C style guide

Version 1.0

David Hoadley, August 2020

vcrumble@westnet.com.au

Table of Contents

1 Introduction

This style guide is designed to improve the reliability of code written in the C programming language. It is based on the idea that reliability is improved by clarity. That is, your code should make it clear to a human reader what your intentions are, and what it is you are trying to do.

Any code that is of reasonable quality is very likely to have more than one programmer working on it. This may be because programmers other than yourself will be asked to make bug fixes or enhancements, or it may be that one or more modules you have written for one project will also be relevant to some other project (perhaps using the same processor, for example), and that project is being implemented by others. Or it may be that other programmers will be involved in a code review process.

This means that you are not merely writing the code to be understood by the compiler — that is simply a given — you are actually writing the code to be understood by other humans.

This document is not simply an appeal to "write better comments", which must surely be the most ignored piece of programming advice ever given. But it does give some rules, which if followed, will improve the quality of your comments, as well as your code.

It should be remembered that C is a fairly low-level language. Its syntax is computer-focused rather than application-focused. So, compared with higher level languages, there is an additional challenge in communicating your intent to a reader, as opposed to just showing the low-level features of your attempted solution. What's more, C comes with a number of shortcomings, idiosyncrasies and hazardous features inherent in its design. It is not necessarily the most appropriate language to use for general programming. But if you are programming for an embedded system, it may be the only choice available to you.

If C is the most suitable language for you, read on.

This guide has been strongly influenced by the MISRA rules for reliability [1]. It does not replace those rules, and nothing in these rules contradicts MISRA rules. But it does not give you MISRA compliance. If you need to write MISRA-compliant software, consult those rules separately in conjunction with the rules here. This guide has also been influenced by the Barr style guide for embedded software [2]. But there are some differences.

2 Terms

Declaration vs definition

A **declaration** introduces an identifier and describes its type, be it a data type, a data object, or a function. But it does not specify where in memory the data or function will be placed. A declaration is *what the compiler needs* to accept references to that identifier. These are declarations:

```
extern int bar;
extern int g(int lhs, int rhs);
double f(int i, double d);
```
(`extern` can be omitted for function declarations)

A **definition** actually allocates memory for data, and/or contains the executable code for functions. It's *what the linker needs* in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
```

Modules

The term **module** is used to refer to a pair of files: a **header** file, whose name ends in ".h", which contains declarations; and a **body** file, whose name ends in ".c", which contains definitions and executable code. The two files will have the same name, as per rule A.1.

Global vs local

The term **global** is used to refer to variables and functions which are defined in one module but are accessed from other modules.

In some other programming environments the term *global* is reserved for variables that a module defines and makes available to other modules, and the term *external* is used for variables defined in some other module but which can be accessed from "this" one. In other words, every such variable is considered *global* to one module and *external* to the other modules. C uses the `extern` keyword to denote *external* variables, but has no keyword for global ones. Rather confusingly, C documentation tends to refer to both types as **external**, regardless of the viewpoint, or the presence or absence of the `extern` keyword.

The term **local** is used to refer to variables and functions which are defined in one module and accessed only from within that module. Local variables will usually be accessed by more than one function within that module. (The MISRA rules rather awkwardly refer to these as "identifiers with internal linkage".)

The term **function-local** is used to refer to variables which are defined and used only within a single function.

Parameter vs argument

In this document, the term **parameter** is used for those quantities that appear in the declaration and definition of a function. (These quantities are also referred to as the formal parameters). The term **argument** is used for the actual values that are passed

to the function when it is called. (These quantities are also known as the actual parameters.)

Case names

The term **camelCase** is used to describe identifiers with words run together, but with embedded capital letters to indicate the start of each word. In this document, the term is used only for identifiers in which the initial letter is lower-case.

The term **PascalCase** is used to describe identifiers similar to camelCase, but with an initial capital letter. (This is also called upper camel case, but the term PascalCase is being used here to keep it distinct from upper case, below).

The term **lower_case** is used to describe the typical C convention for variables: identifiers made up of all words in lower-case, each separated by single underscore characters.

The term **UPPER_CASE** is used to describe the typical C convention for defined constants: identifiers made up of all words in upper-case, each separated by single underscore characters.

3 Naming Convention

There is a traditional convention within the C community for naming entities in programs, as follows:

defined constants and enums	`UPPER_CASE_NAME`
variables	`lower_case_name`
functions	`lower_case_name()`
user-defined types (typedefs)	no convention, or alternatively, `lower_case_name_t`

This convention has its limitations, and reliability would be better served by departing from it in a controlled way. This style guide does so, for the following reasons.

1. In embedded programming, global variables are an evil which cannot entirely be avoided. Since they occur, debugging will be greatly assisted if they stand out to the reader

   A primary aim of this style guide is to define a convention which instantly identifies global entities as (a) being global, and (b) in which module the entity is defined. This is done by insisting that global entities have a prefix on their names which is taken from the module name in which they are defined. (The fact that global items are required to carry this extra baggage in their names could be regarded as an incentive for the programmer to reduce their use to a minimum — an added bonus!)

   It is intended that a name with a prefix be readily distinguished from one without a prefix. This is not so easily done if all variable names are of the form lower_case_name, since a name of the form prefix_lower_case_name does not actually look any different.
2. The practice of appending a `_t` to the end of a type name matches the standard type names for fixed-width integers (such as `uint16_t`) and `size_t`, but it should be noted that the POSIX standard reserves this form of name for future POSIX types. To avoid problems here, creating your own names ending in `_t` is best avoided.
3. Embedded systems often handle measured quantities. These quantities have units (like volts, degrees, milliseconds etc.) Sometimes quantities are stored in fixed point form using integers (e.g. the least significant bit might represent 0.1). Reliability will be improved if a standard format is used to indicate such variables.

As a result, this style guide departs from the traditional C convention. It has been influenced by conventions used in C++ and Java programming. (But for those who find this idea too much of an affront, an alternative form which sticks with the lower_case convention is also offered, despite it being visually more awkward.)

The general form is

defined constants and enums	`[[PREFIX_]]UPPER_CASE_NAME[[_suffix]]`
variables	`[[prefix_]]camelCaseName[[_suffix]]`
functions	`[[prefix_]]camelCaseName[[_suffix]]()`
user-defined types (typedefs)	`[[Prefix_]]PascalCaseName`

where the items shown between double-brackets [[...]] appear only on some identifiers. The prefix appears only for global entities. The suffix only appears on

measured quantities and/or fixed-point quantities. In practice, most identifiers will have neither prefix or suffix, so this is not as ugly as it first appears. Underscores only appear in names that have a prefix or a suffix. The decision to use camelCase for variables and functions rather than lower_case makes the prefix and suffix much more visible when they do appear.

The alternative form, for those who insist on lower case only, is as follows:

```
[[prefix_g_]]lower_case_name[[_suffix]]
```

i.e. a "`_g_`" separating the prefix from the rest of the name, for global variables.

It is a common Java convention to use PascalCase for class names and camelCase for variable names [3], and there is some use of this convention in the C++ community also [4]. The nearest that C gets to a C++ (or Java etc.) class is a `typedef`. So by analogy, using PascalCase for typedef's will make sense to those familiar with C++ or Java, and it avoids the problem of the reserved `_t` suffix mentioned previously.

4 The Rules

Rules - contents

A File naming

Rule A.1 Give the header and body file in every module the same name

For example, the config module comprises the two files config.h and config.c

Exception 1. The body file containing the main() function does not need to have (and probably should not have) a corresponding header file.

Exception 2. If you are writing a library of routines, made up of several modules, you will need to have a header file for customers of your library. This header will carry the name of the library as a whole, not the name of any of the individual module body files.

Rule A.2 The first 8 characters of every module name in a project shall be unique

The first 8 characters of every module name will be used as a prefix on the names of any global functions or variables declared in that module (see rules C.2, D.2, E.2 & F.2), so this rule is required in order to avoid ambiguity. (Thanks to Barr style guide [2])

Rule A.3 The file containing the main() function shall contain "main" in its name

This is a great help to the person who needs to understand your code. (Thanks to Barr style guide [2])

B Naming within the code

General

Rule B.1 Do not start any name with an underscore character

In C, certain names starting with an underscore character are reserved (but not all of them). But rather than get tied up in the subtleties of the rules, is is just safer to avoid trouble here by not ever creating such a name.

Rule B.2 Do not give anything a name that is already used at an outer level of scope

Although the compiler is quite happy if you do this, the inner level name will "hide" the outer level one. This is referred to as shadowing, and it is quite likely to confuse a person who is maintaining your program later. It increases the risk of a bug being introduced.

This is MISRA-2004 rule 5.2 [1].

Enforcement: If using gcc or Clang, enable the warning flag `-Wshadow`

Suffixes

Rule B.3 Use a suffix to show the units of a measured quantity

In embedded systems, often quantities appear that are based on measured quantities (like temperature, voltage, speed, etc). The suffix shall be used where relevant to declare the engineering unit that applies to such a variable. Being consistent with this can save you a lot of trouble. For example, is your angle in degrees or radians? Or is it in a 16-bit number in which wraps around at 360° (i.e. 32768 represents 180°)?

Here are some suggested suffixes:

angles:	`_deg`	`_rad`	`_16bit`	`_arcmin`	`_arcsec`	
temperatures:	`_degC`	`_degF`	`_K`			
distances:	`_km`	`_m`	`_mm`	`_um`	`_nm`	
	`_NMi`	`_mi`			(nautical mile, mile)	
	`_yd`	`_ft`	`_in`		(yard, feet, inches)	
times:	`_d`	`_h`	`_mins`	`_s`	`_ms` `_us` `_lc`*	
volumes:	`_L`	`_mL`	`_ML`		(litres, millilitres, megalitres)	
	`_galUK`	`_galUS`			(Imperial gallon, US gallon)	
pressures:	`_kPa`	`_MPa`	`_bar`			
	`_psi`					
speeds:	`_mps`	`_kmph`			(m/s, km/h)	
	`_kn`	`_miph`			(knots, miles/hour)	
other:	`_V`	`_mV`	`_A`	`_mA`	(volts, millivolts, amperes, milliamps)	

As you can see, the suffixes are case-sensitive. A lower-case 'p' has been used for "per".

(*) `_lc` is used for loop counts i.e. counts of the main (timing) loop of the program, assuming there is one.

Rule B.4 The suffix shall be in mixed case

The suffix shall not be restricted to lower case nor to camelCase. It shall have upper case and lower case letters as required by the units in question. Examples are given in rule B.3.

If the suffix is appended to a constant which is otherwise entirely in upper case, do not convert that suffix to upper case. For example there is a world of difference in meaning between

