The COMPOSITE Style Guide:
a Guide to C and Tips for Systems Programming

Gabriel Parmer

Computer Science Department
The George Washington University
Washington, DC

Abstract

This document should be seen as a rough guide first to explain the coding conventions and style in COMPOSITE[1] that borrows significantly from the Linux style – for better or worst – and second as a set of reasonable conventions and code formatting methodologies that can lead to (subjectively) easier to understand low-level code. As with all coding style and convention guidelines, they are very subjective; take them or leave them, but take them at least as a data-point with some reasoning behind their rules. Certainly, if you are adding to, or modifying COMPOSITE code, you should follow these conventions.

Many of these conventions and tips derive from a very useful property of C – one that drives its use in low-level systems programming: If you look at a block of code, there should be no "magic". The code itself is self-descriptive, and there is nothing happening "behind the scenes". For example, there are no hidden memory allocations (*e.g.* for closures), no garbage collection, and no implicit reference counting or synchronization (*e.g.* `synchronized` in Java). The more that C is taken away from this transparency, the more you probably shouldn't be using C[2].

This document can be seen as one researcher/developer's opinion on how to let C be C. Alternatively, it can be viewed as a set of hints for making predominantly self-documenting code. That is, code that is descriptive, and readable on its own. This is not as an excuse to not use comments – they are required to explain the non-obvious. Instead the focus is on making what should be obvious, self-evident from the code itself. It has been informed by writing thousands of lines of code of low-level systems code in COMPOSITE and elsewhere, and from reading many more lines of code in a variety of code bases including Linux.

Please see the Appendix (Section 10) for a terse summary of the coding conventions in COMPOSITE.

Currently, this document has organizational problems, so take your editor hat off for the time being.

The audience for this document is intended to either graduate students working with me on COMPOSITE or other projects, or students who are taking my classes. If you don't fall into these groups, realize that the text may be written in a tone and with a certainty that was not intended for you.

1 On Reading and Understanding Other's Code

To understand conventions for *writing* code, it is important to understand how one can productively *read* other's code. Conventions are typically motivated by making code readable and maintainable. They are more rarely motivated by making it more efficient to lay code down onto the screen (programmers usually don't need discipline to encourage them to take the shortest path from A to B). Therefore, this section will discussion how to read code, so that you can better understand why the following conventions exist.

[1] `http://www.seas.gwu.edu/~gparmer/composite`

[2] One reason that C++ is difficult, is because a given block of code can have nearly any meaning, depending on how classes are defined, and what syntax is overridden. This can be powerful and if you want this power, use C++, not C. They say that using C is like being given the gun that you can use to shoot yourself in the foot, and that C++ is the shotgun.

1. **Interfaces as the purpose of the code.** Most systems have well defined entry points and exit points at which other parts of the system interact. These interfaces are often quite nuanced, and it can take some effort to understand their main purpose. Understanding the system interface enables you to understand the purpose of the system. It reveals the main *abstractions* provided by the system, and the operations involving those abstractions. Without an understanding of the purpose of the system, it is difficult to read through the code in a principled manner.
2. **Data-structures as the maps of the code.** Systems programs are often less related to algorithmic design than they are about data-structure design. Web servers are more concerned with tracking sockets, processes, and client requests, and the relationships between them, than they are about algorithmic complexity. Data-structures often include quite a bit of information about the concurrency and locking structures of the program.

   When reading systems code, *it is of the utmost importance to seek to understand the data-structures and how they are related to each*. Use your understanding of the purpose of the system to investigate what the data-structures are meant to record. Draw mental connections between the data-structures and the abstractions provided by the interface. The code is often there to simply glue together the data-structures. It is important to be able to understand them to the degree where you can draw them out on paper as a block diagram.
3. **Data-structure manipulations and operations.** If the data-structures are one of the most important part of the code to understand, the second step one should take is to browse through the functions that operate on those data-structures and maintain their invariants. This will include the functions that encode the interactions between data-structures (these are perhaps the most important).

   When browsing through code, you should always ask "What do I think this code will do, and how it will do it", and only *then* should you look through the code to either confirm your intuition (*i.e.* you understand the code), or refute it. In the latter case, you learn more about the system from the differences between what you expected from the code, and what is actually there. This is your opportunity to refine your own mental model of how the system works. Here it is important to emphasize that *reading code is not a passive activity.* You must always be thinking about what you expect from code before you read it, and then confirm or refute and refine. If you simply read code without actively building expectations about how things are done, you will simply not understand the system for a very long time.
4. **Algorithmic complexity and implementation.** Once you understand the data-structures, and have a reasonable mental model about how they fit together and the invariants on them, then the algorithmic glue to bind it all becomes important. Before you get to this level, you should have a good idea about how the code comes together to provide an implementation for the interface. After you go through this code, you should have a good idea about how the system as a whole fits together, and the trade-offs made therein.

Throughout this process, your concentration should be on refining your mental model about how the system provides an implementation of the interface. Predict how a function will accomplish its goals before looking at the code based on your understanding of the data-structures, and refine your understanding where you are wrong, or oversimplified.

When reading the rest of this document, keep in mind that many of the conventions are tailored specifically to making this process of others reading your code easier. It assumes that they will dig through your code using this methodology.

1.1 On Refining Your Mental Model

It is perhaps easier said than done that you should have a mental model about what a section of code should be doing, *before* actual reading the code. How does one go about forming this mental model? First, realize that this is the essence of *reading* code! To understand code, means that you understand not trivial things like why one conditional is formed the way it is, but why the overall code is structured the way it is. It is about understanding the underlying approach and design to approaching the problem at hand.

That said, the devil is in the details, and when you want to *modify* the code you've been reading, *then* it is important to understand the nitty-gritty in the code. However, you won't be able to achieve this understanding if you don't understand the overall model for how the code fits together.

Building a mental model about a body of code takes practice. A lot of practice. However, I can guarantee that so long as you code in your future, you *will* have to read code. Likely, you will have to read tons of code. This is a skill that is important to acquire. Additionally, it will make you a better programmer. Seeing how others code is very helpful in our personal growth.

To be able to predict what a function does before reading it does rely on some conventions in the code. This is one reason why I am emphasizing conventions in this document. Well named functions, with well-named arguments often make their intention self-evident (see Section 2).

1.2 How "Deep" Do You Go?

One of the most difficult questions you will face when reading complicated code is which function calls to "follow"? This is actually a very difficult question to answer, and really requires that you gain an intuition about a code base. Does the author commonly put significant functionality into functions? If so, you have to be more aggressive in following function calls.

The goal in deciding how deep in function calls to go is "do I have to go into this function to understand the common-case path through this code"? Section 3 is devoted to how to make the common-case code path obvious to readers!

