointing to information that the function read).
- *Out* parameters point to things in the caller's memory that the procedure modifies.
- *In-out* parameters do both.

Parameters should normally appear in the order `{`in, in-out, out`}`.

Parameters which have source/destination semantics must always appear in the order `{`destination, source`}`.

If there is a group of functions, all of which operate on structures of a particular type (hash table, device descriptor structure, etc), parameter which points to this structure should be the first argument of each function. All such parameters must have the same name.

Do not use `auto` memory class specifier with local variables. If you think that some (3-4 maximum) variables are used in a critical part of code or are frequently used, you may give `register` memory class specifier.

Avoid local declarations that override declarations at higher levels. In particular, local variables must not be redeclared in nested blocks.

9.6 Local variables naming conventions

It is recommended to give short but clear names to local variables. Here are some recommended names for local variables. All these names may be used with possible digit suffixes.

Names `i`, `j`, `k` must be used as names for loop variables.

Names `n`, `m` must be used as a names for variables which contain a number of some entities. Variable name `l` (ell) may be used for length of something.

Names `s` or `str` (only one of them in one function!) may be used as a pointer to character strings.

Names `s` or `sock` may be used for a socket file descriptors.

Name `fd` may be used to refer to a file descriptor variable.

Name `f` may be used for a file descriptor or `FILE` variable associated with file (*not* socket or device).

Name `fn` may be used for a string constanting the file name.

Names `p`, `q` must be used as a pointer to some data element (structures, elements of array, etc). It is acceptable to use `p` and `q` identifiers as pointers to character arrays too.

Names `t`, `tmp` must be used as temporary variables.

Name `src` may be used as a pointer to source of string, buffer, structure, etc. Name `dst` may be used as a pointer to destination.

Names `in` and `out` must be used for pointers or indexes in input and output data.

Name `c` may be used as a temporary variable for character.

Name `que` may be associated with some kind of queue.

Names `x`, `y`, and `z` may be associated with coordinates.

Names `w` and `h` may be associated with width and height of something.

Name `a` may be used for a some kind of addresses. Names `pa` and `va` may be used for physical and virtual addresses accordingly.

Name `conn` may be used for a file descriptor or other entity representing connection.

Example:

```
int i;

...

p = &queue_pool[0];
n = sizeof(queue_pool);
if (n > max_queue_size)
{
    n = max_queue_size;
}
for (i = 0; i < n; i++)
{
    p->data = NULL;
    p->id = i;
    p->next = p + 1;
    p++;
}
```

It is not allowed to have more than three variables with the same name and different suffixes.

Comments for local variables are necessary only when it is really necessary. You must comment non-obvious variables in long functions only. It is not necessary to comment variables which are used in clear way, for example loop counters, etc.

10 Types definitions formatting

10.1 `struct` declaration

10.1.1 `struct` comments

You must insert comment immediately before structure declaration which describes purpose of this structure, usage conventions and give other necessary information.

You must give a detailed comment for each field of declared structure.

Usually understanding of data structures used in program is a key to understanding of a whole system. Please, write accurate and detailed comments for structure declarations.

Comments for structure fields must be placed in the same line where a field is declared. All comments must be indented to the same level. It is allowed (but not recommended) to shift comments to the right from common level if some field names are longer than others and such shifts allow to avoid splitting comments to several lines. If comment is split a new line must be indented to the level of first comment letter following the `/*` sequence. See examples below.

10.1.2 `struct` formatting

The `struct` declaration must look like this:

```
/** Node of the double-linked list */
struct list_node {
    struct list_node  *next;    /**< Next node in double-linked list */
    struct list_node  *prev;    /**< Previous node in double-linked list */
    int                size;    /**< Information field size */
    char               node[0]; /**< Information field designator */
};
```

If `struct` declaration is used in a variable declaration or `typedef` declaration, `typedef` or variable specifiers must be placed before `struct` keyword on the same line, and identifier must be placed after close bracket on the same line:

```
/** NCR875 SCSI host adapter user-settable hardware parameters */
typedef struct ncr875_hw_regs {
    int diff_scsi;  /**< Differential SCSI bus enable */
    int tolerant;   /**< Tolerant enable */
    int slow_cable; /**< Slow cable mode (Extra clock cycle of Data Setup) */
    int burst_len;  /**< Maximum number of transfers performed per PCI bus
                         ownership, legal values is 2,4,8,16,32,64,128 */
} ncr875_hw_regs;
```

In `typedef` declaration, it is recommended to give the same name to `struct` name in `struct` namespace and to `typedef` name.

All field names must be started at the same position. The choice of the position is based on the maximum length of field type specifiers used in structure. Note that reference specifiers '`*`' must be coupled to field name, but placed at left from common field indentation level.

It is *highly not recommended* to join struct and variable declaration. Declare types and variables separately.