```
#define    MAX_POWER_mW  10
#define    MAX_POWER_MW  10
```

Better understanding of your code by a reader will be achieved by this slight deviation from the standard C convention.

Rule B.5 Use SI abbreviations for SI units

Use `_s` for seconds, not `_sec` or `_secs`. Use `_h` for hours, not `_hr` or `_hrs`. Use `_d` for days.

Use an upper-case `_L` for litres. SI allows both upper and lower case for this unit, but a lower-case `l` is too easily mistaken for a `1`. Likewise with millilitres and megalitres (`_mL` and `_ML`).

Exception: A case can be made for not using `_min` for minutes. This can potentially be confused with using min for minimum. This is a greater risk if you are using lower_case_names, but it can still arise regardless of the capitalisation convention.

Exception: don't use `_t` to mean tonnes (which Americans like to call "metric tons"), even though this is the accepted unit abbreviation for this SI-derived unit. The suffix `_t` is widely used for type definitions within both the C and POSIX standards, so it would be confusing to use it as part of a variable name.

Rule B.6 Don't use SI abbreviations for non-SI units

For example, `_m` is reserved for metres. Don't use `_m` for miles, use `_mi` instead.

Rule B.7 Use SI prefixes for SI multiples

Use `_k`, `_M`, `_G`, etc. for multiples greater than 1. For example `_kW`, `_MW`, `_GW`. Use `_m`, `_n`, `_p` etc. for multiples less than 1. For example `_mA`, `_nm`, `_pF`.

Multiples of 1000 or less are lower case. Multiples of 10^6 or more are upper case.

Exception: Use `_u` for micro, since the character µ is not supported.

Rule B.8 Don't use SI prefixes for non-SI multiples

In particular, don't use `_k` or `_K` for 2^{10}. Use `_Ki` (kibi — the standard IEC/ISO abbreviation). Likewise, don't use `_M` for 2^{20}, use `_Mi`, and don't use `_G` for 2^{30}, use `_Gi`.

There should be a special hall-of-shame entry for the programmer who first thought "I am going to use megabyte *not* to mean 1,000,000 bytes but to mean 1,048,576 bytes. After all it is less than a 5% error". Some programmers even now are reluctant to give up this bad habit. It is wrong — get over it.

Rule B.9 Don't use ambiguous units for the suffix

Some examples of ambiguous suffixes are:

`_deg`	for a temperature (Celsius or Fahrenheit?)
`_pint, _qt, _gal`	for volume (do you mean US or Imperial?)
`_ton`	for weight (do you mean US or Imperial?)
`_mpg`	for fuel economy (as above)

Multipliers

Rule B.10 Use a multiplier in the suffix for all fixed point quantities

Often in embedded programming, it is useful to use fixed point arithmetic quantities. If this is done, the suffix must reflect this. This may be combined with the specifier of the engineering units. For example:

`uint16_t foo_x10`	a fixed point number in which the least-significant bit represents 0.1

`uint16_t`	`elapsed_sx10`	time, where the least significant bit represents 0.1 seconds
`int16_t`	`battery_Ax8`	amps, where the least significant bit represents 0.125 A, (so shift right by 3 to truncate to whole amps)
`int16_t`	`battery_Af3`	an alternative notation for `_Ax8`, indicating the number of bits to shift. This notation becomes more convenient when the number of bits shifted is larger.

Where the least significant bit represents a number greater than one of the quantity in question, use `_q` as an indicator. For example:

`uint8_t`	`alertTime_sq5`	least significant bit = 5 s. Range 0 – 1275 s

C Variable names

Rule C.1 Name local variables using the form `camelCaseName`

This rule covers local variables declared at file level, and function-local variables (declared within a function). Don't include a prefix on these variable names.

Append a suffix to the name where relevant, as discussed in sub-section "Suffixes".

Rule C.2 Name global variables using the form `prefix_camelCaseName`

The prefix shall be used only for global variables. It shall be separated from the rest of the name with an underscore. The prefix shall be taken from the name of the header file in which the global variable is declared (or the first 8 characters of the header file name, if the file name is long). All variables declared in header files shall carry this prefix. The prefix shall be in lower case.

Append a suffix to the name where relevant, as discussed in sub-section "Suffixes".

Alternative form: If you are insisting on lower_case_names, then global variables shall be named with the form `prefix_g_lower_case_name[[_suffix]]`.

Rule C.3 Include a state verb in the names of Boolean variables

Boolean variables shall have names like `isFound`, `hasSubfolders`, `isFull` etc. In this way, statements like `if (buffer.isFull) { … }` read very clearly.

D Function naming

Rule D.1 Name local functions using the form `camelCaseName()`

Functions are named like variables. Append a suffix to the name if the function returns a value in particular engineering units. Other functions and void functions do not need a suffix.

Rule D.2 Name global functions using the form `prefix_camelCaseName()`

The prefix is to be taken from the module name exactly as in rule C.2 for variable names.

Again, append a suffix to the name only if the function returns a value in particular engineering units.

Alternative form: If you are insisting on lower_case_names, then global functions shall be named with the form `prefix_g_lower_case_name[[_suffix]]()`.

Rule D.3 Name every function parameter in a prototype

C allows function prototypes to be declared with a list of un-named types. For example

```
extern void tone_start(int, int, uint16_t, uint16_t, uint16_t);
```

Don't do this. Name them all, as it makes your intentions much clearer:

```
extern void tone_start(int channel,
                       int type,
                       uint16_t period_sx10,
                       uint16_t startFreq_Hz,
                       uint16_t endFreq_Hz);
```

This is MISRA-2004 rule 16.3 [1].

Rule D.4 Use a suffix on function parameter names as much as possible

Consider the following function declaration:

```
float tiltDevice(float angle);
```

It can be quite frustrating to find a function that someone has written containing a parameter like this, without any information telling you whether you must pass a value in degrees or radian. (If you are lucky, a nearby comment will tell you, but frequently you are not lucky.) Declaring the parameter as `float angle_deg` makes life easier for everyone.

E Defined constants

Rule E.1 Constants created with `#define` shall be in UPPER_CASE

It is a standard convention in the C community to use upper case for constants.

Exception: But if there is a suffix present, that suffix shall be in mixed case — see rule B.4.

Rule E.2 Name global constants using the form `PREFIX_UPPER_CASE_NAME`

The prefix shall be derived from the module name, like rule C.2 (for variables) but shall be in upper case.

Exception: constants defined in general.h (see section 6).

F Type names (typedef)

Rule F.1 Name local typedefs using the form `PascalCaseName`

The initial capital letter gives an immediate clue that this is a type name, not a variable name.

Rule F.2 Name global typedefs using the form `Prefix_PascalCaseName`

The prefix, derived as usual from the module name (rule C.2), but shall have an initial capital letter.

Rule F.3 Don't use the same name for a struct tag and the typedef'ed struct

For example, don't write

```
typedef struct Foo Foo;
```

as this can be confusing for a reader. The two `Foo`'s are quite distinct entities.

Structures tend to be defined in one of two ways:

```
struct Foo {
    type field1;
    type field2;
};
```

or

```
typedef struct {
    type field1;
    type field2;
} Foo;
```

In the first case, variables and function parameters are defined with the syntax

```
struct Foo varname (or struct Foo *pointername)
```

and in the second, simply by

```
Foo varname    (or Foo *pointername)
```

It is easy to overlook that in the first form, Foo is a "tag", which exists in a separate namespace from our other variable and typedef names. The second form declares Foo to be a type. But the second of these forms is actually an abbreviation of the full declaration, which is of the form

```
typedef struct a_tagname {
    type field1;
    type field2;
} Foo;
```

in which the tag name (shown here as `a_tagname`) has been omitted. If you do not supply a name for `a_tagname`, the compiler treats this as an anonymous struct, and generates an internal name for its own use.

There is one circumstance in which you may not want the compiler to generate its own internal name, and two circumstances in which you must use the full form of the declaration.

Optional: If your compiler gives error messages which use the tag name but not the typedef name, you will find those messages not terribly helpful if the name is one the compiler created rather than you. Likewise, if you are examining a linker map file which uses the tag name rather than the typedef name, you will want to use the full form of the declaration. In these cases, you may want to use the full form of the declaration, providing your own name for `a_tagname`.

Compulsory: You will need to use the full form for the following two purposes.

1. Opaque types
2. Forward references.

If you want to define an opaque struct (i.e. one in which you do not want the internal details of the struct to appear in the header file), you will need to put a declaration in the header of the form

```
typedef struct Foo_tag Foo;
```

and in your body file

```
typedef struct Foo_tag {
    type field1;
    type field2;
} Foo;
```

In other modules, you will not be able to declare variables of the type `Foo variableName;`, because the declaration of Foo that is visible to those modules is incomplete. But you can declare pointers of the form `Foo *pointerName;`. Those modules cannot access the fields of type `Foo`, all they can do is pass the pointer to functions in the module that defines `Foo`, to do whatever it wants with.

Declaring an opaque struct like this cannot be done without specifying the tag name (here – `Foo_tag`).

For a forward reference, use the form

```
typedef struct Foo_tag Foo;
typedef struct Foo_tag {
    type field1;
    type field2;
    Foo  *next;
} Foo;
```

This struct contains a field (`*next`) that points to another variable of type `Foo`, included inside the definition of `Foo` itself. This can only work with the single-line incomplete declaration before the full definition, and both must contain the tag name.

This rule requires that you do not use the same name for a tag and a typedef. This is also required by MISRA-2004 rules 5.4 and 5.6 [1].

G Module contents

Header files

Rule G.1 Header file guards must be created using a standard form

This is needed to prevent problems when a header is included more than once into a source code file.

The standard form is to `#define` a constant using the header file's name, followed by _H. The first line of the file must be an `#ifndef` statement, testing whether that constant is already defined, and the last line of the file must be an `#endif` statement, followed by a comment indicating the same constant.

For example, if the header file is config.h, the first two lines in the file must be

```
#ifndef CONFIG_H
#define CONFIG_H
//  rest of header file
```

and the very last line of the file must be