1.3 How Learn to Read Code – the Guaranteed Method

The only method that will guarantee that you can effectively read code is to *read a lot of code*. It becomes much easier to create a mental model about the gist of a body of code when you've seen quite a bit of other code. In short, experience is the main technique for increasing code comprehension. The rest of these conventions will help significantly in making your code readable, but experience is immensely helpful.

2 The Most Important Stylistic Rule

This is the most important convention. I will be so assertive as to call it a truth of programming, a guiding light without which you *will* produce bad code (as in – code that is impossible to read and understand).

Requirement 1. Choose your names well. Choose your names to make an ideal trade-off between descriptive power and terseness. Names include

1. type names (*i.e.* from `typedefs`, `structs`, and `unions`),
2. function names,
3. function argument names,
4. variable names.

These have been ordered from the most important to least. This is not to say that any of them are unimportant.

Convention 1. Function names should use a noun-first naming scheme when possible. This borrows from object oriented programming.

2.1 Naming Requirement 1 Justification

The summary here is simply, do not blow off or skip on making good names for types, functions, arguments, or variables. They are *the most important* part of your code when it comes to others understanding and reading it. "Self-documenting code" is code that does not need comments for someone of sufficient experience to understand it. Though there is no such thing as completely self-documenting code (in any non-trivial code-base, there is some area that does something complicated that requires a comment), but that doesn't mean that it isn't a good goal. This does *not* mean that you just don't write comments. Many people interpret this as such. Instead, it means that you should craft your code such that writing comments doesn't add much to help people understand it.

Toward writing self-documenting code. Proper and intelligent naming is the main way you can attempt to achieve the goal of self-documenting code. Variable names themselves *are* comments in that they convey human readable information about the code. Data-structure names are exceedingly important. This include structure, union, and typedef names. If data-structures are the maps of your code, their names are the legend of the map. Without it, no-one will have the context to understand the map. Function names are also abundantly important as they give a reader the first hint about what the function does (*i.e.* it allows them to predict given their mental model of the code how the function should be implemented). The arguments to the function provide an initial context about what is important in the function. *A function declaration including the function name, argument types and names, and return value tell a startling vivid story.* Do not waste that opportunity to tell the story. Variables in conjunction with the argument names partially document what the body of a function is doing. When deciding if you should create a variable, or simply have longer computation somewhere, the most important question to ask is "would the new variable's name give more of an indication about what this function is doing"? An additional mechanism you have for writing self documenting functions (in addition to the rest of the conventions in this document) is to move some of a function's implementation into separate functions. This enables you to choose more function names that can describe the sub-computations of the function. For example:

```
int request_handle(char *request_txt)
{
	struct request req;
	struct response resp;

	request_parse(&req, request_txt);
	request_process(&req);
	request_formulate_resp(&req, &resp);
	request_send_resp(&req, &resp);
}
```

If all of the functions within `request_handle` were instead the expanded code, it would be much more difficult to understand what the function was doing. However, now it becomes obvious. There are no comments, but the code would also not benefit from them in the current form – ergo, self-documenting code.

An aside: the *onion* model of writing and reading code. Good code (*i.e.* code written to be read and modified) often exhibits what we will call the *onion* property: When tracking through the code, any given function should give the reader some understanding of the computation at a specific level – or layer – of detail. We start at the functions that define the interface into the system. They will look much like `request_handle` above. There is very little detail there, but the reader does understand the flow of the function, and generally what it does. As with an onion, we can "peel off" a layer of abstraction, and go into more detail by looking at the functions that are called. Also like an onion, as you go into deeper layers, the impulse to cry increases. The programmer should realize that a reader does not want to go into *all* the gory details, and protect them from that by providing functions written as in `request_handle`. A reader does not have to dive to the next layer to understand the function, unless they require that level of detail.

In the onion model, it is important to not only provide good function names, but also to know when to push functionality into a separate function.

Tension between descriptive names, and concision. There is always a tension between choosing very descriptive names, and maintain some concision in your code. You'll have to use your own experience here to make the proper trade-off. Certainly know conventions: `i`, `j`, `k` are iterator values. You should *not* use names other than these as they are concise *and* give any programmer that also understands that common convention as much information as almost any other variable name you could choose. It is often OK to use short form names for variables in functions, so long as their type is descriptive. A reader can always look at the type to better understand the variable. See `req` and `resp` in the previous example. However, remember that humans can store 7 ± 2 items in their short term memory. If you rely on this trick for more than 6 variables, then some readers will have to go back to the variable declarations very often. If you do it for 10 variables, nearly all readers will have to go back to the variable declarations constantly. Make their life easier. However, if you have greater than 10 variables, you might consider breaking the function up into sub-functions anyway.

2.2 Naming Convention 1 Justification

This convention is suggesting using function names of the form `noun_operation`. `operation` might be multiple words, but the important part is that `noun` leads the name. An example of this can be seen in the code above. The noun (*i.e.* `request`) is the object that the function is operating on, and the first argument is the actual data-structure for that noun (`req`). This mirrors object oriented programming in which methods operate on a specific object. Where-ever you have a set of functions that operate on a given data structure (see "Data-structure manipulations and operations" in Section 1), this is a good strategy.

Why? This gives your functions that operate on the data-structure some visual unity that a reader can immediately identify. A reader will realize that the functions have a conceptual unity around the data-structure, and will enable them to better form their mental model.

3 Handling Exceptional Cases in Functions

As you will hopefully see in this section, the following two conventions are motivated by (i) keeping the common-case code compact and easy to read, and (ii) avoiding redundancy in code, especially error handling code.

Convention 1. Make sure that the "common-case" code path through your functions reads from top to bottom, and at the same indentation level as much as possible. To achieve this, make your conditionals be true on *error*.

Convention 2. Use `goto` statements to avoid code redundancy and handle error cases. These can and should be used as exceptions that are local to the function (without all the cruft of real exceptions).

Convention 3. Do error handling as soon as you can in a function. This is most important for input (function argument) sanity checking.

3.1 Exception Cases Convention 1 Justifications

Find the bug in the following example of code that take a struct and a list, and make two copies of the struct, adding both into the list. Assume the list is protected by a lock.

```
int list_copy_and_dup(struct bar *b, struct list *l)
{
    if (b) {
        if (l) {
            struct bar *b2, *b3;

            list_lock_take(l)
            b2 = malloc(sizeof(struct bar));
            if (b2) {
                memcpy(b2, b, sizeof(struct bar));
                list_add(l, b2);

                b3 = malloc(sizeof(struct bar));
                if (b3) {
                    memcpy(b3, b, sizeof(struct bar));
                    list_add(l, b3);

                    list_lock_release(l);

                    return 0;
                } else {
                    return -ENOMEM;
                }
            } else {
                return -ENOMEM;
            }
        } else {
            return -EINVAL;
        }
    } else {
        return -EINVAL;
    }
}
```

This code could certainly be improved in various ways. Lets avoid that now, and use this as an example of two ways to structure a given piece of code.

