

VCN Bots coding style

(Copy-pasted and mainly based on Linux Kernel coding style)

(This document is under development. Date: 29/01/2016)

Contents

1	Introduction	3
2	Why is written in C (Standard 2011)?	3
3	Defensive programming	3
4	Quasi-deprecated statements	4
5	Indentation	4
6	Breaking long lines and strings	6
7	Placing Braces and Spaces	6
7.1	Spaces	8
8	Naming	9
9	Typedefs	10
10	Functions	12
11	Centralized exiting of functions	13
12	Commenting	14
13	You've made a mess of it	15
14	Data structures	16
15	Macros and Enums	17
16	Printing messages	18
17	Allocating memory	19
18	The inline disease	20
19	Function return values and names	20

20 Don't re-invent the containers	21
21 Editor modelines and other cruft	21
22 Conditional Compilation	22
Appendix I: References	23

1 Introduction

This is a short document describing the preferred coding style for VCN Bots. Coding style is very personal, and we won't `_force_` our views on anybody, but this is what goes for anything that we have to be able to maintain, and we'd prefer it for most other things too. Please at least consider the points made here.

First off, we'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

2 Why is written in C (Standard 2011)?

For many reasons. The main one is compability with all systems, arquitectures and third-party APIs, such as MPI, OpenMP, Posix Threads, OpenCL, Cuda-C, Java NDK, Python-C (Cython).

3 Defensive programming

The few basic rules of defensive programming are explained in a short chapter in Steve McConnell's classic book on programming, *Code Complete*:

- Protect your code from invalid data coming from outside, wherever you decide outside is. Data from an external system or the user or a file, or any data from outside of the module/component. Establish barricades or safe zones or trust boundaries everything outside of the boundary is dangerous, everything inside of the boundary is safe. In the barricade code, validate all input data: check all input parameters for the correct type, length, and range of values. Double check for limits and bounds.
- After you have checked for bad data, decide how to handle it. Defensive Programming is NOT about swallowing errors or hiding bugs. Its about deciding on the trade-off between robustness (keep running if there is a problem you can deal with) and correctness (never return inaccurate results). Choose a strategy to deal with bad data: return an error and stop right away (fast fail), return a neutral value, substitute data values, ...Make sure that the strategy is clear and consistent.

- Don't assume that a function call or method call outside of your code will work as advertised. Make sure that you understand and test error handling around external APIs and libraries.
- Use assertions to document assumptions and to highlight impossible conditions, at least in development and testing. This is especially important in large systems that have been maintained by different people over time, or in high-reliability code.
- Add diagnostic code, logging and tracing intelligently to help explain what's going on at run-time, especially if you run into a problem.
- Standardize error handling. Decide how to handle normal errors or expected errors and warnings, and do all of this consistently.
- Use exception handling only when you need to, and make sure that you understand the VCN exception handler inside out.

Paranoid style:

```
/* Bad */
if (b == NULL) {}
/* Good */
if (NULL == bad) {}
```

4 Quasi-deprecated statements

”register” key word is deprecated, because most compilers ignore it, and store in register the appropriate variables.

5 Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

The preferred way to ease multiple indentation levels in a switch statement is to align the "switch" and its subordinate "case" labels in the same column instead of "double-indenting" the "case" labels. E.g.:

```
switch (suffix) {
case 'G':
case 'g':
mem <=&= 30;
break;
case 'M':
case 'm':
mem <=&= 20;
break;
case 'K':
case 'k':
mem <=&= 10;
/* fall through */
default:
break;
}
```

Don't put multiple statements on a single line unless you have something to hide:

```
if (condition) do_this;
    do_something_everytime;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.

Get a decent editor and don't leave whitespace at the end of lines.

6 Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks, unless exceeding 80 columns significantly increases readability and does not hide information. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. However, never break user-visible strings such as `printf` messages, because that breaks the ability to `grep` for them.

7 Placing Braces and Spaces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {  
we do y  
}
```

