NAME

style — Kernel source file style guide (KNF)

DESCRIPTION

This file specifies the preferred style for kernel source files in the OpenBSD source tree. It is also a guide for preferred userspace code style. These guidelines should be followed for all new code. 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.

```
/*
 * Style guide for the OpenBSD KNF (Kernel Normal Form).
 */

/*
 * 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.
 */
```

Kernel include files (i.e., <sys/*.h>) come first; normally, you'll need <sys/types.h> OR <sys/param.h>, but not both! <sys/types.h> includes <sys/cdefs.h>, and it's okay to depend on that.

```
#include <sys/types.h> /* Non-local includes in brackets. */
```

If it's 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>
```

Then there's a blank line, followed by the /usr/include files. The /usr/include files, for the most part, should be sorted.

Global pathnames are defined in /usr/include/paths.h. Pathnames local to the program go in pathnames.h in the local directory.

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

Then there's a blank line, and the user include files.

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

All non-static functions are prototyped somewhere.

Function prototypes for private functions (i.e., functions not used elsewhere) go at the top of the first source module. In the kernel, private functions do not require a prototype as long as they are defined before they are used. In userspace, functions local to one source module should be declaredstatic. This should not be done in the kernel since it makes it impossible to use the kernel debugger.

Functions used from other files are prototyped in the relevant include file.

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

Prototypes should not have variable names associated with the types; i.e.,

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

not:

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

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

```
static char    *function(int, const char *);
static void     usage(void);
```

There should be no space between the function name and the argument list.

Use `__dead` from <`sys/cdefs.h`> for functions that don't return, i.e.,

```
__dead void    abort(void);
```

In header files, put function prototypes within `__BEGIN_DECLS / __END_DECLS` matching pairs. This makes the header file usable from C++.

Macros are capitalized and parenthesized, and should avoid side-effects. If they are an inline expansion of a function, the function is defined all in lowercase; the macro has the same name all in uppercase. If the macro needs more than a single line, use braces. Right-justify the backslashes, as the resulting definition is 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)
```

If a macro with arguments declares local variables, those variables should use identifiers beginning with two underscores. This is required for macros implementing C and POSIX interfaces and recommended for all macros for consistency.

Enumeration values are all uppercase.

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

When defining unsigned integers use "unsigned int" rather than just "unsigned"; the latter has been a source of confusion in the past.

When declaring variables in structures, declare them sorted by use, then by size (largest to smallest), then by alphabetical order. The first category normally doesn't apply, but there are exceptions. Each one gets its own line. Put a tab after the first word, i.e., use `int^Ix;` and `struct^Ifoo *x;`.

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;
};
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;   /* Queue macro glue for foo lists */
        struct  mumble amumble;         /* Comment for mumble */
        int     bar;
};
LIST_HEAD(, foo) foohead;       /* Head of global foo list */
```

Avoid using typedefs for structure types. This makes it impossible for applications to use pointers to such a structure opaquely, which is both possible and beneficial when using an ordinary struct tag. 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.

```
/*
 * 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[])
{
        int aflag, bflag, ch, num;
        const char *errstr;
```

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.

```
while ((ch = getopt(argc, argv, "abn:")) != -1) {
        switch (ch) {           /* Indent the switch. */
        case 'a':               /* Don't indent the case. */
                aflag = 1;
                /* FALLTHROUGH */
        case 'b':
                bflag = 1;
                break;
        case 'n':
                num = strtonum(optarg, 0, INT_MAX, &errstr);
                if (errstr) {
                        warnx("number is %s: %s", errstr, optarg);
                        usage();
                }
                break;
        default:
                usage();
        }
}
argc -= optind;
argv += optind;
```

Use a space after keywords (`if`, `while`, `for`, `return`, `switch`). No braces are used for control statements with zero or only a single statement unless that statement is more than a single line, in which case they are permitted.

```
for (p = buf; *p != '\0'; ++p)
        continue;
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;
}
```

Parts of a for loop may be left empty.

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

Indentation is an 8 character tab. Second level indents are four spaces. All code should fit in 80 columns.

```
while (cnt < 20)
        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 aren't necessary may be left out, unless they cause a compiler warning.

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

Do not use spaces after function names. Commas have a space after them. Do not use spaces after '(' or '[' or preceding ']' or ')' characters.