Did you find the bug? If you ran the program on a low-memory system, you quickly would. But you have to ask yourself how often you test in those circumstances. We forgot to (i) free `b2` if `malloc` returned `NULL` when attempting to allocate `b3`, and (ii) release the lock if either `malloc` was unsuccessful. It is *very* easy to make these mistakes. This is the reason that languages such as `go`, `D`, and `C++` provide facilities such as `defer`, `scope`, and `RAII` with automatic destruction, respectively. You can have something similar in C, but you'll have to wait for the second convention. First, lets reformat the code under the first convention.

Following this convention, all error handling is dealt with using conditionals that trigger on error.

```
int list_copy_and_dup(struct bar *b, struct list *l)
{
	struct bar *b2, *b3;

	if (!b) return -EINVAL;
	if (!l) return -EINVAL;

	list_lock_take(l)

	b2 = malloc(sizeof(struct bar));
	if (!b2) {
		list_lock_release(l);
		return -ENOMEM;
	}

	b3 = malloc(sizeof(struct bar));
	if (!b3) {
		free(b2);
		list_lock_release(l);
		return -ENOMEM;
	}

	memcpy(b2, b, sizeof(struct bar));
	list_add(l, b2);

	memcpy(b3, b, sizeof(struct bar));
	list_add(l, b3);

	list_lock_release(l);

	return 0;
}
```

Shorter and easier to understand. The main benefit (and you come appreciate this when you begin to *read* lots of code), is that it is obvious what the common-case path through the code is. Sometimes the erroneous case requires significant processing (*e.g.* deallocating a partially allocated list when `malloc` returns `NULL`). In these cases, consider moving this computation into a separate function to avoid it complicating the code flow for the common-case.

This convention does *not* say "avoid levels of indentation for the common-case path". Often you cannot avoid conditionals, and almost certainly you cannot avoid loops. This conventions makes a statement about how to format *error cases* that can only clutter the code.

3.2 Exception Cases Convention 2 Justifications

This convention is most useful when used in conjunction with the previous. Looking at the previous code, even when following our convention on how to format the error conditionals, and notice that there is significant redundancy in the error handling code. For example, the code for `list_lock_release` is repeated twice. If we were to change the error semantics of this code, we might need to change error code in multiple places. This is a recipe for disaster, but can be fixed with a simple convention: use `goto`s as a way to clean up erroneous cases (similar, as mentioned earlier, to mechanisms in other languages such as `go`, `D`, and `C++`).

```
int list_copy_and_dup(struct bar *b, struct list *l)
{
    int ret = -EINVAL; // default error case.
    struct bar *b2, *b3;

    if (!l) return ret;

    list_lock_take(l)
    if (!b) goto err;

    b2 = malloc(sizeof(struct bar));
    if (!b2) {
        ret = -ENOMEM;
        goto err;
    }

    b3 = malloc(sizeof(struct bar));
    if (!b3) {
        ret = -ENOMEM;
        goto err_free;
    }

    memcpy(b2, b, sizeof(struct bar));
    list_add(l, b2);
    memcpy(b3, b, sizeof(struct bar));
    list_add(l, b3);
    ret = 0;
done:
    list_lock_release(l);
    return ret;
err_free:
    free(b2);
err:
    goto done;
}
```

A few things have happened here:

1. The return value is now stored in the variable `ret`. By default it is set to the most commonly returned error code (here `EINVAL` is the "invalid argument" return code – see `man errno`).
2. The normal return conditions (non-erroneous) are annotated by the `done` label. We know that whenever we want to exit the function, this is what we should jump to. Everything between `done` and the return must be executed for the common path to exit the function. In this case, the lock must always be released.
3. The erroneous cases are laid out in a "first error, last label" listed order. This is intentional, and lets the latter errors (`b3`'s `malloc` failing) fall through, and execute the return code for the previous error cases. This essentially means that when an error happens, it will always execute the "clean up" code for the previous errors as well *without having to duplicate that code anywhere*. Notice how if there is an error allocating `b3`, we free `b2`, and also execute the error handlers for everything else (in this case, this simply jumps to `done` which releases the lock).
4. The return value for each of the error cases is computed at the code location where we know what the error was caused by. This enables the error description to still be as detailed as we want (here just returning `-EINVAL` or `-ENOMEM`), while still avoiding code replication.

Code replication in error code is the worst type of code replication as it can often go untested. This convention is tailored to prevent this from happening by removing the replication. `goto`s can actually add to the structure of your code!

On program size. This convention often creates a few more lines of code, rather than getting rid of them. This is a case where having slightly more code is beneficial as it makes the error handling more explicit and adaptable to future changes. Most of the increase in program size is simply due to labels and gotos, which certainly isn't adding too much conceptual burden. Program size is not the goal; easy to read/understand code, and code that can adapt to future changes are the goals.

On program efficiency. All I can say for this is to look at your compiler's output when using `-O3`. I have never noticed this style being appreciably slower than alternative code formats. Please trust your compiler. It is smarter than you here. Make your code easy to read instead.

Is this encouraging programmers to use `goto`s? I am encouraging programs to use `goto` as a structured technique for exceptional case handling. `goto`s, like many language devices can be used very badly, and produce the very definition of spaghetti code. Use `goto`s in a structured way to remove error-case code redundancy. Do not use them as your new fancy control flow manipulation mechanism.

3.3 Exception Cases Convention 3 Justifications

A reader wishes to distinguish between erroneous cases that are often not important to understand the "main gist" of the code (*i.e.* the mental model), so readers should be given as many hints as possible that distinguish error checking from the main flow of the code.

A reasonable convention to aid this is to do your error checking as soon as possible, and where possible cluster error checks together. A natural consequence of this convention is that all sanity checking on function arguments should be done at the top of the function, even if the arguments aren't used till later[3]. This rule, used in conjunction with those earlier in this section to make it very obvious what the common path through the code is.

4 `typedef` Usage and Naming

Convention 1. Do *not* use `typedef` when the type synonym being created is 1) a composite type (`struct` or `union`), or 2) an array. Use `typedef` to create types only when it adds context to a variable declaration that will help the programmer. Do not use `typedef`s just to avoid typing `struct` in your code. Seeing the `struct` lets the programmer know that passing the variable by value will be potentially costly. Seeing a `typedef` should indicate to the programmer that they can pass the value directly to functions (and across the system call boundary without complications).

Convention 2. *Never* make a `typedef` to for a pointer (*i.e.* `typedef int * ptrtypdf`) as its correct use will violate normal visual norms for accessing and setting the variable.