```
#endif /*CONFIG_H*/
```

Rule G.2 Header files shall contain only things that are used by other modules

Do not put `#define`'s in a header file that are used only in the corresponding body file. Put those `#define`'s in that body file itself.

Likewise, do not put variable or function declarations in a header file that are used only in the corresponding body file. Put those declarations in the body file itself, and declare them `LOCAL` or `static` (see rules G.6 & G.8)

Rule G.3 Everything declared within a header file shall have the standard prefix

All defined constants, macros, typedefs, function and variable declarations in a header file shall be named with a prefix which is taken from the name of the header file itself (or its first 8 characters, if it has a long name). See rules C.2, D.2, E.2 & F.2.

Exception: general.h (see section 6). This would be too tedious.

Rule G.4 No variables shall be defined in a header file

Variables may be *declared* using the `extern` keyword, but they may not be *defined*. (See the difference between declaration and definition in section 2 Terms.)

If you attempt to define variables within header files, you run the risk that the linker will see the variable as being "multiply defined" — i.e. defined separately in each .c file which includes that header — and abort. But sometimes no error is flagged. This is because C has a concept known as "tentative definition" which can allow such definitions without causing an error, if the variables are not initialised. Don't make use of this feature.

Rule G.5 Each global object or function shall be declared in one and only one header file

This is MISRA-2004 rule 8.8 [1].

Body files

Rule G.6 All file-level variables defined within body files must be declared `static`, unless they are declared in the header

The design of the C language has some rather unfortunate features. One of these is that variables defined at file level in any body file are treated by the compiler by default as global variables. This is not what we want in general. We want only those variables that we intend to be global (and have declared in the header file) to be accessible to other modules. We don't want accidental access to all of the other internal variables of our module. So we must tell the compiler explicitly to make the variables local.

A second unfortunate feature is that C uses the keyword "static" to define such variables as local variables. This is one of three quite distinct uses of the keyword, and it doesn't really make logical sense (since both local and global variables are in static, as opposed to automatic, storage).

So all local variables must be declared `static`. Only global variables which have been declared in the module's header file may be defined in the body file without the use of the keyword `static`. Better still, include the header file general.h (see section 6) and declare the local variables as `LOCAL` instead.

(Note that this is not referring to function-local variables. The scope of these variables is already local to the function without any special declarations. In fact, within a function, declaring a variable `static` has a distinctly different meaning.)

This is part of MISRA-2004 rule 8.10 [1].

Rule G.7 Do not declare any `extern` variables within body files

If you need access to a global variable that is defined in another module, you must include that module's header file. That header file will contain the extern declaration you need (as per rule G.4).

This rule ensures that all modules that access a global variable get the same declaration of that variable. (Otherwise, it would be quite possible to declare a variable as `int` in one module and `float` in another, for example without the compiler alerting you to the bug.)

Rule G.8 All functions declared within body files must be declared `static`, unless they are global functions declared in the header

This is analogous to rule G.6 for local variables. (This is a second use of the keyword `static` which has nothing to do with static versus automatic storage.)

Better, include the general.h file (see section 6) and declare these functions `LOCAL`.

This is part of MISRA-2004 rule 8.10 [1].

Rule G.9 All local functions (as per rule G.8) shall have a separate prototype declaration

These prototypes are to be in section 7 of the file as per rule J.2.

Enforcement: If using gcc or Clang, enable the `-Wmissing-prototypes` warning flag and the `-Wimplicit-function-declaration` flag (or `-Wall`, which includes this flag).

This is MISRA-2004 rule 8.1 [1].

H Function definitions

Rule H.1 Place comments describing what a function does, what it returns, and what its parameters are, with every function

Go on, write these comments before you actually write the code. You'll be surprised how often it clarifies your thinking before you start coding, and might even save you time overall!

See rule J.19 for where to place these comments, and see the example code given in rule H.2 for an example.

Rule H.2 Function comments must describe the consequences of invalid input for every parameter

With every parameter, if it has limitations on its valid range, the comments in the function header must include

(a) what the valid range is, and

(b) what happens if an out-of-range value is supplied.

For example, does the function return an error code, does it abort with a precondition assertion failure, does it quietly return an invalid answer, or does the program crash? Be explicit.

Likewise, if the parameter is a pointer, what happens if `NULL` is passed to this function? Be sure to say.

For example:

```
global void amps_setAttenuation(int device, uint8_t atten_dBx2)
/* Set the output volume (attenuation) on the specified device
 Inputs
    device      - which TAS2505 are we talking to. Valid range [1,5], otherwise
                  communication will be ignored by all devices
    atten_dBx2  - approximate attenuation in dB (x 2, i.e. LSB = 0.5 dB). At
                  attenuations higher than about 50 dB the attenuation is
                  greater than the figure specified.
                    Valid range [0, 116 (0x74)]. Numbers above this range are
                  clamped, but 0x7f is accepted, as a special value meaning
                  "mute the output" (and is defined as TAS2505_MUTE_OUTPUT).
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
{
    uint8_t     regValue;

    regValue = atten_dBx2 & 0x7f;
    if ((regValue != 0x7f) && (regValue > 0x74)) { regValue = 0x74; } /* Clamp*/

    setRegister(device, P1_R46_SPEAK_VOL, regValue);
}
```

Rule H.3 Doxygen comments describing functions to be in body files, not header files

If you are using doxygen to create documentation from your code, keep the doxygen comments as close as possible to the relevant code. This reduces the chances of forgetting to keep the doxygen comments up-to-date when the code is modified.

Doxygen is normally set up to expect the comments describing functions to be in the header file, but it can be configured to find them in body files instead. This makes the header file easier to read. There is no reason to have this information in the header, since you are providing other people with doxygen-created documentation which gives them this information.

Parameters to functions

Rule H.4 All function parameters passed by reference (pointer) shall be declared `const` *unless the function modifies the parameter*

This enables better compiler checking for possibly uninitialised variables, and better optimisation.

This is MISRA-2004 rule 16.7 [1].

I Declarations

Declaration of pointer variables

Rule I.1 Pointer declarations must declare only one pointer per line

Consider the following definitions.

```
int *foo;
int * foo;
int* foo;
```

All three declare foo to be of type “pointer to int”, and they are equally valid to the compiler. Semantically, it makes sense to prefer the third form, i.e. to consider `int*` to be a type (i.e. a pointer) just as `int` is a type. But C has a dreadful trap for the unwary here.

Consider the following declaration of two variables
```
int* foo, bar;
```

Although this looks like it declares two pointers to int, it actually declares one pointer to an int, and one int. (It could be argued that this is a design fault in the language.)

There are two possible approaches to avoid this problem:

1. Always use the first of the three forms listed above, even though it does not make sense semantically. (In C, this is the usual approach.)
2. Always declare a pointer on a line of its own. In this case, any of the forms listed above can be used safely.

This rule has opted for the second of these options.

Rule I.2 Don’t typedef pointers to data

Consider the following code

```
struct Foo_tag {
    int bar;
    int baz;
};
typedef struct Foo_tag  Foo;
typedef struct Foo_tag  *FooRef;
```

The second typedef has unexpected effects. Consider what happens for example, if `Foo` and `FooRef` are combined with `const`. The following declarations appear to be equivalent to each other, but they are not:

```
Foo          v, w;
const Foo   *x = &v;
```
This is a pointer to constant data
e.g. x->bar = 1 is not allowed, but x = &w is allowed.