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

Unary operators don't require spaces; binary operators do. Don't use parentheses unless they're required for precedence, the statement is confusing without them, or the compiler generates a warning without them. Remember that other people may be confused more easily 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 non-zero for errors.

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

The function type should be on a line by itself preceding the function.

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

When declaring variables in functions, declare them sorted by size (largest to smallest), then in alphabetical order; multiple ones per line are okay. If a line overflows, reuse the type keyword.

Be careful not to obfuscate the code by initializing variables in the declarations. Use this feature only thoughtfully. DO NOT use function calls in initializers!

```
struct foo one, *two;
double three;
int *four, five;
char *six, seven, eight, nine, ten, eleven, twelve;

four = myfunction();
```

Do not declare functions inside other functions.

Casts and **sizeof**() calls are not followed by a space. Note that `indent`(1) does not understand this rule.

Use of the "register" specifier is discouraged in new code. Optimizing compilers such as gcc can generally do a better job of choosing which variables to place in registers to improve code performance. The exception to this is in functions containing assembly code where the "register" specifier is required for proper code generation in the absence of compiler optimization.

When using **longjmp**() or **vfork**() in a program, the **-W** or **-Wall** flag should be used to verify that the compiler does not generate warnings such as

```
warning: variable 'foo' might be clobbered by 'longjmp' or 'vfork'.
```

If any warnings of this type occur, you must apply the "volatile" type-qualifier to the variable in question. Failure to do so may result in improper code generation when optimization is enabled. Note that for pointers, the location of "volatile" specifies if the type-qualifier applies to the pointer, or the thing being pointed to. A volatile pointer is declared with "volatile" to the right of the "*". Example:

```
char *volatile foo;
```

says that "foo" is volatile, but "*foo" is not. To make "*foo" volatile use the syntax

```
volatile char *foo;
```

If both the pointer and the thing pointed to are volatile use

```
volatile char *volatile foo;
```

"const" is also a type-qualifier and the same rules apply. The description of a read-only hardware register might look something like:

```
const volatile char *reg;
```

Global flags set inside signal handlers should be of type "volatile sig_atomic_t" if possible. This guarantees that the variable may be accessed as an atomic entity, even when a signal has been delivered. Global variables of other types (such as structures) are not guaranteed to have consistent values when accessed via a signal handler.

`NULL` is the preferred null pointer constant. Use `NULL` instead of (type *)0 or (type*)NULL in all cases e xcept for arguments to variadic functions where the compiler does not know the type.

Don't use '`!`' for tests unless it's a boolean, i.e., use

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

not

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

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

Use the `err`(3) and `warn`(3) family of functions. Don't roll your own!

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

Always use ANSI function definitions. 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)
{
```

Usage statements should take the same form as the synopsis in manual pages. Options without operands come first, in alphabetical order inside a single set of braces, followed by options with operands, in alphabetical order, each in braces, followed by required arguments in the order they are specified, followed by optional arguments in the order they are specified.

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

If numbers are used as options, they should be placed first, as shown in the example below. Uppercase letters take precedence over lowercase.

```
"usage: f [-12aDde] [-b b_arg] [-m m_arg] req1 req2 [opt1 [opt2]]\n"
"usage: f [-a | -b] [-c [-de] [-n number]]\n"
```

The `getprogname`(3) function may be used instead of hard-coding the program name.

```
fprintf(stderr, "usage: %s [-ab]\n", getprogname());
exit(1);
```

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.

Whenever possible, code should be run through a code checker (e.g., "`gcc -Wall -W -Wpointer-arith -Wbad-function-cast ...`" or splint from the ports tree) and produce minimal warnings. Since lint has been removed, the only lint-style comment that should be used is FALLTHROUGH, as it's useful to humans. Other lint-style comments such as ARGSUSED, LINTED, and NOTREACHED may be deleted.

Note that documentation follows its own style guide, as documented in `mdoc`(7).

FILES

`/usr/share/misc/license.template` Example license for new code.

SEE ALSO

`indent`(1), `err`(3), `queue`(3), `warn`(3), `mdoc`(7)

HISTORY

This man page is largely based on the src/admin/style/style file from the 4.4BSD−Lite2 release, with updates to reflect the current practice and desire of the OpenBSD project.