In some cases nested `struct/union` declarations are acceptable, but it is not recommended.

10.1.3 Bit fields

You may use bit fields to save memory if you know how much bits is sufficient to store some specific value. But in any case you should not make any assumptions about order in which these bit fields are allocated, because it may be compiler-specific. Therefore, it is prohibited to declare, for example, hardware register layout as a structure with bit fields, or represent structure of binary file in such way, or define structure of data which will be sent over network.

10.2 `union` declaration

All conventions provided for struct formatting are applied to `union` declarations.

10.3 `enum` declaration

10.3.1 `enum` formatting conventions

The `enum` declaration must look like this:

```
/** LED status constants */
enum led_status {
    LED_STATUS_OFF   = 0, /**< LED is off */
    LED_STATUS_ON    = 1, /**< LED is on */
    LED_STATUS_FAIL  = 2, /**< LED is unfunctional */
    LED_STATUS_MAINT = 4, /**< LED is now in maintenance mode; operations
                               are limited */
};
```

The same rules for typedef/variable definitions as for structures are applied.

Each `enum` component must occupy a separate line.

10.3.2 `enum` elements conventions

`enum` members must be prefixed with some abbreviation common for all members.

It is necessary to give a value for each member, if this value is specified externally (in manual, data sheet, technical specification, etc). Otherwise it is recommended not to assign values, or assign a value to the first element only.

10.4 Type qualifiers

Don't forget to give type qualifiers (`const` and `volatile`) when it is necessary, especially `volatile` specifier.

Absence of `volatile` specifier in place where it is required is a serious subtle error. Save your time and think every time when `volatile` specifier may be significant. Do not give `volatile` specifier to types, fields, variables which do not require it, because it may lead to misunderstanding and make your code worse.

It's *highly* recommended to give `const` specifier when it is applicable, because it may make compiler output better and help you to avoid some errors.

11 Variable declaration formatting

Each variable declaration must be started on a separate line. Global-space variables definitions must be started at column 0. Local variables must be started at block indent level, same as for the statements in this block. '`*`' specifiers must be placed near variable (or function name in forward declaration) name.

Variable declarations may be divided into blocks separated by empty line and block comment. The common formatting rules are applied inside one block.

```
/* Phase correction algorithm variables */
static fixed16  phase_error; /**< Currently estimated phase error */
static int      phcorr_num;  /**< Number of processed samples */
static fixed16 *sample;      /**< Next sample pointer */

static phase *phase_rotate(phase *cur, float angle);
```

Variable names in a block start at the same column. Variable comments start at the same column. If comment couldn't be fitted in one line, it may be split. Continuation line should begin at the same column where first letter of comments after `/*` was placed.

```
static uint32     input_bit_number;   /**< Number of bits received from
                                           external interface */
```

If variable block has only one declaration inside, variable comment may be omitted.

12 Literals and constants

Try not to use numeric constant values in code (except simple cases, like constants 0, 1, -1 and in some rare situations when direct constant usage may be more clear than identifier usage - for example bit masks like 0xff).

When multiline string constant is used, it is recommended to insert '`\n`' character explicitly and close quote at each line.

Examples:

```
/* Right: */
void
usage_information_print(void)
{
    printf("check -- check C source code to the C Source Code Style "
           "conformance\n"
           "Usage:\n"
           "    check <filename>\n");
}

/* Wrong: */
void
usage_information_print(void)
{
    printf("
check -- check C source code to the C Source Code Style conformance
Usage:
    check <filename>\n");
}
```

If constant length suffix is used, the capital suffix letter must be used (use `0L`, not `0l`). Similarly, for floating point constants, use capital exponent letter '`E`' (use `1.25E+35`, not `1.25e+35`).

In hexadecimal constants, lower-case letters must be used (use `0xdeadbeef`, not `0xDEADBEEF`).

13 C Expressions

Unary operators '`&`', '`*`', '`+`', '`-`', '`~`', '`!`', '`++`', '`--`' must be coupled with operands to which such operators are applied.

Binary operators '`*`', '`/`', '`%`', '`+`', '`-`', '`>>`', '`<<`', '`<`', '`>`', '`<=`', '`>=`', '`==`', '`!=`', '`&`', '`^`', '`|`', '`&&`', '`||`', '`=`', '`*=`', '`/=`', '`%=`', '`+=`', '`-=`', '`<<=`', '`>>=`', '`&=`', '`^=`', '`|=`' must be separated by one space at left and right.

Operators '`.`' and '`->`' must be coupled with both operands.

Operator '`,`' (comma) must be coupled to its left operand and separated by one space character from the right operand.

Operator '`[]`' must be coupled to the left and inner operands.

Syntactic components of conditional operator '`a ? b : c`' must be separated by one space character. It is recommended to enclose conditional operator into the parenthesis.

