NAME

style — kernel source file style guide

DESCRIPTION

This file specifies the preferred style for kernel source files in the FreeBSD source tree. It is also a guide for the preferred userland code style. Many of the style rules are implicit in the examples. Be careful to check the examples before assuming that **style** is silent on an issue.

```
/*
 * Style guide for FreeBSD.  Based on the CSRG's KNF (Kernel Normal Form).
 *
 *      @(#)style      1.14 (Berkeley) 4/28/95
 * $FreeBSD$
 */

/*
 * VERY important single-line comments look like this.
 */

/* Most single-line comments look like this. */

/*
 * Multi-line comments look like this.  Make them real sentences.  Fill
 * them so they look like real paragraphs.
 */
```

The copyright header should be a multi-line comment, with the first line of the comment having a dash after the star like so:

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause-FreeBSD
 *
 * Copyright (c) 1984-2025 John Q. Public
 *
 * Long, boring license goes here, but trimmed for brevity
 */
```

An automatic script collects license information from the tree for all comments that start in the first column with "`/*-`". If you desire to flag `indent`(1) to not reformat a comment that starts in the first column which is not a license or copyright notice, change the dash to a star for those comments. Comments starting in columns other than the first are never considered license statements. Use the appropriate SPDX-License-Identifier line before the copyright. If the copyright assertion contains the phrase "`All Rights Reserved`" that should be on the same line as the word "`Copyright`". You should not insert a new copyright line between an old copyright line and this phrase. Instead, you should insert a new copyright phrase after a pre-existing "`All Rights Reserved`" line. When making changes, it is acceptable to fold an "`All Rights Reserved`" line with each of the "`Copyright`" lines. For files that have the "`All Rights Reserved`" line on the same line(s) as the word "`Copyright`", new copyright assertions should be added last. New "`Copyright`" lines should only be added when making substantial changes to the file, not for trivial changes.

After any copyright and license comment, there is a blank line, and the `$FreeBSD$` for non C/C++ language source files. Version control system ID tags should only exist once in a file (unlike in this one). Non-C/C++ source files follow the example above, while C/C++ source files follow the one below. All VCS (version control system) revision identification in files obtained from elsewhere should be maintained, including,

where applicable, multiple IDs showing a file's history. In general, do not edit foreign IDs or their infrastructure. Unless otherwise wrapped (such as "`#if defined(LIBC_SCCS)`"), enclose both in "`#if 0 ... #endif`" to hide any uncompilable bits and to keep the IDs out of object files. Only add "`From: `" in front of foreign VCS IDs if the file is renamed.

```
/* From: @(#)style     1.14 (Berkeley) 4/28/95 */

#include <sys/cdefs.h>
__FBSDID("$FreeBSD$");
```

Leave one blank line before the header files.

Kernel include files (`sys/*.h`) come first. If `<sys/cdefs.h>` is needed for **__FBSDID**(), include it first. If either `<sys/types.h>` or `<sys/param.h>` is needed, include it before other include files. (`<sys/param.h>` includes `<sys/types.h>`; do not include both.) Next, include `<sys/systm.h>`, if needed. The remaining kernel headers should be sorted alphabetically.

```
#include <sys/types.h> /* Non-local includes in angle brackets. */
#include <sys/systm.h>
#include <sys/endian.h>
#include <sys/lock.h>
#include <sys/queue.h>
```

For a network program, put the network include files next.

```
#include <net/if.h>
#include <net/if_dl.h>
#include <net/route.h>
#include <netinet/in.h>
#include <protocols/rwhod.h>
```

Do not include files from `/usr/include` in the kernel.

Leave a blank line before the next group, the `/usr/include` files, which should be sorted alphabetically by name.

```
#include <stdio.h>
```

Global pathnames are defined in `<paths.h>`. Pathnames local to the program go in "`pathnames.h`" in the local directory.

```
#include <paths.h>
```

Leave another blank line before the local include files.

```
#include "pathnames.h"        /* Local includes in double quotes. */
```

Do not **#define** or declare names in the implementation namespace except for implementing application interfaces.

The names of "unsafe" macros (ones that have side effects), and the names of macros for manifest constants, are all in uppercase. The expansions of expression-like macros are either a single token or have outer parentheses. Put a single tab character between the **#define** and the macro name. If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. Right-justify the backslashes; it makes it easier to read. If the macro encapsulates a compound statement, enclose it in a **do** loop, so that it can safely be used in **if** statements. Any final statement-terminating semicolon should be supplied by the macro invocation rather than the macro, to make parsing easier for pretty-printers and editors.