This applies to all non-function statement blocks (`if`, `switch`, `for`, `while`, `do`). E.g.:

```
switch (action) {  
case KOBJ_ADD:  
return "add";  
case KOBJ_REMOVE:  
return "remove";  
case KOBJ_CHANGE:  
return "change";  
default:  
return NULL;  
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are right and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the same statement, ie a "while" in a do-statement or an "else" in an if-statement, like this:

```
do {
body of do-loop
} while (condition);
```

and

```
if (x == y) {
..
} else if (x > y) {
...
} else {
....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
action();
```


and

```
if (condition)
do_this();
else
do_that();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

```
if (condition) {
do_this();
do_that();
} else {
otherwise();
}
```

7.1 Spaces

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are `sizeof`, `typeof`, `alignof`, and `__attribute__`, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "sizeof info" after "struct fileinfo info;" is declared).

So use a space after these keywords:

```
if, switch, case, for, do, while
```

but not with `sizeof`, `typeof`, `alignof`, or `__attribute__`. E.g.,

```
s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is **bad**:

```
s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of '*' is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
= + - < > * / % | & ^ <= >= == != ? :
```

but no space after unary operators:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

no space before the postfix increment & decrement unary operators:

```
++ --
```

no space after the prefix increment & decrement unary operators:

```
++ --
```

and no space around the `'.'` and `"->"` structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.

8 Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like `ThisVariableIsATemporaryCounter`. A C programmer would call that variable

"tmp", which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function "foo" is a shooting offense.

GLOBAL variables (to be used only if you *really* need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that "count_active_users()" or similar, you should *not* call it "cntusr()".

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called "i". Calling it "loop_counter" is non-productive, if there is no chance of it being misunderstood. Similarly, "tmp" can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See chapter 6 (Functions).

9 Typedefs

Please don't use things like "vps_t". It's a *mistake* to use typedef for structures and pointers. When you see a

```
vps_t a;
```

in the source, what does it mean? In contrast, if it says

```
struct virtual_container *a;
```

you can actually tell what "a" is.

Lots of people think that typedefs "help readability". Not so. They are useful only for:

- totally opaque objects (where the typedef is actively used to `_hide_` what the object is).

Example: `"pte_t"` etc. opaque objects that you can only access using the proper accessor functions.

NOTE! Opaqueness and "accessor functions" are not good in themselves. The reason we have them for things like `pte_t` etc. is that there really is absolutely `_zero_` portably accessible information there.

- Clear integer types, where the abstraction `_helps_` avoid confusion whether it is "int" or "long".

`u8/u16/u32` are perfectly fine typedefs, although they fit into category (d) better than here.

NOTE! Again - there needs to be a `_reason_` for this. If something is "unsigned long", then there's no reason to do

```
typedef unsigned long myflags_t;
```

but if there is a clear reason for why it under certain circumstances might be an "unsigned int" and under other configurations might be "unsigned long", then by all means go ahead and use a typedef.

- when you use `sparse` to literally create a `_new_` type for type-checking.
- New types which are identical to standard C99 types, in certain exceptional circumstances.

Although it would only take a short amount of time for the eyes and brain to become accustomed to the standard types like `'uint32_t'`, some people object to their use anyway.

Therefore, the Linux-specific `'u8/u16/u32/u64'` types and their signed equivalents which are identical to standard types are permitted – although they are not mandatory in new code of your own.

When editing existing code which already uses one or the other set of types, you should conform to the existing choices in that code.

- Types safe for use in userspace.

In certain structures which are visible to userspace, we cannot require C99 types and cannot use the `'u32'` form above. Thus, we use `__u32` and similar types in all structures which are shared with userspace.

Maybe there are other cases too, but the rule should basically be to NEVER EVER use a typedef unless you can clearly match one of those rules.

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should `_never_` be a typedef.

10 Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

In source files, separate functions with one blank line. If the function is exported, the `EXPORT*` macro for it should follow immediately after the closing function brace line. E.g.:

```
int system_is_up(void)
{
return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.

11 Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done. If there is no cleanup needed then just return directly.