Grouping parenthesis must be coupled with the contents inside brackets.

In function call no delimiters must be added between function name or function expression and left round bracket. Also, no delimiters must be inserted between open round bracket and first operand and between close round bracket and between last operand.

'`sizeof`' operator must use syntax similar to function call.

Round brackets of type cast operator must be coupled with type specifier and with operand.

It is allowed to use embedded assignment statement if it make code simpler and better readable. Don't use embedded assignment in artificial places. For example,

```
x = x1 + w;
pos = ofs + x;
```

should not be replaced by

```
pos = ofs + (x = x1 + w);
```

More examples:

```
p = p->next;  /* Correct */
n = - n;      /* Incorrect */
++ b;         /* Incorrect */
b++;          /* Correct */
printf("Internal error: %d\\n", (class << 8) | errno);
scsi_freq = (freq_multiplier ? freq * 2 : freq);
list = hash_tab[hash_func(key) % hash_mod];
memcpy(dest, src, n);
```

It is recommended to insert additional unnecessary parenthesis to expression to make it more readable. It is recommended to assume only that unary operators have higher priority than binary and that multiplication operators has higher priority that addition.

It is allowed to insert additional spaces between binary operators, if it is improve readability. For example, if you have a group of statements which fill fields of structure, code with aligned assignment operators seems better. Sometimes, additional spaces may help to show structure of complex expressions.

Do not test non-boolean expression as you test a boolean. For example, where `x` is integer, use `if (x == 0)` instead `if (!x)`.

When you split expression into multiple lines, break line after operator. Line up continuation lines with the part of the preceding line they continue.

14 C Statements Formatting

14.1 `if` statement

`if` statement must look like this:

```
if (p != NULL)
{
    if (p->flag & FEL_INACTIVE)
        continue;
    else
        p = p->next;
}
```

It is recommended to enclose `if` branches into curly braces in each case, except it is a small simple statement. Particularly, it is recommended to enclose `if` branches when it is nested to other `if` or when this branch contains nested `if` statements. Also, it is recommended to enclose branch in curly braces if the other branch is enclosed.

`if - else if - else if...` statements must be formatted like this:

```
if (strcmp(reply, "yes") == 0)
{
    ...
}
else if (strcmp(reply, "no") == 0)
{
    ...
}
else
{
    ...
}
```

14.2 `for` statement

`for` statement may look like this:

```
for (i = 0; i < n; i++)
    x += buf[i];
```

Alternative form of `for` statement applied when components of `for` can not fit into the one 80-character line.

```
for (p = list->head, i = 0;
     p && !(element_is_invalid(p));
     p = list_element_next(p), i++)
{
    list_element_register(p);
}
```

It is recommended to enclose `for` body in curly braces in every case except when `for` components fit in one line and `for` body is sufficiently simple (something like `n += j;`, not a compound statement).

14.3 `while` statement

`while` statement must look like this:

```
while (p != NULL)
{
    interp_evaluate_node(p);
    p = p->next;
}
```

It is recommended to enclose `while` body in curly braces in every case except when `while` condition fits in one line and `while` body is sufficiently simple (something like `n += j;`, not a compound statement).

14.4 `do` statement

`do` statement must look like this:

```
do {
    entry_process(&n, entry);
} while (n > 0);
```

14.5 `switch` statement

`switch` statement must look like this:

```
switch (action)
{
    case ACTION_NOP:
    case ACTION_NEXT:
        break;

    case ACTION_COPY:
        *new_type = *type;
        state = STATE_COPIED;
        break;

    case ACTION_MOVE:
        *new_type = *type;
        type = NULL;
        break;

    case ACTION_CLONE:
    {
        int i;

        for (i = 0; i < type_num; i++)
        {
            clone_type[i] = *type;
        }
        state = STATE_CLONED;
        break;
    }

    default:
```

```
            assert(0);
    }
```

`switch` branches must be separated by empty lines. Each `case` label must be placed on a separate line. You must enclose statements in curly braces only if some local variables must be declared in block.

It is recommended to finish each `switch` branch with `break` statement. If you really want to continue execution in the next `switch` branch, insert comment `/*@fallthrough@*/` before empty line which separates two branches (it is used by *Splint* to annotate suspicious places in code).

The `default` case, if used, should be last and does not require a `break` if it is last. To make your code safer, it is recommended to add `default` branch always.

14.6 `goto` statement

It is *highly not recommended* to use `goto` statement in your program. In most cases it means that you code has poor design. Try to use "structural goto" statements, like `break`, `continue`, `return`. In some rare cases `goto` usage may be acceptable. For example, if function performs multiple resource allocations, and fail occurs, all allocated resources must be released. If fail condition may occur in many places, it is possible to use `goto` statement to jump to the resource deallocation fragment at the end of the function. If it is possible to avoid such `goto` usage (for example, if resource deallocation function exists and may be used) `goto` must not be used.