Convention 3. Always create `typedef` names with a trailing `_t` (*e.g.* `pthread_t`).

4.1 `typedef` Convention 1 Justification

When looking at a block of C code, one should be able to distinguish without much further investigation certain properties about its variables:

1. If I pass the variable to a function by value, will it pass a large chunk of memory, possibly blowing the stack?

[3]Sometimes this is not possible, as some computation must be done that will enable the sanity checking. This is a convention, not a rule that must always be followed.

2. If I return this variable from this function, will it return in a register, or have to force stack allocation in the calling convention?

If it is unclear why passing a `struct` around by value requires stack allocation, see the `cdecl` C calling conventions [4].

Long story short, when you look at a variable, you should know if you can pass it to functions by value, and if this will cause implicit stack allocation or not. You should know if you can return the value from a function, and if you do if it will be doing implicit stack allocations.

`typedef`s gone wrong. As with many language features, there are good ways to use it, and bad. In this case, `typedef` s make seeing what is happening with stack allocation and argument passing difficult in general, as a type name can be equivalent to any type declaration. Take the following code:

```
imatype value;
imafunction(value);
return value;
```

1. If `typedef int imatype;` in the previous code, then nothing out of the ordinary happens. No memory is allocated transparently.
2. If `typedef struct { int array[1024]; } imatype;`, then we have an entirely different case. Passing a structure by value into a function will copy the entire struct, which is expensive and takes 4KB worth of memory here. Even declaring the variable value, does the same. Returning `value` here also causes memory allocation and copying (in the calling function).

An additional reason to not use `typedef` with composite types. The typedef hides the access mechanism to the variable: Do I use = to set its value? Do I use a field in a structure as in `value.array`? As one programming in C usually assumes that they can "see" everything that's going on, hiding information is not in keeping with consistent C programming.

4.2 `typedef` Convention 2 Justification

Using a `typedef` to make a synonym to a pointer type makes code difficult to understand. For example,

```
typedef int * imatype;
...
imatype value;
value->field = x;
```

is simply awkward. Note that the implicit pointer definition is hidden by the `typedef`. This makes common-case code look abnormal – dereferencing `value` when it wasn't declared a pointer.

4.3 `typedef` Convention 3 Justification

`typedef` Naming Conventions All type names created with `typedef` should end with a `_t`. This allows a reader of the code to understand immediately that the name is a type. Imagine the use of the following in code.

```
typedef int u32_t;
typedef int thdid_t;
```

It becomes obvious that `u32_t` is a type, avoiding any confusion.

As with all variable/type names, the name of a type should be as short as it can be (types are used in many places in code that can expand a line to greater than 128 characters), but not too short that its purpose is unclear. Clarity is goal #1.

[4] http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

4.4 When to use `typedef`s?

When *should* you use `typedef`s? `typedef`s have two main purposes:

1. For code clarity. Often having a descriptive type name can make code much easier to read. This is doubly true when you are using function pointers extensively. Creating an appropriate function type saves complicated annotations. For example, compare:

   ```
   int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      void *(*start_routine)(void*), void *arg);
   ```

   with

   ```
   typedef void *(*pthd_create_fn_t)(void*);
   int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      pthd_create_fn_t start_fn, void *arg);
   ```

   Do note the use of `pthread_t` and `pthread_attr_t` that makes the code much more legible. Also, however, note that it is unclear if `thread` or `attr` are structs, or primitive types (to use Java terminology).

   When making a `typedef` for a function, I find a useful convention to be to end the type name with `_fn_t`, so that it is more obvious what the type designates.
2. For portability. Often the concrete type behind a typedef should be different depending on the platform being used. For instance, a `u64_t` should be a `unsigned long` on a 64 bit machine, or an `unsigned long long` on a 32 bit machine.

   It can be used as a form of restricted polymorphism, but I'll leave that for another time.

5 Macro Usage

Macros are one area in C that can completely destroy the *transparency* of C. When looking at a block of code, it can literally do almost anything given specific macros. However, macros really are the best solution (available in C) for some problems. Instead of specific recommendations here, I'll make high-level suggestions, and you'll have to hone your own judgment as to if the proposed macro is tasteful or not.

Suggestion 1. Only use macros when they will reduce code duplication significantly. We use extensive macros in COMPOSITE to define the system call interaction layer. This layer must include inline assembly that is mostly shared across different system calls, and the generated code only differs in the number of arguments passed to a call. We use macros to generate the stubs for each system call, thus avoiding replication of the inline assembly code (`libc` uses macros in a similar way). In Linux, the linked list implementation uses extensive macros to help provide the equivalent of an iterator for simply looping through the list (*i.e.* `for_each`). This avoid duplication of code at each loop site. Macros are C's only alternative to C++'s templates. They can reduce code in the same way by inlining type definitions into `structs` and code (*e.g.* see `SGLIB`).

Suggestion 2. Assume that someone reading your code can lookup the macro definition. Make the macro easy to read, or comment it if it is doing something truly crazy. Make it clear in the code when a macro is being used, so that programs can treat this as a "red flag" – something tricky going on here. The traditional convention is to use all caps for any macro definition, though this does not always hold true (see `for_each` described above).

Suggestion 3. Do *not* use macros where simple inline functions will suffice. Functions carry type information that can be useful for someone reading and using the code, whereas macros often only acquire type context when placed into the code.

Macros vs. `enum`s Macros are still often used for constant values. This tradition is so entrenched in C culture, that it isn't worth fighting against. You *will* see code that uses macros for constant values, so you should get used to it. However, some instead use `enum`s for constant values (see plan 9 source code).

Compare:

```
#define THD_RUNNING 1
#define THD_KILLED  2
#define THD_BLOCKED 3
```

with

```
enum {
  THD_RUNNING = 1,
  THD_KILLED  = 2,
  THD_BLOCKED = 3
};
```

Even better, compare these with

```
typedef enum {
  THD_RUNNING = 1,
  THD_KILLED  = 2,
  THD_BLOCKED = 3
} thd_state_t;
```

When you are defining constants that are part of a larger conceptual whole (*i.e.* different values that describe the state of a thread), then you should use an `enum`, and as the thread state will probably be stored in a Thread Control Block (TCB), you might as well name the "larger conceptual whole" with a `typedef`. This is an instance of using typedefs to give program readers more context to understand your code.

Macros are sufficient, however, if they name a constant that is not part of a larger conceptual whole. For example

```
#define DEBUG_ON 1
```

or

```
#define LOWEST_KERNEL_ADDRESS 0xc0000000
```

These are one of the ways that they are commonly used. Though it can certainly be argued that even in these cases one should use `const int DEBUG_ON = 1;` instead, I have not had a sufficient number experiences where this additional descriptive information was more significant than simply sticking with the C traditions.