```
const FooRef y = &v;
```
Here the pointer is constant, but the data is not
e.g. y = &w is not allowed, but y->bar = 1 is allowed.

Typically when we declare a const pointer, we want the data to be constant (see for example rule H.4 — parameters to functions). Declaring an object or a parameter as being of type `FooRef` will not have the desired effect.

Exception: it is unlikely to cause confusion if the pointer is to an opaque type. Since the pointer is never able to be dereferenced by client code, a `typedef Foo *FooRef` will be harmless.

Other declarations

Rule I.3 Use the `char` data type only for text characters

Use the `int8_t` and `uint8_t` data types for any other byte-sized variables or arrays. (Note that if you are using a C89/C90 compiler, you will need to define these two datatypes yourself, using typedef statements (or include the general.h header file (see section 6):

```
typedef signed char    int8_t;
typedef unsigned_char  uint8_t;
```

Keeping a clear distinction between text and 8-bit numbers is part of "reliability is improved by clarity"

This is MISRA-2004 rule 6.1 [1].

Rule I.4 Do not (directly) use the `signed char` or `unsigned char` data types

Use `int8_t` and `uint8_t` instead. This rule and the one before it help make things clearer to a reader of your program.

This is MISRA-2004 rule 6.2 [1].

Rule I.5 Use `int8_t`, `int16_t`, `int32_t`, `uint8_t`, (etc) for integers where the size matters

Typically this will be for data derived from system registers, or from structs which define transmitted data. (Note that if you are using a C89/C90 compiler, you will need to define these datatypes yourself, using typedef statements. Or include the general.h header file (see section 6), but be aware that you may need to modify the definitions.)

This is MISRA-2004 rule 6.3 [1].

Rule I.6 Bit fields shall only be defined to be of type `unsigned int` or `signed int`

Declare every bit field to be signed or unsigned, and don't use any type names that imply something about the size of the data (such as uint8_t, or short). The specifier "int" may be omitted in this specific instance — e.g. the following is acceptable:

```
struct {
    unsigned fieldA : 1;
    unsigned fieldB : 2;
    signed   fieldC : 2;
} foo;
```

The word `signed` or `unsigned` must appear. (As an aside, when defining variables (as opposed to bit fields), do not omit the `int` keyword. Be specific.)

This is MISRA-2004 rule 6.4 [1].

Rule I.7 Bit fields of signed type shall be at least 2 bits long

Basically it makes no sense to have a signed value of 1 bit in length. Unfortunately, this rule means that you cannot declare a bool field to be of one bit in length, even though that would seem to make sense. C defines booleans to be a signed integer type, unfortunately.

This is MISRA-2004 rule 6.5 [1].

Rule I.8 Define the `bool` data type if it is not already defined

If you are using a C89/C90 compiler, you will need to add definitions like the following:

```
typedef int     bool;
#define false   (bool)(0)
#define true    (bool)(1)
```

Alternatively, include the general.h header file (see section 6), which defines this for you for C89/90 (or includes the relevant system header for C99 and later). (Note that `false` and `true` have been defined in lower case rather than upper, for compatibility with C99 and later standards.)

Use this data type for all boolean variables and flags; don't use `int`.

J Code layout

The rules in this section are intended to provide a consistent look for your code. If your organisation has conflicting style guidelines, stick to those instead of these ones. Use those rules here that are not covered by any other guidelines. In particular, see rule J.17.

Rule J.1 Group header file contents into standard order

The order of declarations shall be:

1. Opening two lines of header file guard, as per rule G.1.
2. Header comments, describing this module
3. `#include` files, if any. Only include those files which are essential for the header itself; don't include files that the body will need (include those in the body file instead).
4. `#define` statements for global constants, if any
5. `typedef's` and declarations of `structs` and `enums`, if any
6. Global function declarations (prototypes)

7. Global variable declarations, if any
8. Closing line of header guard, as per rule G.1.

Rule J.2 Group body file definitions and code into standard order

The order of items in a body file shall be:

1. Block comment, describing the file.
2. `#include`s of standard and system headers (those enclosed in angle brackets e.g. `#include <stdlib.h>`), if any
3. `#include` of this module's header file.
4. `#include`s of all other project-related header files, if any
5. `#define` statements for local constants and macros, if any
6. `typedef`s, `struct` and `enum` declarations, if any
7. Declarations (prototypes) for local functions, as per rule G.9.
8. Definitions of global variables (which have been declared `extern` in the corresponding header file, as per rule G.4), if any
9. Definitions of local variables, as per rule G.6.
10. Definitions of global functions (executable code)
11. Definitions of local functions (executable code)

Do not put any variable definitions between function definitions.

Rule J.3 Place at least 2 blank lines between function definitions

Rule J.4 Indent code by four spaces for each level

Rule J.5 Do not use tab characters in the code for indentation

Set your editor so that the tab key automatically converts to the relevant number of space characters at the point of typing.

This rule is imposed to ensure that when your code is listed or printed using a program other than your original editor, it will still appear the same. (This particularly applies to viewing code within a terminal window).

Rule J.6 Place white space on each side of an assignment operator

Each of the assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `~=`, and `!=` shall always be preceded and followed by at least one space. (More than one space may be desired for alignment purposes.)

Rule J.7 Place white space on each side of a binary operator

Each of the binary operators `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `<<`, `>>`, `&`, `|`, `^`, `&&`, and `||` shall always be preceded and followed by at least one space. (More than one space may be desired for alignment purposes.)

Rule J.8 Don't put white space between a unary operator and its operand

Each of the unary operators `+`, `-`, `++`, `--`, `!`, and `~` shall be written without a space on the operand side.

Rule J.9 Items in parentheses or brackets shall have no whitespace before the first and after the last item

For example:

```
function(item1, item2, item3)       preferred
function( item1, item2, item3 )     not preferred
array[index]                        preferred
array[ index ]                      not preferred
```

Rule J.10 Restrict your code to 80 characters in width

This rule is imposed for three reasons:

1. to save the reader of your code from having to scroll sideways back and forth to understand your code. He or she may not have the same width of window that you do.
2. to make it easier for teams to do code reviews, which may well be done using printouts of the code. Printed code becomes very hard to read if lines wrap around, destroying the structured indentation.
3. should you be comparing two versions of code in side-by-side windows, it is much easier to do if the code does not stretch out to the right.

Rule J.11 Use a space between the `if`, `for`, `while`, and `switch` keywords and the opening parenthesis

```
if (condition)      preferred
if(condition)       not preferred
if( condition )     not preferred
```

These statements should be visually distinct from function calls.

Rule J.12 Use no space between a function name and its opening parenthesis

```
doStuff(arg);       preferred
doStuff (arg);      not preferred
```

This is the corollary of rule J.11.

Rule J.13 Don't use so-called "Yoda syntax" in conditional expressions

In C it is easy to mistakenly write code such as

```
if (value = CONSTANT) { ... }
```

where what was intended was

```
if (value == CONSTANT) { ... }
```

Because the first form is valid C code (but not what was intended) the bug that has been introduced can be subtle, and hard to diagnose. For this reason some people recommend using the form

```
if (CONSTANT == value) { ... }
```

instead, because a compilation error will occur if "=" is entered instead of "==". To a reader, this is very strange syntax, which is why others have dubbed it "Yoda syntax" after the speaking style of the character in the Star Wars movies. And it does not solve the problem if both terms in the conditional expressions are variables, such as

```
if (variable1 = variable2) { ... }
```

So don't use this syntax. Instead, use a compiler warning to alert you to this problem instead.

If using gcc or Clang, the relevant warning flag is (obscurely) called `-Wparentheses`, (or specify `-Wall`, which includes this flag). See also rule K.2

Rule J.14 Use the so-called "one true brace style" for `if`, `for` and `while` statements

This is the style of putting an opening brace on the same line as the `if`, `for` or `while` expression, but putting the closing brace on a line by itself. For example

```
if (condition) {
    statement(s);
    ...
} else {
    statement(s);
}
```

This is a conventional style within the C community, even though it looks very odd when first encountered. An alternative style is to put braces on lines by themselves, such as:

```
if (condition)
{
    statement(s);
    ...
}
else
{
    statement(s);
}
```

The title "one true brace style" hints that devotees of the first style, who coined the term, are rather dogmatic about it. But there is a practical reason for using it. Because we insist on rule J.15 below, code can end up with a lot of extra white space if every instance uses the second form. Too much white space (i.e. code too spread out) can make code harder to understand, just as too little white space can.

Exception: See rule J.16.

Rule J.15 Always use braces with `if`, `else`, `for`, `while`, `do` and `switch` statements

Although the compiler is perfectly happy with no braces if an `if`, `for` or `while` condition is followed by a single statement only, the aim of this rule is to reduce the chances that any future modification to the code will introduce a bug. This includes you inserting temporary debugging statements (or even worse, commenting out a statement), or someone else maintaining the software in future.

This is also MISRA-2004 rule 14.8 [1].

Rule J.16 Don't use the "one true brace style" when the `if` condition, the `while` condition, or the `for` loop specification takes up more than one line

In these cases, do put the "{" on a line by itself after all. It makes it easier for a reader to see where the test condition ends and the executable statements begin.

Example: don't do this:

```
if ((((!isEmergencyState) && (trackIntervalCode != 0) &&
    (baseStationTime_lc >= trackInterval_lc)) || (isEmergencyState &&
    (emergTrackIntCode != 0) && (baseStationTime_lc >=
    emergTrackInterval_lc))) {
    baseStationTime_lc = 0;
    fillTrackingPkt();
}
```

Rule J.17 In multi-line `if` and `while` conditions, use indenting to show the logic clearly

Look at the following example:

```
if ((((!isEmergencyState) && (trackIntervalCode != 0) &&
    (baseStationTime_lc >= trackInterval_lc)) || (isEmergencyState &&
    (emergTrackIntCode != 0) && (baseStationTime_lc >=
    emergTrackInterval_lc)))
{
    baseStationTime_lc = 0;
    fillTrackingPkt();
}
```

How long does it take you to work out what the logic is? Compare that with the layout below:

```
if (   (   (!isEmergencyState)
        && (trackIntervalCode != 0)
        && (baseStationTime_lc >= trackInterval_lc))
    || (   isEmergencyState
        && (emergTrackIntCode != 0)
        && (baseStationTime_lc >= emergTrackInterval_lc)))
{
    baseStationTime_lc = 0;
    fillTrackingPkt();
}
```

Clearly the second form is much easier for a reader to understand. Unfortunately, you will probably need to do this formatting manually — it is unlikely that any automatic formatting in your editor can do it.

Rule J.18 Don't use the "one true brace style" for function definitions

Place the opening "{" on a line by itself after the function declaration.

Rule J.19 Place the function-description comments of rule H.1 on lines between the function declaration and the opening "{" character.

Many people place the comments ahead of the function declaration, but this is harder to read, unless the comments themselves repeat the function name and its parameters. But repeating things in the comments is a bad idea — the more duplication of material in comments that exists, the higher the maintenance burden. Therefore the greater the risk that the code and the comments will get out-of-sync with each other.

See the example shown in rule H.2.

K Conditional expressions

Rule K.1 Use the forms `if (variable)` and `if (!variable)` only when `variable` is declared `bool`

The C language allows the syntax `if (variable)` to be used with any integer type, in which case it tests whether the value is zero or not. But greater clarity comes with not using this feature.