Choose label names which say what the goto does or why the goto exists. An example of a good name could be "out_buffer:" if the goto frees "buffer". Avoid using GW-BASIC names like "err1:" and "err2:". Also don't name them after the goto location like "err_malloc_failed:"

The rationale for using `gotos` is:

- Unconditional statements are easier to understand and follow.
- Nesting is reduced.
- Errors by not updating individual exit points when making modifications are prevented.
- Saves the compiler work to optimize redundant code away.

```
int fun(int a)
{
    int result = 0;
    char *buffer;

    buffer = vcn_malloc(SIZE, GFP_KERNEL);
    if (!buffer)
        return -ENOMEM;

    if (condition1) {
        while (loop1) {
            ...
        }
    }
```

```

result = 1;
goto out\_buffer;
}
...
out\_buffer:
vcn_free(buffer);
return result;
}

```

A common type of bug to be aware of it "one err bugs" which look like this:

```

err:
vcn_free(foo->bar);
vcn_free(foo);
return ret;

```

The bug in this code is that on some exit paths "foo" is NULL. Normally the fix for this is to split it up into two error labels "err_bar:" and "err_foo:".

12 Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the `_working_` is obvious, and it's a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 6 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

When commenting the kernel API functions, please use the kernel-doc format. See the files `Documentation/kernel-doc-nano-HOWTO.txt` and `scripts/kernel-doc` for details.

Linux style for comments is the C89 `"/ * ... */` style. Don't use C99-style `// ...` comments.

The preferred style for long (multi-line) comments is:

```

/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description:  A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */

```

For files in `net/` and `drivers/net/` the preferred style for long (multi-line) comments is a little different.

```

/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */

```

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.

13 You've made a mess of it

That's OK, we all do. You've probably been told by your long-time Unix user helper that "GNU emacs" automatically formats the C sources for you, and you've noticed that yes, it does do that, but the defaults it uses are less than desirable (in fact, they are worse than random typing - an infinite number of monkeys typing into GNU emacs would never make a good program).

So, you can either get rid of GNU emacs, or change it to use saner values. To do the latter, you can stick the following in your `.emacs` file:

```

(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by
  tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
        (column (c-langelem-2nd-pos c-syntactic-element))
        (offset (- (1+ column) anchor))
        (steps (floor offset c-basic-offset)))
    (* (max steps 1) c-basic-offset)))

```



```
(add-hook 'c-mode-common-hook (lambda () ;; Add kernel style (c-add-  
style "linux-tabs-only" '("linux" (c-offsets-alist (arglist-cont-nonempty c-lineup-  
gcc-asm-reg c-lineup-arglist-tabs-only))))))
```

```
(add-hook 'c-mode-hook (lambda () (let ((filename (buffer-file-name))) ;;  
Enable kernel mode for the appropriate files (when (and filename (string-  
match (expand-file-name " /src/linux-trees") filename)) (setq indent-tabs-  
mode t) (setq show-trailing-whitespace t) (c-set-style "linux-tabs-only")))))
```

This will make emacs go better with the coding style for C.

But even if you fail in getting emacs to do sane formatting, not everything is lost: use "indent".

Now, again, GNU indent has the same brain-dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that's not too bad, because even the makers of GNU indent recognize the authority of K&R (the GNU people aren't evil, they are just severely misguided in this matter), so you just give indent the options "-kr -i8" (stands for "K&R, 8 character indents"), or use "scripts/Lindent", which indents in the latest style.

"indent" has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the man page. But remember: "indent" is not a fix for bad programming.

14 Data structures

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely *have* to reference count all your uses.

Reference counting means that you can avoid locking, and allows multiple users to have access to the data structure in parallel - and not having to worry about the structure suddenly going away from under them just because they slept or did something else for a while.

Note that locking is *not* a replacement for reference counting. Locking is used to keep data structures coherent, while reference counting is a memory management technique. Usually both are needed, and they are not to be confused with each other.

Many data structures can indeed have two levels of reference counting,

when there are users of different "classes". The subclass count counts the number of subclass users, and decrements the global count just once when the subclass count goes to zero.