6 Formatting

A "religious war" that will never end because it is driven mainly by subjective opinions is one of formatting and indentation. Instead of coming down strongly on either side of any argument here, I want to posit some general principles and goals, and analyze how different means of formatting your code can aid those goals.

Goal 1. We want the formatting itself to encourage good practices, and discourage bad.

Goal 2. We want the formatting to aid the reading process. This is made difficult by the subjective nature of "correct formatting". Your reader might not agree with your style, but the formatting can still aid their reading.

Variable declaration location. c99 enables you to do variable declarations inline with code, as opposed to at the top of blocks (as in C++ and most other languages). Though the original purpose of variables being declared at the top of blocks, was likely due to old, constrained C compilers.

There are arguments that can go either way:

1. Variables should be declared at the top of the scope block. All variables are in one place, so it is easy to reference them (see discussion on variables names in Section 2), and there is a predictable progression about the code. When using this style, It is often best to declare variables in the inner-most block possible (*i.e.* inside for loops, and conditional blocks) especially when the inner block is an erroneous condition. This enables the reader to know by looking at an enclosing block what the common-case variables are, and not those that exist to handle the error cases.
2. Declare variables where they are used. This is very appealing as the first use has the luxury of being very close to the declaration. It has the additional benefit that you can use `const` in more situations. However, for the second, third, and 20th use of the variable, it can be difficult to backtrack through the function and find out where it is defined. It is this lack of predictability and regularity in the code that makes this option not a clear win.

I prefer option 1, as I find it provides more structure to the code, and makes it more predictable to read. This is the style COMPOSITE uses. However, I find it breaks down when you have large functions, and it can be difficult to find the top of a block. However, we probably shouldn't have that long of functions anyway, should we (Section 2.1)?

Use of blank lines. Where should you use blank lines in your code? First, I'll comment simply on many versus few blank lines. Though it might be nice to see your code spaced out (*i.e.* with a blank line between each line of code), it is much more preferable for it to be compact. If a reader has a strong preference for spaced out code, it isn't that difficult to increase the space between lines in their editor. However, when code is compact (without blank lines), more of it can fit on the screen, which makes it easier to read as you can see the variable declarations, function definition, utility functions, etc. – all without moving the code[5].

That said, compact code can have its own issues. You want to use blank lines. You simply want to use them well. The analogy to text is appealing: When do you create a separate paragraph? There are few hard and fast rules concerning this, and it often comes down to style. A rule of thumb is often "use paragraphs to separate thoughts". A similar guide can be used with code: add line breaks when the new code is in some way a separate thought than the previous. For example, setting up some data-structure, and doing some error checking on it, is a separate thought from doing the same for a different data-structure. The goal here is to make your code easier to read. Not more beautiful. Use line breaks to tell the reader "this is something different".

On using brackets as line breaks. Some programmers use brackets in a similar manner to line breaks. That is, they will do the following:

```c
void *malloc(int sz)
{
	if (!sz)
	{
		return NULL;
	}
	...
}
```

instead of

[5]As an aside, you probably want to use something like `follow-mode` in emacs, or comparable technologies, so that you can make the most of your screen to see you can see as much code on screen as possible. Also, when reading code it is essential that it is all "hyperlinked". You can use webpages like `lxr.linux.no`, or `etags` to quickly jump to function definitions.

```
void *malloc(int sz) {
        if (!sz) {
                return NULL;
        }
        ...
}
```

Notice the brackets are essentially acting as line breaks. The same question as before applies here: are these brackets signifying "a different thought"? I don't believe all brackets are the same here. The bracket after the function prototype *is* signifying a change of thoughts. It emphasizes the function prototype independent of the function's code. This is a useful function. In COMPOSITE, the bracket after function prototype is on a separate line. However, for conditionals, loops, and all other brackets used inside of a function, they do *not* signify a different thought. In fact the code inside of the condition is intimately interrelated with the condition itself – it is the same thought. Thus, in COMPOSITE, we do not put the bracket on a different line after conditional, loops, etc...

COMPOSITE conventions. There are some rules that I follow, and that you should follow if modifying the COMPOSITE code-base. These include:

1. After the variable declarations at the top of a block, always insert a line-break.
2. Before the final return statement in a function, always insert a line-break.
3. After clusters of error checking, insert a line break.
4. Put open brackets on a separate line after function prototypes, but not on a separate line after conditionals and loops.

Indentation. First, I'll start off by repeating, there is no correct indentation style. Space, tabs, 2 spaces, 4 spaces, 8 spaces – there is no correct answer. However, I have a preference, and reasons for that preference that might help inform your own preferences. Goal 2 is to have formatting aid the reading process. I believe that the question of indentation is so subjective, that all arguments I've heard for it aiding reading are unconvincing. If you're used to a specific indentation, then it will be easier to read code using that indentation. Thus, I focus on Goal 1: can formatting itself encourage good practices?

I will make an assumption, largely motivated by my own experience: code with many levels of indentation becomes hard to read. This is evident in Section 3. This is largely because it is more difficult to follow the main path of the code. Often this indentation is unavoidable (especially when writing algorithms) as loops are required. However, as you get deeper than 4 levels of indentation, code simply becomes more difficult to parse: you must remember the context for where you are – I'm in 2 loops and 2 conditionals, you must remember what the purpose of those conditionals and loops are, and you are left with only $(7 \pm 2) - 4$ other items in your short term memory. Indentation levels are simply a proxy for "some state that you must remember you're to understand this code". My experience has confirmed this. For example, when writing a complicated algorithm for doing hierarchical min-cut on a graph (see mpd.c in COMPOSITE), the first time I wrote it, I grappled with the complexity. Realizing that I was in too deep, I simply moved some of the inner loops and conditionals into appropriately named functions. This enabled me (as the programmer, not just the reader!) to visually abstract away chunks of the code, and focus on the problem areas more easily.

If we want to discourage high indentation levels, how can we do that? Simple: make indentation more "heavy weight". Thus in COMPOSITE, I define each level of indentation to be 8 spaces. This is consistent with Linux. When you get to 4 levels of indentation, you visually want to avoid any more by moving logic to utility functions. It is a visual cue that your code might be getting out of hand. Of course, there are exceptions. The code in these exceptions is often a little ugly as it stretches across the screen. These conventions optimize for the 99.9% of cases out there that will aid in writing and reading code. It is reasonable to accept that 0.1% might be formatted sub-optimally.

Consistency. Perhaps the most important stylistic advice w.r.t formatting is to simply be consistent. Whichever formatting style you use, use it pervasively throughout a code-base. If you work on a large team, and this becomes difficult, then use `indent` as part of the build/check-in process. Readers rely on conventions to understand which parts of the code are important, and which aren't. If you're schizophrenic about your formatting, you are denying readers this important tool toward understanding your code.