15 Preprocessor usage

15.1 Preprocessor directives formatting

Preprocessor directives must be started at column zero. It is not allowed to have spaces between '`#`' sign and directive name.

15.2 `#include` usage

Use '`<...>`' with file name in `#include` to include ANSI C headers or header files provided by the target operating system.

Use quotes '`"..."`' with file name to include other headers.

15.3 `#define` constant definitions

It is alternative to `enum` method to give symbolic names to constants. In some cases it is more convenient than `enum`, but `enum` is a preferred method for constant definitions (at least, because many debuggers couldn't use preprocessing information).

`#define` must be used for a constant definition when the constant is used in preprocessor expressions. Also, for example, hardware register definitions may be included in the assembler source, so it is better to use `#define` to define them.

15.4 Macros

It is required to around all macro parameters with parenthesis, because complex expressions can be used as macro parameters, and operator-precedence problems can arise. Body of function-like macros must be arounded with parenthesis too.

Lets consider the next code fragment:

```
#define DEBUG(s) \
    if (debug)                      \
    {                               \
        printf("debug: %s\n", s);  \
    }

    ...

    if (p == NULL)
        DEBUG("p is NULL");
    else
        ...
```

In this example, else branch of `if (p == NULL)` statement will become associated with `if` statement in `DEBUG` macro. To avoid this, all statement-like macros must be arounded by `do { ... } while (0)` statement:

```
#define DEBUG \
    do {                                \
        if (debug)                      \
        {                               \
            printf("debug: %s\n", s);  \
        }                               \
    } while (0)
```

Also, lets consider the following example:

```
#define INC(i) \
    do {                                                    \
        printf("Loop counter new value is %d\n", ++i);  \
    } while (0)

    ...

    for (i = 0; i < 5; ({INC(i);}))
    {
        ...
    }
```

The example states how macros arounded by `do { ... } while (0)` statement may be used in `for()` statement components. The macro call is arounded by curly and, then, round brackets.

It is recommended to hold macros on one line if it is possible. If no, format macros as it is shown in a bad example above. Line up all continuation backslashes on right side of macro body. Check that backslash is a last character in this string. Do not put any code in first line where macros name and parameters given. Follow to the indentation rules in body of macros.

Function-like macros must be commented like functions. Types of parameters should be stated in macro description.

It's recommented to use inline functions instead of function-like macros, if it is applicable. Compiler can help to avoid some errors and its output in the case of compilation errors more friendly, if inline function is used.

15.5 Conditional compilation

Conditional compilation is useful for making your code portable by handling different machines and OS behavior. Also, it is useful for debugging and for setting certain options at compile

time. Be careful: various controls may easily combine in unforeseen ways. Check for error condition combination or absence or incorrect value of some preprocessor variables. In last case, use preprocessor directive `#error` to handle such situations.

It is recommended to maintain syntactic correctness in false branches.

Try to put `#if`/`#ifdef` directives in header files instead C source file. Use `#if`/`#ifdef` to define macros that can be used uniformly in the code. For instance, a header file for checking memory allocation might look like:

```
#ifdef DEBUG
    extern void *mm_malloc(size_t bytes);
#define MALLOC(size) (mm_malloc(size))
#else
    extern void *malloc(size_t bytes);
#define MALLOC(size) (malloc(size))
#endif
```

It is recommended to use indentation rules for preprocessor conditions too.

Conditional compilation should generally be on a feature-by-feature basis. Although, one of usage of conditional preprocessor directives is to handle differences between different machines or operating systems. OS type, CPU kind, board, implementation etc. dependencies via conditional compilation should be avoided in most cases. Use name of the feature or property instead of name of bearing entity.

You may exclude fragment of your code using `#if 0` directive.

16 Safe programming

It is *highly* recommended to use `snprintf()` instead of `sprintf()`. If `sprintf()` is still used, it has to be guaranteed and explained in comment just before the usage why the buffer is sufficient.

17 Warning messages

Your program must be compiled without any warning messages for all target architectures. All warning messages must be enabled in compiler (`'-Wall -W'` flags enable all warnings in GCC).

Of course, error messages not acceptable too.

These conditions are applied to any code surrounded by conditional preprocessor statements, including traces, checks, debugging prints, assertions etc. The only exception is a temporary code fragment between `#if 0`/`#endif` statements. Such fragments must be removed before code freezing or releasing.

18 Project-Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. It is good and recommended practice to define on a per-project basis additional naming conventions, including common prefix conventions, rules for function or data structures grouping, restrictions to libraries and headers which may be used in project, preprocessor variables and macros usage, include files organization etc.

It is highly recommended to make those standards compliant to this one.