```
#define MACRO(x, y) do {                                        \
        variable = (x) + (y);                                   \
        (y) += 2;                                               \
} while (0)
```

When code is conditionally compiled using **#ifdef** or **#if**, a comment may be added following the matching **#endif** or **#else** to permit the reader to easily discern where conditionally compiled code regions end. This comment should be used only for (subjectively) long regions, regions greater than 20 lines, or where a series of nested **#ifdef 's** may be confusing to the reader. The comment should be separated from the **#endif** or **#else** by a single space. For short conditionally compiled regions, a closing comment should not be used.

The comment for **#endif** should match the expression used in the corresponding **#if** or **#ifdef**. The comment for **#else** and **#elif** should match the inverse of the expression(s) used in the preceding **#if** and/or **#elif** statements. In the comments, the subexpression "`defined(FOO)`" is abbreviated as "`FOO`". For the purposes of comments, "**#ifndef** `FOO`" is treated as "**#if** `!defined(FOO)`".

```
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

#ifdef COMPAT_43
/* A large region here, or other conditional code. */
#else /* !COMPAT_43 */
/* Or here. */
#endif /* COMPAT_43 */

#ifndef COMPAT_43
/* Yet another large region here, or other conditional code. */
#else /* COMPAT_43 */
/* Or here. */
#endif /* !COMPAT_43 */
```

The project prefers the use of ISO/IEC 9899:1999 ("ISO C99") unsigned integer identifiers of the form *uintXX_t* rather than the older BSD-style integer identifiers of the form *u_intXX_t*. New code should use the former, and old code should be converted to the new form if other major work is being done in that area and there is no overriding reason to prefer the older BSD-style. Like white-space commits, care should be taken in making *uintXX_t* only commits.

Similarly, the project prefers the use of ISO C99 *bool* rather than the older *int* or *boolean_t*. New code should use *bool*, and old code may be converted if it is reasonable to do so. Literal values are named `true` and `false`. These are preferred to the old spellings `TRUE` and `FALSE`. Userspace code should include `<stdbool.h>`, while kernel code should include `<sys/types.h>`.

Likewise, the project prefers ISO C99 designated initializers when it makes sense to do so.

Enumeration values are all uppercase.

```
enum enumtype { ONE, TWO } et;
```

The use of internal_underscores in identifiers is preferred over camelCase or TitleCase.

In declarations, do not put any whitespace between asterisks and adjacent tokens, except for tokens that are identifiers related to types. (These identifiers are the names of basic types, type qualifiers, and **typedef**-names other than the one being declared.) Separate these identifiers from asterisks using a single space.

When declaring variables in structures, declare them sorted by use, then by size (largest to smallest), and then in alphabetical order. The first category normally does not apply, but there are exceptions. Each one gets its own line. Try to make the structure readable by aligning the member names using either one or two tabs depending upon your judgment. You should use one tab only if it suffices to align at least 90% of the member names. Names following extremely long types should be separated by a single space.

Major structures should be declared at the top of the file in which they are used, or in separate header files if they are used in multiple source files. Use of the structures should be by separate declarations and should be **extern** if they are declared in a header file.

```
struct foo {
        struct foo     *next;          /* List of active foo. */
        struct mumble  amumble;        /* Comment for mumble. */
        int            bar;            /* Try to align the comments. */
        struct verylongtypename *baz;  /* Does not fit in 2 tabs. */
};
struct foo *foohead;                   /* Head of global foo list. */
```

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```
#include <sys/queue.h>

struct foo {
        LIST_ENTRY(foo)        link;           /* Use queue macros for foo lists. */
        struct mumble  amumble;        /* Comment for mumble. */
        int            bar;            /* Try to align the comments. */
        struct verylongtypename *baz;  /* Does not fit in 2 tabs. */
};
LIST_HEAD(, foo) foohead;              /* Head of global foo list. */
```

Avoid using typedefs for structure types. Typedefs are problematic because they do not properly hide their underlying type; for example you need to know if the typedef is the structure itself or a pointer to the structure. In addition they must be declared exactly once, whereas an incomplete structure type can be mentioned as many times as necessary. Typedefs are difficult to use in stand-alone header files: the header that defines the typedef must be included before the header that uses it, or by the header that uses it (which causes namespace pollution), or there must be a back-door mechanism for obtaining the typedef.

When convention requires a **typedef**, make its name match the struct tag. Avoid typedefs ending in "_t", except as specified in Standard C or by POSIX.

```
/* Make the structure name match the typedef. */
typedef struct bar {
        int     level;
} BAR;
typedef int            foo;            /* This is foo. */
typedef const long     baz;            /* This is baz. */
```

All functions are prototyped somewhere.

Function prototypes for private functions (i.e., functions not used elsewhere) go at the top of the first source module. Functions local to one source module should be declared **static**.

Functions used from other parts of the kernel are prototyped in the relevant include file. Function prototypes should be listed in a logical order, preferably alphabetical unless there is a compelling reason to use a different ordering.