7 `assert`s, `BUG`s, and Error Conditions

Functions go wrong. The arguments to the function might not be valid, the assumed structure of a data-structure (see *invariants* in Section 8) might be inconsistent, or many other possible exceptional cases might occur. Different types of exceptions should be dealt with using different mechanisms.

`assert`ion usage. Often the inputs to a functions are assumed to have some properties. For example, perhaps pointers should not be `NULL`, different arguments must be initialized in a certain manner, or should be related to each other in specific ways, or numerical arguments should be within allowed ranges. If such a condition is not true, the question is this: is this an error within your logic? If it is simply an error, then you should use `assert` statements to verify the truthfulness of the constraints.

You can get rid of assertions using preprocessor macros (by undefining debugging), so you should not put functions that have side-effects that must be executed for program correctness inside assertions (because they might be compiled away!). However, we have not observed more than a 2% performance degradation for including assertions, so we suggest using them even in production code.

In systems, it is very important to differentiate between constraints that must be true concerning input values that are passed from the same "source" as the currently executing function. For example, if you are implementing a data-structure manipulation function in the kernel, if the arguments come from other functions within the kernel, it is appropriate to use assertions on the arguments. In such a case, if the invariant is violated, you'll find out immediately (fail-stop). *However*, if the arguments come from code with different trust properties (i.e. user-level), you do *not* want to assume the inputs are valid. Do not use `assert`ions here.

Exceptional conditions, error codes, and return values. As pointed out in the previous bullet, if the input values do not come from a trusted source, then you should not `assert` on them (such assertions would provide a trivial vector for users to crash a component, or the kernel), and instead appropriate return values should be synthesized. See Section 3 for mechanisms to structure your code to handle such erroneous conditions. If your function is returning an integer (the common case), and negative values are not return values within the defined range, then you should return a `-EVAL`, where `EVAL` is defined in `errno.h`. For example, if the error is the result of an invalid argument, you can return `-EINVAL`, and if you run out of memory, return `-ENOMEM`. Checking for the occurrence of an error in the calling function is often done by checking for a negative return value.

Often the error condition is not actually an error, and is instead part of the interface. For example, returning `-EAGAIN` specifies that "data is not ready now, please try again later" is used on non-blocking interfaces. This is an *exceptional* condition, but not an *erroneous* condition.

`BUG` usage. `BUG()` will kill the current thread by accessing the `NULL` pointer (just as `assert` does). This is done unconditionally. `BUG` statements will not be compiled away even if debugging is not enabled. This is not for checking conditions on inputs, and is instead of stopping the system when something is verifiably wrong. This is similar to `panic()` in other systems such as Linux.

8 Comments

A few notes on notation, and comment style. These are largely convention, and there isn't much of a justification for this style over alternatives. In COMPOSITE, we simply have standardized around these rules.

- `/* TODO: ... */` This comment denotes a message about a functionality that should be added. This functionality is not necessary for correct system operation, but might result in sub-optimal performance or inconvenient functionality. This is work to be done in the future, and work that is not essential.
- `/* FIXME: ... */` This comment denotes a message about functionality, or logic that must be augmented for correct functionality. If, for example, an edge case is not implemented, a `FIXME` should be added to document the lack of this edge case logic. In an ideal world, we'd write code that deals with every case. However, this is often not possible due to the realities of software development. Please minimize the number of `FIXME`s, and use only where appropriate.
- */* ... */ vs. //.* We simply prefer the use of the former over the latter. For multi-line comments, we use the formatting:

  ```
  /*
   * Multi-line
   * comment
   */
  ```

  Ascii-art in multi-line comments to diagram data-structures is welcomed, and the emacs `artist-mode` can be quite useful in producing such diagrams.
- *Pre-conditions, post-conditions, and invariants.* Often during the design process, and later in development, it is useful to employ contract-driven development. Often this is encoded in assertions, which is the optimal solution, but occasionally, it is not realistically possible to include the assertion due to the cost of the checks (i.e. on fast paths). In these cases, you should still document the conditions that must exist before the call of the function (pre-conditions), the conditions that must be true after the function executes (post-conditions), and the invariants often imposed on data-structures (e.g. trees must be acyclic). In such a case, it is useful to include a comment before the function. We use the function to add a child to a tree as an example:

  ```
  /*
   * preconditions: p != NULL, c != NULL, c.parent == NULL
   * postconditions: is_child(p, c), c.parent == p
   * invariants: tree is acyclic.
   */
  int tree_add_child(struct tree_node *p, struct tree_node *c);
  ```

  It is also acceptable to abbreviate "preconditions" as "pre", and "postconditions" as "post".

9 Function Visibility, Location, and Modifiers

A practical question you must answer when using C, is what should a function's visibility be? A function's visibility is determined by a number of factors:

1. Is the function `static`?
2. Is the function implemented in a header file (`*.h`), or a `*.c` file?

How do these decisions change software development characteristics, and efficiency? First, I'm going to assume that we're using `gcc` as your compiler, and not a compiler that does link-time inlining (such as `llvm`). When you implement a function in a `*.c` file, it will generate an object file that will be linked together with other objects. When linking together objects, calls to that function will have the proper function address inserted into the executable. When a function is implemented in a `*.h` file, that function becomes part of the same object as the functions that call it. There are multiple implications:

- `gcc` does not inline functions across objects. You will always have a function all overhead when the function is compiled into a separate object.
- When a function is in the same object (i.e. compiled in the same compilation unit), then it can be inlined according to the compiler's heuristic.

When a function is marked as `static`, the function is not visible outside of this object (compilation unit). You can check this by looking at the output of `nm object.o`; functions with lower case "symbol type" are not visible outside of the object. Thus, a function inside another object cannot call that function. If the function is not `static`, then `nm` will yield a capital-case symbol type, which denotes that it is visible from other objects: functions within another compilation unit can call that function.

A note: a function defined in multiple object files that are linked together cannot be visible outside of either object, or else the linker will fail with an error. This is the equivalent of defining the function twice! No compiler can deal with this case well.

The question is how to decide for any function, where to implement it, and what modifiers to use? Lets go through the conditions:

- `static` and `*.h`: This is a common usage for performance-sensitive code. The function being `static` means that it can be defined in this compilation unit, and it won't conflict if it is also used in another compilation unit and the objects are linked together. The `inline` modifier is often used as well to suggest that the compiler inline the function. This is generally the only reason to implement functions in this way.

