clang-format

`clang-format` is a tool to format C/C++/... code according to a set of rules and heuristics. Like most tools, it is not perfect nor covers every single case, but it is good enough to be helpful.

`clang-format` can be used for several purposes:

- Quickly reformat a block of code to the kernel style. Specially useful when moving code around and aligning/sorting. See clangformatreformat.
- Spot style mistakes, typos and possible improvements in files you maintain, patches you review, diffs, etc. See clangformatreview.
- Help you follow the coding style rules, specially useful for those new to kernel development or working at the same time in several projects with different coding styles.

Its configuration file is `.clang-format` in the root of the kernel tree. The rules contained there try to approximate the most common kernel coding style. They also try to follow :ref:`Documentation/process/coding-style.rst <codingstyle>` as much as possible. Since not all the kernel follows the same style, it is possible that you may want to tweak the defaults for a particular subsystem or folder. To do so, you can override the defaults by writing another `.clang-format` file in a subfolder.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master) (Documentation) (process)clang-format.rst`, line 22); *backlink*

Unknown interpreted text role "ref".

The tool itself has already been included in the repositories of popular Linux distributions for a long time. Search for `clang-format` in your repositories. Otherwise, you can either download pre-built LLVM/clang binaries or build the source code from:

> https://releases.llvm.org/download.html

See more information about the tool at:

> https://clang.llvm.org/docs/ClangFormat.html
>
> https://clang.llvm.org/docs/ClangFormatStyleOptions.html

Review files and patches for coding style

By running the tool in its inline mode, you can review full subsystems, folders or individual files for code style mistakes, typos or improvements.

To do so, you can run something like:

```
# Make sure your working directory is clean!
clang-format -i kernel/*.[ch]
```

And then take a look at the git diff.

Counting the lines of such a diff is also useful for improving/tweaking the style options in the configuration file; as well as testing new `clang-format` features/versions.

`clang-format` also supports reading unified diffs, so you can review patches and git diffs easily. See the documentation at:

> https://clang.llvm.org/docs/ClangFormat.html#script-for-patch-reformatting

To avoid `clang-format` formatting some portion of a file, you can do:

```
int formatted_code;
// clang-format off
    void    unformatted_code  ;
// clang-format on
void formatted_code_again;
```

While it might be tempting to use this to keep a file always in sync with `clang-format`, specially if you are writing new files or if you are a maintainer, please note that people might be running different `clang-format` versions or not have it available at all. Therefore, you should probably refrain yourself from using this in kernel sources; at least until we see if `clang-format` becomes commonplace.

Reformatting blocks of code

By using an integration with your text editor, you can reformat arbitrary blocks (selections) of code with a single keystroke. This is specially useful when moving code around, for complex code that is deeply intended, for multi-line macros (and aligning their backslashes), etc.

Remember that you can always tweak the changes afterwards in those cases where the tool did not do an optimal job. But as a first approximation, it can be very useful.

There are integrations for many popular text editors. For some of them, like vim, emacs, BBEdit and Visual Studio you can find support built-in. For instructions, read the appropriate section at:

> https://clang.llvm.org/docs/ClangFormat.html

For Atom, Eclipse, Sublime Text, Visual Studio Code, XCode and other editors and IDEs you should be able to find ready-to-use plugins.

For this use case, consider using a secondary `.clang-format` so that you can tweak a few options. See clangformatextra.

Missing support

`clang-format` is missing support for some things that are common in kernel code. They are easy to remember, so if you use the tool regularly, you will quickly learn to avoid/ignore those.

In particular, some very common ones you will notice are:

- Aligned blocks of one-line `#defines`, e.g.:

  ```
  #define TRACING_MAP_BITS_DEFAULT       11
  #define TRACING_MAP_BITS_MAX           17
  #define TRACING_MAP_BITS_MIN           7
  ```

  vs.:

  ```
  #define TRACING_MAP_BITS_DEFAULT 11
  #define TRACING_MAP_BITS_MAX 17
  #define TRACING_MAP_BITS_MIN 7
  ```

- Aligned designated initializers, e.g.:

  ```
  static const struct file_operations uprobe_events_ops = {
          .owner          = THIS_MODULE,
          .open           = probes_open,
          .read           = seq_read,
          .llseek         = seq_lseek,
          .release        = seq_release,
          .write          = probes_write,
  };
  ```

  vs.:

  ```
  static const struct file_operations uprobe_events_ops = {
          .owner = THIS_MODULE,
          .open = probes_open,
          .read = seq_read,
          .llseek = seq_lseek,
          .release = seq_release,
          .write = probes_write,
  };
  ```

Extra features/options

Some features/style options are not enabled by default in the configuration file in order to minimize the differences between the output and the current code. In other words, to make the difference as small as possible, which makes reviewing full-file style, as well diffs and patches as easy as possible.

In other cases (e.g. particular subsystems/folders/files), the kernel style might be different and enabling some of these options may approximate better the style there.

For instance:

- Aligning assignments (`AlignConsecutiveAssignments`).
- Aligning declarations (`AlignConsecutiveDeclarations`).
- Reflowing text in comments (`ReflowComments`).
- Sorting `#includes` (`SortIncludes`).

They are typically useful for block re-formatting, rather than full-file. You might want to create another `.clang-format` file and use that one from your editor/IDE instead.