Examples of this kind of "multi-level-reference-counting" can be found in memory management ("struct mm_struct": mm_users and mm_count), and in filesystem code ("struct super_block": s_count and s_active).

Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.

15 Macros and Enums

Names of macros defining constants and labels in enums are capitalized.

```
#define CONSTANT 0x12345
```

Enums are preferred when defining several related constants.

CAPITALIZED macro names are appreciated but macros resembling functions may be named in lower case.

Generally, inline functions are preferable to macros resembling functions.

Macros with multiple statements should be enclosed in a do - while block:

```
#define macrofun(a, b, c) \
do { \
if (a == 5) \
do\_this(b, c); \
} while (0)
```

Things to avoid when using macros:

1. macros that affect control flow:

```
#define FOO(x) \
do { \
if (blah(x) < 0) \
return -EBUGGERED; \
} while(0)
```

is a `_very_` bad idea. It looks like a function call but exits the "calling" function; don't break the internal parsers of those who will read the code.

2. macros that depend on having a local variable with a magic name:

```
#define FOO(val) bar(index, val)
```

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.

3. macros with arguments that are used as l-values: `FOO(x) = y;` will bite you if somebody e.g. turns `FOO` into an inline function.
4. forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

5. namespace collisions when defining local variables in macros resembling functions:

```
#define FOO(x) \
({ \
    typeof(x) ret; \
    ret = calc\_ret(x); \
    (ret); \
})
```

`ret` is a common name for a local variable - `__foo_ret` is less likely to collide with an existing variable.

16 Printing messages

VCN Bots developers like to be seen as literate. Do mind the spelling of messages to make a good impression. Do not use crippled words like "dont";

use "do not" or "don't" instead. Make the messages concise, clear, and unambiguous.

The messages do not have to be terminated with a period.

Printing numbers in parentheses (%d) adds no value and should be avoided.

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. However debug message printing is handled differently than printing other non-debug messages. While the other `pr_XXX()` functions print unconditionally, `pr_debug()` does not; it is compiled out by default, unless either `DEBUG` is defined or `CONFIG_DYNAMIC_DEBUG` is set. That is true for `dev_dbg()` also, and a related convention uses `VERBOSE_DEBUG` to add `dev_vdbg()` messages to the ones already enabled by `DEBUG`.

17 Allocating memory

VCN Bots provides the following general purpose memory allocators: `vcn_malloc()`, `vcn_calloc()` and `vc_realloc()`.

The preferred form for passing a size of a struct is the following:

```
p = vcn_malloc(sizeof(*p), ...);
```

The alternative form where struct name is spelled out hurts readability and introduces an opportunity for a bug when the pointer variable type is changed but the corresponding `sizeof` that is passed to a memory allocator is not.

Casting the return value which is a void pointer is redundant. The conversion from void pointer to any other pointer type is guaranteed by the C programming language.

The preferred form for allocating an array is the following:

```
p = vcn_malloc\_array(n, sizeof(...), ...);
```

The preferred form for allocating a zeroed array is the following:

```
p = vcn_calloc(n, sizeof(...), ...);
```

Both forms check for overflow on the allocation size `n * sizeof(...)`, and return `NULL` if that occurred.

18 The inline disease

There appears to be a common misperception that gcc has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 milliseconds. There are a LOT of cpu cycles that can go into these 5 milliseconds.

A reasonable rule of thumb is to not put inline at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you **know** the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the `vcn_malloc()` inline function.

Often people argue that adding inline to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, gcc is capable of inlining these automatically without help, and the maintenance issue of removing the inline when a second user appears outweighs the potential value of the hint that tells gcc to do something it would have done anyway.

19 Function return values and names

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error-code integer (-Exxx = failure, 0 = success) or a "succeeded" boolean (0 = failure, non-zero = success).

