#General
This document overrides the original Baseflight style that was referenced before.
This document has taken inspiration from that style, from Eclipse defaults and from Linux, as well as from some Cleanflight developers and existing code.

There are not so many changes from the old style, if you managed to find it.

#Formatting style

##Indentation
K&R indent style with 4 space indent, NO hard tabs (all tabs replaced by spaces).

##Tool support
Any of these tools can get you pretty close:

Eclipse built in "K&R" style, after changing the indent to 4 spaces and change Braces after function declarations to Next line.

```
astyle --style=kr --indent=spaces=4 --min-conditional-indent=0 --max-
instatement-indent=80 --pad-header --pad-oper --align-pointer=name --align-
reference=name --max-code-length=120 --convert-tabs --preserve-date --
suffix=none --mode=c
```

```
indent -kr -i4 -nut
```

(the options for these commands can be tuned more to comply even better)

Note: These tools are not authorative.
Sometimes, for example, you may want other columns and line breaks so it looks like a matrix.

Note2: The Astyle settings have been tested and will produce a nice result. Many files will be changed, mostly to the better but maybe not always, so use with care.

##Curly Braces
Functions shall have the opening brace at the beginning of the next line.

All non-function statement blocks (if, switch, for) shall have the opening brace last on the same line, with the following statement on the next line.

Closing braces shall be but on the line after the last statement in the block.

If it is followed by an `else` or `else if` that shall be on the same line, again with the opening brace on the same line.

A single statement after an `if` or an `else` may omit the "unnecessary" braces only when ALL conditional branches have single statements AND you have strong reason to know it will always be that way.

If in doubt, do not omit such "unnecessary" braces.
(Adding a statement to a branch will break the logic if the braces are forgotten and otherwise make the PR longer).

##Spaces
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).

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);
```

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;
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.

'*' and '&', when used for pointer and reference, shall have no space between it and the following variable name.

#typedef
enums with a count should have that count declared as the last item in the enumeration list, so that it is automatically maintained, e.g.:

```
typedef enum {
    PID_CONTROLLER_MW23 = 0,
    PID_CONTROLLER_MWREWRITE,
    PID_CONTROLLER_LUX_FLOAT,
    PID_COUNT
} pidControllerType_e;
```

It shall not be calculated afterwards, e.g. using PID_CONTROLLER_LUX_FLOAT + 1;

typedef struct definitions should include the struct name, so that the type can be forward referenced, that is in:

```
typedef struct motorMixer_s {
    float throttle;
...
    float yaw;
} motorMixer_t;
```

the motorMixer_s name is required.

#Variables

##Naming
Generally, descriptive lowerCamelCase names are preferred for function names, variables, arguments, etc.
For configuration variables that are user accessible via CLI or similar, all_lowercase with underscore is preferred.

Variable names should be nouns.

Simple temporary variables with a very small scope may be short where it aligns with common practice.
Such as "i" as a temporary counter in a `for` loop, like `for (int i = 0; i < 4; i++)`.
Using "temporaryCounter" in that case would not improve readability.

##Declarations
Avoid global variables.

Variables should be declared at the top of the smallest scope where the variable is used.
Variable re-use should be avoided - use distinct variabes when their use is unrelated.
One blank line should follow the declaration(s).

Hint: Sometimes you can create a block, i.e. add curly braces, to reduce the scope further.
For example to limit variable scope to a single case branch.

Variables with limited use may be declared at the point of first use. It makes PR-review easier (but that point is lost if the variable is used everywhere anyway).

##Initialisation
The pattern with "lazy initialisation" may be advantageous in the Configurator to speed up the start when the initialisation is "expensive" in some way.
In the FC, however, it’s always better to use some milliseconds extra before take-off than to use them while flying.

So don't use "lazy initialisation".

An explicit "init" function, is preferred.

##Data types
Be aware of the data types you use and do not trust implicit type casting to be correct.

Angles are sometimes represented as degrees in a float. Sometimes as decidegrees in a uint8_t.
You have been warned.

Avoid implicit double conversions and only use float-argument functions.

Check .map file to make sure no conversions sneak in, and use -Wdouble-promotion warning for the compiler

Instead of sin() and cos(), use sin_approx() and cos_approx() from common/math.h.

Float constants should be defined with "f" suffix, like 1.0f and 3.1415926f, otherwise double conversion might occur.

#Functions

##Naming
Methods that return a boolean should be named as a question, and should not change any state. e.g. 'isOkToArm()'.

Methods should have verb or verb-phrase names, like deletePage or save. Tell the system to 'do' something 'with' something. e.g. deleteAllPages(pageList).

Non-static functions should be prefixed by their class. Eg baroUpdate and not updateCompass .

Groups of functions acting on an 'object' should share the same prefix, e.g.