  You must really know what you're doing with this implementation style. If you implement functions in this style indiscriminately, you can end up using quite a bit of memory because when functions are inlined, their contents are duplicated at all call-points.
- `static` and `*.c`: By default, this should be the way that you implement functions. Most functions are not called so frequently from outside of their object that they should be in header files, and you want to avoid any naming conflicts with functions with the same name in other objects, so static makes sense.
- Not `static` and `*.h`: Don't do this. There are uses, but they are sufficiently rare, that I'll simply say: don't do this. You are asking for link-time errors where the same header file is included in two `*.c` files, and they are linked together.
- Not `static` and `*.c`: If you have a function that you want to be called from another `*.c` file, this is the default. You will often also have a `*.h` file that includes `only` the prototype of the function. That will enable the other `*.c` file to compile, as it sees the prototype, and upon linking, it can now call the function.

  You can safely think about these functions as defining the external *interface* into the object.

Example. A good example of the split of functionality into `.c` and `.h` files is the `cbuf` implementation. The functions in `src/components/include/cbuf.h` are performance critical, so they should be inlined into the calling code. In contrast, the functions in `src/components/interface/cbuf_c/cbuf_c.c` are also compiled into the same components and define the rest of the `cbuf` interface. Specifically, you can see that the functions in the `.h` files attempt to be small, and call out to "slow paths" that are implemented in the `.c` file. This avoids memory waste by inlining all of the functionality, and makes the "fast path" as fast as possible.

10 Appendix: Style Rules for Systems@GWU

A summary of the coding rules in COMPOSITE follows. I don't provide justifications for these rules as the previous sections cover that. Many of these are adapted from the BSD (`man 9 style`) and Linux Style guides (`Documentation/CodingSyle` in Linux). Note that many of the rules line up with the BSD coding style well.

10.1 Comments

- Do not use comments of the style

```
// comment
```

- Single-line comments:

```
/* single-line comment */
```

- Multi-line comments:

```
/*
 * Multi-line comments use
 * a structure like this
 */
```

- Comments at the head of functions to explain their high-level functionality, and comments that should be emphasized:

```
/**
 * Emphasized comments use a structure like this.
 * More ``*'' entail more emphasis.
 */
```

- Comments do not have to be alone on a line, and can follow code, so long as this does not create too long a line.
- Note that in `emacs`, `M-;` will create an empty comment for you of the single-line style.

10.2 Indentation and Spacing

Indentation. Use an indentation level of 8 spaces. Your code will get ugly if it is indented more than 3 levels. This is intentional and means that you should not do that. If your code indents more than 3 levels, move some of it to a (well-named) sub-function, or handle your error handling correctly.

Function spacing. Function definitions should take the form:

```
static int
sched_block(spdid_t spdid)
{
        /* contents */
}
```

Return values and qualifiers/attributes are on a separate line, and function names are always at the same level of indentation so they are easy for the eye to find.

The only exception to this for functions whose body is a single statement:

Function spacing. Function definitions should take the form:

```
static int
sched_block(spdid_t spdid)
{ /* single statement */ }
```

Function prototypes can take a less stringent form if they can fit onto a single line:

```
static int sched_block(spdid_t spdid);
```

Please specify the name of each of the arguments, not just their types.

Conditional/loop spacing. Spaces/newlines should be used as follows in conditionals that have contents of a single line, or multiple lines:

```
if (expr0 && expr1) {
        /* contents */
}
```

Comparably, `for` loops should be formatted as:

```
for (exprinit ; exprsentinel ; exprincr) {
        /* contents */
}
```

Use a space after these keywords: `if, switch, case, for, do, while`, but not with `sizeof, typeof, alignof, __attribute__`. Note the space before *and* after the `;`. This is to visually emphasize the transition between sections.

The only exception to this spacing, for both conditionals and loops is when the contents can fit onto the same line as the conditional or loop. In that case, the following is acceptable. This is the *only* case where the enclosing { and } are not required.

```
if (expr0 && expr1) stmt;
```

This is commonly used in the following scenario:

```
if (expr0) stmt0;
else       stmt1;
```

If an `if` branch or an `else` branch is multiple lines, then they both must have brackets. It is OK to use the following style with } `else` { on the same line:

```
if (expr0) {
        /* contents */
} else {
        /* contents */
}
```

Breaking up lines. If the expressions constituting the proposition in a conditional do not fit well on a single line, then break them up appropriately onto separate lines, and make sure that the second line is indented to match its enclosing context. Two examples:

```
if (some_complicated_function(a) && some_other_thing(b) &&
    a_really_long_function_name(c)) {
      /* contents */
}
```

or

```
if (a_really_long_function_name(some_function(a), some_other_fn(b),
                                third_function(c))) {
        /* contents */
}
```

Aligned expressions. As shown above, the two statements are vertically lined up. Try and do this uniformly throughout your code. It is especially relevant for assignments.

Example. Many of the above rules can be seen below.

```
struct cbufp_bin *bin;

cbi = malloc(sizeof(struct cbufp_info));
if (!cbi) goto done;

/* Allocate and map in the cbuf. */
cbid              = cmap_add(&cbufs, cbi);
cbi->cbid         = cbid;
size              = round_up_to_page(size);
cbi->size         = size;
cbi->owner.m      = NULL;
cbi->owner.spdid  = spdid;
INIT_LIST(&cbi->owner, next, prev);
INIT_LIST(cbi, next, prev);

bin = cbufp_comp_info_bin_get(cci, size);
if (!bin) bin = cbufp_comp_info_bin_add(cci, size);
if (!bin) goto free;

if (bin->c) ADD_LIST(bin->c, cbi, next, prev);
else        bin->c = cbi;
if (cbufp_alloc_map(spdid, &(cbi->owner.addr),
                    (void**)&(cbi->mem), size)) goto free;
```

Empty lines. It is necessary to use blank lines to break up different "thoughts" in your code, just as we use paragraphs in text. As your function transitions between different phases (error checking, setting up the environment, changing state in structure A, then structure B, and preparing to return), use blank lines to visually denote the change. The following blank lines are mandatory:

- After your variable declarations in a block.
- After clusters of error checking, insert a line break.
- Before the return statement at the end of the function.

`goto` **labels.** `goto` labels should be on a line on their own. They should often be named either `done`, `err`, or `retry`, depending on their function. `retry` labels should be used very infrequently.

10.3 Error Conditions

You **must** have pervasive error checking. If you call a function, you should understand how it passes errors to the caller. You should have a conditional after *every* function call that can have an error to check for that error. If you do not do this, not only will your code never be merged, but you'll waste days of your own debugging time in the future.