Mixing up these two sorts of representations is a fertile source of difficult-to-find bugs. If the C language included a strong distinction between integers and booleans then the compiler would find these mistakes for us... but it doesn't. To help prevent such bugs, always follow this convention:

If the name of a function is an action or an imperative command, the function should return an error-code integer. If the name is a predicate, the function should return a "succeeded" boolean.

For example, "add work" is a command, and the `add_work()` function returns 0 for success or `-EBUSY` for failure. In the same way, "PCI device present" is a predicate, and the `pci_dev_present()` function returns 1 if it succeeds in finding a matching device or 0 if it doesn't.

All EXPORTed functions must respect this convention, and so should all public functions. Private (static) functions need not, but it is recommended that they do.

Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. Generally they indicate failure by returning some out-of-range result. Typical examples would be functions that return pointers; they use `NULL` or the `ERR_PTR` mechanism to report failure.

20 Don't re-invent the containers

The header file `include/vcn/container_bot.h` contains a number of macros that you should use, rather than explicitly coding some variant of them yourself. For example, if you need to calculate the length of an array, take advantage of the macro

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Similarly, if you need to calculate the size of some structure member, use

```
#define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
```

There are also `min()` and `max()` macros that do strict type checking if you need them. Feel free to peruse that header file to see what else is already defined that you shouldn't reproduce in your code.

21 Editor modelines and other cruft

Some editors can interpret configuration information embedded in source files, indicated with special markers. For example, emacs interprets lines marked like this:

```
-*- mode: c -*-
```

Or like this:

```

/*
Local Variables:
compile-command: "gcc -DMAGIC\_DEBUG\_FLAG foo.c"
End:
*/

```

Vim interprets markers that look like this:

```
/* vim:set sw=8 noet */
```

Do not include any of these in source files. People have their own personal editor configurations, and your source files should not override them. This includes markers for indentation and mode configuration. People may use their own custom mode, or may have some other magic method for making indentation work correctly.

22 Conditional Compilation

Wherever possible, don't use preprocessor conditionals (`#if`, `#ifdef`) in `.c` files; doing so makes code harder to read and logic harder to follow. Instead, use such conditionals in a header file defining functions for use in those `.c` files, providing no-op stub versions in the `#else` case, and then call those functions unconditionally from `.c` files. The compiler will avoid generating any code for the stub calls, producing identical results, but the logic will remain easy to follow.

Prefer to compile out entire functions, rather than portions of functions or portions of expressions. Rather than putting an `ifdef` in an expression, factor out part or all of the expression into a separate helper function and apply the conditional to that function.

If you have a function or variable which may potentially go unused in a particular configuration, and the compiler would warn about its definition going unused, mark the definition as `__maybe_unused` rather than wrapping it in a preprocessor conditional. (However, if a function or variable **always** goes unused, delete it.)

Within code, where possible, use the `IS_ENABLED` macro to convert a Kconfig symbol into a C boolean expression, and use it in a normal C conditional:

```

if (IS_ENABLED(CONFIG_SOMETHING)) {
...

```

}

The compiler will constant-fold the conditional away, and include or exclude the block of code just as with an `#ifdef`, so this will not add any runtime overhead. However, this approach still allows the C compiler to see the code inside the block, and check it for correctness (syntax, types, symbol references, etc). Thus, you still have to use an `#ifdef` if the code inside the block references symbols that will not exist if the condition is not met.

At the end of any non-trivial `#if` or `#ifdef` block (more than a few lines), place a comment after the `#endif` on the same line, noting the conditional expression used. For instance:

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

Appendix I: References

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback). URL: <http://cm.bell-labs.com/cm/cs/cbook/>

The Practice of Programming by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X. URL: <http://cm.bell-labs.com/cm/cs/tpop/>

GNU manuals - where in compliance with K&R and this text - for cpp, gcc, gcc internals and indent, all available from <http://www.gnu.org/manual/>

WG14 is the international standardization working group for the programming language C, URL: <http://www.open-std.org/JTC1/SC22/WG14/>

Kernel CodingStyle, by greg@kroah.com at OLS 2002: <http://www.kroah.com/linux/talks/ols.2>