```
float biQuadFilterApply(...);
void biQuadFilterInit(...);
boolean biQuadIsReady();
```

rather than

```
float applyBiQuadFilter(...);
void newBiQuadLpf(...);
boolean isBiQuadReady();
```

##Parameter order
Data should move from right to left, as in memcpy(void *dst, const void *src, size_t size).
This also mimics the assignment operator (e.g. dst = src;)

When a group of functions act on an 'object' then that object should be the first parameter for all the functions, e.g.:

```
float biQuadFilterApply(biquad_t *state, float sample);
void biQuadNewLpf(biquad_t *state, float filterCutFreq, uint32_t refreshRate);
```

rather than

```
float biQuadFilterApply(float sample, biquad_t *state);
void biQuadNewLpf(float filterCutFreq, biquad_t *state, uint32_t refreshRate);
```

##Declarations
Functions not used outside their containing .c file should be declared static (or STATIC_UNIT_TESTED so they can be used in unit tests).

Non-static functions should have their declaration in a single .h file.

Don't make more than necessary visible for other modules, not even types. Pre-processor macros may be used to declare module internal things that must be shared with the modules test code but otherwise hidden.

In the .h file:

```
#ifdef MODULENAME_INTERNALS_
… declarations …
#endif
```

In the module .c file, and in the test file but nowhere else, put `#define MODULENAME_INTERNALS_` just before including the .h file.

Note: You can get the same effect by putting the internals in a separate .h file.

##Implementation
Keep functions short and distinctive.
Think about unit test when you define your functions. Ideally you should implement the test cases before implementing the function.

Never put multiple statements on a single line.
Never put multiple assignments on a single line.
Never put multiple assignments in a single statement.

Defining constants using pre-processor macros is not preferred.
Const-correctness should be enforced.
This allows some errors to be picked up at compile time (for example getting the order of the parameters wrong in a call to memcpy).

A function should only read data from the HW once in each call, and preferably all at one place. For example, if gyro angle or time is needed multiple times, read once and store in a local variable.

Use `for` loops (rather than `do` or `while` loops) for iteration.

The use of `continue` or `goto` should be avoided.
Same for multiple `return` from a function and multiple `break` inside a `case`.
In general, they reduce readability and maintainability.
In rare cases such constructs can be justified but only when you have considered and understood the alternatives and still have a strong reason.

Use parentheses around each group in logical and mathematical statements,
rather than relying on the implicit logic and operator priority.
The compiler knows what it’s doing but it should be easy for people too.

#Includes
All files must include their own dependencies and not rely on includes from the included files or that some other file was included first.

Do not include things you are not using.

"#pragma once (https://en.wikipedia.org/wiki/Pragma_once)" is preferred over "#include guards" to avoid multiple includes.

#Other details
No trailing whitespace at the end of lines or at blank lines.

Stay within 120 columns, unless exceeding 120 columns significantly increases readability and does not hide information.
(Less is acceptable. More than 140 makes it difficult to read on Github so that shall never be

exceeded.)

Take maximum possible advantage of compile time checking, so generally warnings should be as strict as possible.

Don't call or reference "upwards". That is don't call or use anything in a software layer that is above the current layer. The software layers are not that obvious in Cleanflight, but we can certainly say that device drivers are the bottom layer and so should not call or use anything outside the device drivers.

Target specific code (e.g. #ifdef CC3D) should be absolutely minimised.

`typedef void handlerFunc(void);` is easier to read than `typedef void (*handlerFuncPtr)(void);`.

Code should be spherical.
That is its surface area (public interfaces) relative to its functionality should be minimised.
Which is another way of saying that the public interfaces shall be easy to use,
do something essential and all implementation should be hidden and unimportant to the user

Code should work in theory as well as in practice.
It should be based on sound mathematical, physical or computer science principles rather than just heuristics.
This is important for test code too. Tests shall be based on such principles and real-world properties so they don't just test the current implementation as it happens to be.

Guidelines not tramlines: guidelines are not totally rigid - they can be broken when there is good reason.