Error values. If a function cannot have any error (i.e. it does very little interesting including any memory allocation), then it can return `void`. If a function returns an `int`, then any negative value denotes an error. Thus conditionals checking `if (fn(a) < 0) ...` are common. To provide a more descriptive error code, you can use a value from `errno.h`, for example returning `-ENOMEM` if memory allocation failed, or `-EINVAL` if a passed in argument was invalid. It is not uncommon that a calling function won't change its logic depending on the *type* of failure, but will need to pass that failure condition down to its caller. The pattern for dealing with this is:

```
if ((ret = fn(a))) return ret;
```

If a function returns a pointer, then `NULL` is the default return value. If failure can occur for multiple reasons, then the function should return an `int`, and the pointer should be returned in a function pointer argument.

Error handling. If an error is encountered, you often have to unwind state that was created previously in the function (taking locks, allocating memory, etc...). This should be dealt with using local gotos (equivalent to function-local exceptions in C++/Java). See the above code for an example. The labels being jumped to are name corresponding to the action they will perform. Above, the label `free` will free the `cbi` that has been allocated. The generic label `done` is used when there is no clean-up to do.

Often the function must return an error value. The pattern for this is that a `ret` variable is created in the function of the appropriate return type. The `return` at the end of the function simply returns `ret`. For the successful path, `ret` is set to the non-error value (e.g. `0`). In each condition, it is set to the appropriate value. For example:

```
if (fn(a) < 0) {
        ret = -EINVAL;
        goto done;
}
```

See examples of this in the last example in Section 3.3.

Given this is a common pattern, you can instead use the `cos_throw(ret_val, label)` macro defined in `cos_component.h`.

```
if (fn(a) < 0) cos_throw(-EINVAL, done);
```

10.4 Macros

For multi-statement macros, wrap them in an enclosing `do { ... } while(0)` and always align the trialing \ as in:

```
#define FOO(a, b) do {              \
        something((a));             \
        something_else((b));        \
} while (0)
```

Follow all common macro guidelines and don't write complicated macros unless you know what you're doing. Even then, try and avoid it.

Macros are always capital.

10.5 Naming, Variables, Functions, and Types

Functions should be named in a `subject_verb` style, especially in inter-component interfaces. `subject` in this case is the namespace of the family of functions. You have more leeway in naming `static` functions as they are only visible in the current file. All functions that are intended to be used only in this file (e.g. helper functions) must be static.

Variables. Within a function, do not mix your declarations with code. Declare all the function's variables at the head of the function. If a variable is only used in an enclosing scope, then you can define it there. It is important for a reader to be able to find type declarations easily, and this rule aids that.

Place the declaration of a variable in the most restricted set of {...}. If the variable can be declared at the start of a conditional/loop block, do so.

The cases within switch statements often want to contain local variables. In such cases, adding braces is appropriate. For example:

```
switch(var) {
case VAR_CASE1:
        return 0;
case VAR_CASE2:
{
        int used_locally;

        used_locally = blah();
        /* ... */

        return used_locally;
}
default:
        return -EINVAL;
}
```

When a pointer is declared, the * should bind to the variable name, not the type as in:

```
int *foo, *bar;
```

A commonly used pattern is to abbreviate the type for the name of a local variable. See cbi above in the verbose example which is a variable of the type struct cbuf_info. Though cbi isn't descriptive, it is easy to look at the header of the function, and see the type, and maintain an association between the two via the name.

Type naming and usage. All structs and types should be defined at the top of the file or in a header file (if they are shared amongst files, i.e. visible across an API). typedefs should *only* be used when the type they are creating can be passed in a register, and they should be named with a trailing "_t" to denote "type". It is important to be able to look at the type of a variable and understand its calling conventions. Importantly, never pass a struct or union by value. Always pass it by reference. If the variable is a typedef, or a primitive type, then you can pass by value.

enum types should have an associated typedef, and all values are upper-case. For example:

```
typedef enum {ONE, TWO} enumtype_t;
```

Naming in many cases is dependent on taste, so I'm not going to include huge numbers of rules here.

10.6 Constant instantiation.

Favor hex-decimal representations for values that are not primarily thought of as digit values. For power-of-two values, favor using shifted constants to make it explicit which bit is set. For example, here is a typical flag (bit-field) definition:

```
typedef enum {
        THD_PREEMPTED = 1,
        THD_RUNNING   = 1<<1,
        THD_WAITING   = 1<<2,
} thd_flags_t;
```

For a set of constant values from which only a single value can be used at a time, rely on enum naming, unless the specific values are important. Always use a _MAX, or _MAXVAL enum value to designate the maximum possible value for boundary checking.

```
typedef enum {
        THD_KERNEL,
        THD_USER,
        THD_MN,
        THD_MAXVAL
} thd_type_t;
```

...and later...

```
void
foo(thd_type_t t)
{
        if (t >= THD_MAXVAL) return -EINVAL;
        ...
}
```

10.7 Parenthesis

If there is any ambiguity amongst the associativity of operators, just use parenthesis. If there is any question if you should use them, then use them.

10.8 Include Files

All of your `#includes` should be at the top of the file.

`.h` files should include all structures that are exposed by the API of the compile unit, and the prototype of any functions in the compile unit's API. Files that must be `inline`d can be defined within the include file (as `static inline`). Very few functions require this; keep this in mind.

10.9 Visibility

Visibility of global variables, and functions should be restricted to the smallest possible unit. Helper functions within a `.c` file should always be marked as `static`, as with global variables. Additionally, helper functions should often begin with `__` to visually make explicit that this function is not part of the public API (e.g. `__thread_enqueue(...)`).

The ideal is that the function prototypes and `static inline` functions in the `.h` file exactly match the symbols that are visible in the `.c`'s object (i.e. as seen with `nm`).

Global variables should always be `static` local variables if they are only used within a single function. This is rare.

The use of `extern` should be minimized, but there are good reasons to use it. They often revolve around the desire in complicated programs to split the logic into units, often in separate `.c` files. It is sometimes cleaner for two `.c` files to have direct access to a global variable. Given enough time and a willingness to generalize the functionality, this usage can be hidden by a dedicated `.c/.h` file pair to manage the data-structure, but this is not always the right thing to do.

11 Document History

- **Version 0.1**, 7/29/11 – Hacked out in a day. Includes
  - on reading code,
  - most important style rule (*i.e.* naming),
  - exceptional cases,
  - typedef usage,
  - macro usage,
  - indentation and formatting, and
  - plenty of errors, I'm sure.
- **Version 0.11**, 8/28/11 – Small changes and added the onion model.
- **Version 0.12**, 7/11/12 – Added sections on Comments (Section 8) and Exceptions/Errors (Section 7).
- **version 0.13**, 7/12/12 – added section on function visibility, location, and modifiers, Section 9.
- **version 0.2**, 3/6/13 – added appendix section that summarizes many of the coding style rules (Section 10).
- **version 0.21**, 9/10/15 – added to/edited the appendix section that summarizes many of the coding style rules (Section 10).

TODO

- Edit and proof-read
- OO in C
- Makefiles and the build procedure