ThrowTheSwitch.org Coding Standard

Hi. Welcome to the coding standard for ThrowTheSwitch.org. For the most part, we try to follow these standards to unify our contributors' code into a cohesive unit (puns intended). You might find places where these standards aren't followed. We're not perfect. Please be polite where you notice these discrepancies and we'll try to be polite when we notice yours. ;)

Why Have A Coding Standard?

Being consistent makes code easier to understand. We've made an attempt to keep our standard simple because we also believe that we can only expect someone to follow something that is understandable. Please do your best.

Our Philosophy

Before we get into details on syntax, let's take a moment to talk about our vision for these tools. We're C developers and embedded software developers. These tools are great to test any C code, but catering to embedded software has made us more tolerant of compiler quirks. There are a LOT of quirky compilers out there. By quirky I mean "doesn't follow standards because they feel like they have a license to do as they wish."

Our philosophy is "support every compiler we can". Most often, this means that we aim for writing C code that is standards compliant (often C89… that seems to be a sweet spot that is almost always compatible). But it also means these tools are tolerant of things that aren't common. Some that aren't even compliant. There are configuration options to override the size of standard types. There are configuration options to force Unity to not use certain standard library functions. A lot of Unity is configurable and we have worked hard to make it not TOO ugly in the process.

Similarly, our tools that parse C do their best. They aren't full C parsers (yet) and, even if they were, they would still have to accept non-standard additions like gcc extensions or specifying @ 0x1000 to force a variable to compile to a particular location. It's just what we do, because we like everything to Just Work™.

Speaking of having things Just Work™, that's our second philosophy. By that, we mean that we do our best to have EVERY configuration option have a logical default. We believe that if you're working with a simple compiler and target, you shouldn't need to configure very much… we try to make the tools guess as much as they can, but give the user the power to override it when it's wrong.

Naming Things

Let's talk about naming things. Programming is all about naming things. We name files, functions, variables, and so much more. While we're not always going to find the best name for something, we actually put quite a bit of effort into finding *What Something WANTS to be Called*™.

When naming things, we more or less follow this hierarchy, the first being the most important to us (but we do all four whenever possible):

1. Readable
2. Descriptive
3. Consistent
4. Memorable

Readable

We want to read our code. This means we like names and flow that are more naturally read. We try to avoid double negatives. We try to avoid cryptic abbreviations (sticking to ones we feel are common).

Descriptive

We like descriptive names for things, especially functions and variables. Finding the right name for something is an important endeavor. You might notice from poking around our code that this often results in names that are a little longer than the average. Guilty. We're okay with a tiny bit more typing if it means our code is easier to understand.

There are two exceptions to this rule that we also stick to as religiously as possible:

First, while we realize hungarian notation (and similar systems for encoding type information into variable names) is providing a more descriptive name, we feel that (for the average developer) it takes away from readability and therefore is to be avoided.

Second, loop counters and other local throw-away variables often have a purpose which is obvious. There's no need, therefore, to get carried away with complex naming. We find i, j, and k are better loop counters than loopCounterVar or whatnot. We only break this rule when we see that more description could improve understanding of an algorithm.

Consistent

We like consistency, but we're not really obsessed with it. We try to name our configuration macros in a consistent fashion… you'll notice a repeated use of UNITY_EXCLUDE_BLAH or UNITY_USES_BLAH macros. This helps users avoid having to remember each macro's details.

Memorable

Where ever it doesn't violate the above principles, we try to apply memorable names. Sometimes this means using something that is simply descriptive, but often we strive for descriptive AND unique… we like quirky names that stand out in our memory and are easier to search for. Take a look through the file names in Ceedling and you'll get a good idea of what we are talking about here. Why use preprocess when you can use preprocessinator? Or what better describes a module in charge of invoking tasks during releases than release_invoker? Don't get carried away. The names are still descriptive and fulfill the above requirements, but they don't feel stale.

C and C++ Details

We don't really want to add to the style battles out there. Tabs or spaces? How many spaces? Where do the braces go? These are age-old questions that will never be answered… or at least not answered in a way that will make everyone happy.

We've decided on our own style preferences. If you'd like to contribute to these projects (and we hope that you do), then we ask if you do your best to follow the same. It will only hurt a little. We promise.

Whitespace

Our C-style is to use spaces and to use 4 of them per indent level. It's a nice power-of-2 number that looks decent on a wide screen. We have no more reason than that. We break that rule when we have lines that wrap (macros or function arguments or whatnot). When that happens, we like to indent further to line things up in nice tidy columns.

```
if (stuff_happened)
{
    do_something();
}
```

Case

- Files - all lower case with underscores.
- Variables - all lower case with underscores
- Macros - all caps with underscores.
- Typedefs - all caps with underscores. (also ends with _T).
- Functions - camel cased. Usually named ModuleName_FuncName
- Constants and Globals - camel cased.

Braces

The left brace is on the next line after the declaration. The right brace is directly below that. Everything in between in indented one level. If you're catching an error and you have a one-line, go ahead and to it on the same line.

```
while (blah)
{
    //Like so. Even if only one line, we use braces.
}
```

Comments

Do you know what we hate? Old-school C block comments. BUT, we're using them anyway. As we mentioned, our goal is to support every compiler we can, especially embedded compilers. There are STILL C compilers out there that only support old-school block comments. So that is what we're using. We apologize. We think they are ugly too.

Ruby Details

Is there really such thing as a Ruby coding standard? Ruby is such a free form language, it seems almost sacrilegious to suggest that people should comply to one method! We'll keep it really brief!

Whitespace

Our Ruby style is to use spaces and to use 2 of them per indent level. It's a nice power-of-2 number that really grooves with Ruby's compact style. We have no more reason than that. We break that rule when we have lines that wrap. When that happens, we like to indent further to line things up in nice tidy columns.

Case

- Files - all lower case with underscores.
- Variables - all lower case with underscores
- Classes, Modules, etc - Camel cased.
- Functions - all lower case with underscores
- Constants - all upper case with underscores

Documentation

Egad. Really? We use markdown and we like pdf files because they can be made to look nice while still being portable. Good enough?