Functions that are used locally in more than one module go into a separate header file, e.g., "`extern.h`".

Do not use the `__P` macro.

In general code can be considered "new code" when it makes up about 50% or more of the file(s) involved. This is enough to break precedents in the existing code and use the current **`style`** guidelines.

The kernel has a name associated with parameter types, e.g., in the kernel use:

```
void    function(int fd);
```

In header files visible to userland applications, prototypes that are visible must use either "protected" names (ones beginning with an underscore) or no names with the types. It is preferable to use protected names. E.g., use:

```
void    function(int);
```

or:

```
void    function(int _fd);
```

Prototypes may have an extra space after a tab to enable function names to line up:

```
static char    *function(int _arg, const char *_arg2, struct foo *_arg3,
                   struct bar *_arg4);
static void     usage(void);

/*
 * All major routines should have a comment briefly describing what
 * they do.  The comment before the "main" routine should describe
 * what the program does.
 */
int
main(int argc, char *argv[])
{
        char *ep;
        long num;
        int ch;
```

For consistency, `getopt`(3) should be used to parse options. Options should be sorted in the `getopt`(3) call and the **`switch`** statement, unless parts of the **`switch`** cascade. Elements in a **`switch`** statement that cascade should have a `FALLTHROUGH` comment. Numerical arguments should be checked for accuracy. Code which is unreachable for non-obvious reasons may be marked /* `NOTREACHED` */.

```
        while ((ch = getopt(argc, argv, "abNn:")) != -1)
                switch (ch) {          /* Indent the switch. */
                case 'a':              /* Do not indent the case. */
                        aflag = 1;     /* Indent case body one tab. */
                        /* FALLTHROUGH */
                case 'b':
                        bflag = 1;
                        break;
                case 'N':
                        Nflag = 1;
                        break;
                case 'n':
                        num = strtol(optarg, &ep, 10);
                        if (num <= 0 || *ep != '\0') {
```

```
                                warnx("illegal number, -n argument -- %s",
                                    optarg);
                                usage();
                        }
                        break;
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;
```

Space after keywords (**if**, **while**, **for**, **return**, **switch**). Two styles of braces ({ and '}') are allowed for single line statements. Either they are used for all single statements, or they are used only where needed for clarity. Usage within a function should be consistent. Forever loops are done with **for**'s, not **while**'s.

```
        for (p = buf; *p != '\0'; ++p)
                ;       /* nothing */
        for (;;)
                stmt;
        for (;;) {
                z = a + really + long + statement + that + needs +
                    two + lines + gets + indented + four + spaces +
                    on + the + second + and + subsequent + lines;
        }
        for (;;) {
                if (cond)
                        stmt;
        }
        if (val != NULL)
                val = realloc(val, newsize);
```

Parts of a **for** loop may be left empty.

```
        for (; cnt < 15; cnt++) {
                stmt1;
                stmt2;
        }
```

A **for** loop may declare and initialize its counting variable.

```
        for (int i = 0; i < 15; i++) {
                stmt1;
        }
```

Indentation is an 8 character tab. Second level indents are four spaces. If you have to wrap a long statement, put the operator at the end of the line.

```
        while (cnt < 20 && this_variable_name_is_too_long &&
            ep != NULL)
                z = a + really + long + statement + that + needs +
                    two + lines + gets + indented + four + spaces +
                    on + the + second + and + subsequent + lines;
```

Do not add whitespace at the end of a line, and only use tabs followed by spaces to form the indentation. Do not use more spaces than a tab will produce and do not use spaces in front of tabs.

Closing and opening braces go on the same line as the **else**. Braces that are not necessary may be left out.

```
if (test)
        stmt;
else if (bar) {
        stmt;
        stmt;
} else
        stmt;
```

No spaces after function names. Commas have a space after them. No spaces after '(' or '[' or preceding ']' or ')' characters.

```
error = function(a1, a2);
if (error != 0)
        exit(error);
```

Unary operators do not require spaces, binary operators do. Do not use parentheses unless they are required for precedence or unless the statement is confusing without them. Remember that other people may confuse easier than you. Do YOU understand the following?

```
a = b->c[0] + ~d == (e || f) || g && h ? i : j >> 1;
k = !(l & FLAGS);
```

Exits should be 0 on success, or 1 on failure.

```
exit(0);        /*
                 * Avoid obvious comments such as
                 * "Exit 0 on success."
                 */
}
```

The function type should be on a line by itself preceding the function. The opening brace of the function body should be on a line by itself.

```
static char *
function(int a1, int a2, float fl, int a4, struct bar *bar)
{
```

When declaring variables in functions declare them sorted by size, then in alphabetical order; multiple ones per line are okay. If a line overflows reuse the type keyword. Variables may be initialized where declared especially when they are constant for the rest of the scope. Declarations may be placed before executable lines at the start of any block. Calls to complicated functions should be avoided when initializing variables.

```
struct foo one, *two;
struct baz *three = bar_get_baz(bar);
double four;
int *five, six;
char *seven, eight, nine, ten, eleven, twelve;

four = my_complicated_function(a1, f1, a4);
```

Do not declare functions inside other functions; ANSI C says that such declarations have file scope regardless of the nesting of the declaration. Hiding file declarations in what appears to be a local scope is undesirable and will elicit complaints from a good compiler.

Casts and **sizeof**'s are not followed by a space. Note that indent(1) does not understand this rule. **sizeof**'s are written with parenthesis always. The redundant parenthesis rules do not apply to **sizeof**(*var*) instances.

NULL is the preferred null pointer constant. Use NULL instead of (*type* *)0 or (*type* *)NULL in contexts where the compiler knows the type, e.g., in assignments. Use (*type* *)NULL in other contexts, in particular for all function args. (Casting is essential for variadic args and is necessary for other args if the function prototype might not be in scope.) Test pointers against NULL, e.g., use:

```
(p = f()) == NULL
```

not:

```
!(p = f())
```

Do not use **!** for tests unless it is a boolean, e.g., use:

```
if (*p == '\0')
```

not:

```
if (!*p)
```

Routines returning *void* * should not have their return values cast to any pointer type.

Values in **return** statements should be enclosed in parentheses.

Use err(3) or warn(3), do not roll your own.

```
        if ((four = malloc(sizeof(struct foo))) == NULL)
                err(1, (char *)NULL);
        if ((six = (int *)overflow()) == NULL)
                errx(1, "number overflowed");
        return (eight);
}
```

When converting K&R style declarations to ANSI style, preserve any comments about parameters.

Long parameter lists are wrapped with a normal four space indent.

Variable numbers of arguments should look like this:

```
#include <stdarg.h>

void
vaf(const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        STUFF;
        va_end(ap);
        /* No return needed for void functions. */
}

static void
usage(void)
{
        /* Optional blank line goes here. */
```

Optionally, insert a blank line at the beginning of functions with no local variables. Older versions of this **style** document required the blank line convention, so it is widely used in existing code.

Do not insert a blank line at the beginning of functions with local variables. Instead, these should have local variable declarations first, followed by one blank line, followed by the first statement.

Use `printf`(3), not `fputs`(3), `puts`(3), `putchar`(3), whatever; it is faster and usually cleaner, not to mention avoiding stupid bugs.

Usage statements should look like the manual pages **SYNOPSIS**. The usage statement should be structured in the following order:

1. Options without operands come first, in alphabetical order, inside a single set of brackets (`[` and '`]`').
2. Options with operands come next, also in alphabetical order, with each option and its argument inside its own pair of brackets.
3. Required arguments (if any) are next, listed in the order they should be specified on the command line.
4. Finally, any optional arguments should be listed, listed in the order they should be specified, and all inside brackets.

A bar ('`|`') separates "either-or" options/arguments, and multiple options/arguments which are specified together are placed in a single set of brackets.

```
    "usage: f [-aDde] [-b b_arg] [-m m_arg] req1 req2 [opt1 [opt2]]\n"
    "usage: f [-a | -b] [-c [-dEe] [-n number]]\n"

        (void)fprintf(stderr, "usage: f [-ab]\n");
        exit(1);
}
```

Note that the manual page options description should list the options in pure alphabetical order. That is, without regard to whether an option takes arguments or not. The alphabetical ordering should take into account the case ordering shown above.

New core kernel code should be reasonably compliant with the **style** guides. The guidelines for third-party maintained modules and device drivers are more relaxed but at a minimum should be internally consistent with their style.

Stylistic changes (including whitespace changes) are hard on the source repository and are to be avoided without good reason. Code that is approximately FreeBSD KNF **style** compliant in the repository must not diverge from compliance.

Whenever possible, code should be run through a code checker (e.g., various static analyzers or **cc -Wall**) and produce minimal warnings.

New code should use **_Static_assert**() instead of the older **CTASSERT**().

FILES

`/usr/src/tools/tools/editing/freebsd.el`
: An Emacs plugin to follow the FreeBSD **style** indentation rules.

`/usr/src/tools/tools/editing/freebsd.vim`
: A Vim plugin to follow the FreeBSD **style** indentation rules.

SEE ALSO

`indent`(1), `err`(3), `warn`(3), `style.Makefile`(5), `style.mdoc`(5), `style.lua`(9)

HISTORY

This manual page is largely based on the `src/admin/style/style` file from the 4.4BSD–Lite2 release, with occasional updates to reflect the current practice and desire of the FreeBSD project. `src/admin/style/style` is a codification by the CSRG of the programming style of Ken Thompson and Dennis Ritchie in Version 6 AT&T UNIX.