```
int     foo;

if (foo) { ... }          not preferred
if (foo != 0) { ... }     This is immediately clearer to a reader.
```

The idea that the first form is "more efficient" is not a justification for using it. It is a micro-optimisation at best, and compilers may well perform the optimisation anyway.

Likewise for pointers.

```
int*    bar;

if (bar) { ... }          not preferred
if (bar != NULL) { ... }  preferred
```

This rule applies to `while` and `for` conditions also.

This is MISRA-2004 rule 13.2 [1].

Rule K.2 Avoid the assignment operator (=) inside conditional expressions, where possible

This rule is to reduce the chances of bugs being introduced where = has been typed mistakenly instead of ==, a very easy mistake to make. If you simply must use this form, ensure that rule K.1 is obeyed. For example:

```
if (c = getNextChar()) { ... }
```

This is not preferred. Did you mean `if (c == getNextChar()) { ... }` instead? If you do intend an assignment operation, it is a little better to use more explicit syntax:

```
if ((c = getNextChar()) != '\0') { ... }
```

But even more preferred is:

```
c =  getNextChar();
if (c != '\0') { ... }
```

Enforcement: If using gcc, enable the warning `-Wparentheses` (or better, `-Wall`) to detect this problem. If using Clang, this warning is enabled by default.

L Other

Rule L.1 *A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer*

This rule is essential to the success of rule H.4 and suggestion B.1.

This is MISRA-2004 rule 11.5 [1].

Enforcement: If you are using gcc or Clang, enable the `-Wcast-qual` warning option to detect this problem.

5 Advice

The following are some additional suggestions to improve reliability, but they do not have the status of “rules”.

A Comments

Despite endless exhortations to programers not to write useless comments, I still see the following in production code.

```
if (notification_state.timestamp != 0xFFFFF) {
  notification_state.timestamp++;     // increment timestamp
}
```

Since it is so obviously a waste of space, why do people do it? I believe that students are inadvertently taught bad habits right at the point when they are first learning C. Unfortunately, students are presented with innumerable examples of comments within code fragments which merely explain how that particular feature of C actually works. So they learn to write comments like that.

The following suggestions, in conjunction with rules H.1 and H.2, are a minimal attempt to improve the situation.

Suggestion A.1 Use comments to answer the question “why” rather than “what”

Somebody coming to your code from outside is more usually wanting to know “why is the code doing this?” rather than “what is the code doing here?” For example, if you have found that a simple approach did not work and a more complex one is needed, then save the maintenance programmer from re-inventing the wheel by explaining why you chose the course you took.

Suggestion A.2 Use comments within code only to give information not already given by the code

Consider the following:

```
c = getNextChar();
if (c == '\0'){
    // empty string
    ...
}
```

The comment does not add a lot to the reader’s understanding. But if the comment gives us an insight into the higher level behaviour, it can be a great help. For example.

```
c = getNextChar();
if (c == '\0'){
    // the other device did not respond to our request
    ...
}
```

B Global data

In embedded software, global variables cannot be entirely avoided. They are a necessary evil, but an evil nonetheless, so use them as little as possible.

Suggestion B.1 As far as is practicable, implement global variables and structures using read-only pointers

As a further means of reducing the number of obscure bugs that might be introduced by misuse of global variables, try where possible to declare them so that only the module in which they are defined is capable of modifying them. All other modules are to get read-only access to them. For example, suppose you have a structure with two fields, which is defined within file samp1.c, but you want it to be read-only in all other modules. This can be done as follows:

In the header file samp1.h:

```
typedef struct {
    int    field1;
    double field2;
} Samp1_DataType;
extern const Samp1_DataType * const samp1_var;
```

In the C source file samp1.c

```
static Samp1_DataType samp1Var;
const Samp1_DataType * const samp1_var = &samp1Var;
```

This allows code within samp1.c to modify the fields of `samp1Var`, but all other modules cannot see `samp1Var`. But they can see the pointer `samp1_var`, which provides read access to the fields, but which prevents any modification.

(Note that this is not a substitute for the use of critical sections and mutexes to control access to data where that is required, it is just a way of preventing different modules fighting each other to set the values of global variables.)

C Compiler warnings

Compiler warnings are your friend, not your enemy. But you would hardly know it, given the culture and history of the C language. There has been a culture in the C programming community that has used the slogan “Trust the programmer” as a way of justifying a historical lack of compiler checking of dangerous or erroneous programming constructs, or the outsourcing of those checks to a “lint” program that is often not installed.

Like car drivers, it seems that 90% of C programmers consider themselves to be above average. They don’t make programming errors, so they don’t need warnings. As a result, compiler warnings are a dog’s breakfast. For example, take the popular gcc compiler:

- Its default behaviour (i.e. if no warning flags are explicitly specified on the command line) is to issue very few warning messages indeed.
- If you compile with the `-Wall` (so-called “all warnings”) option you will receive many useful warnings, but by no means all of the ones that you might actually find informative. It is horribly misnamed.

  Why would this be? The gcc documentation describes `-Wall` using the phrase

> *This enables all the warnings about constructions that **some users** consider questionable, and that are easy to avoid*

which is rather grudging, and hints that most people wouldn't *really* want all these warnings.

- If you compile with the `-Wall -Wextra` options, you will get more, but still by no means all the warnings that it would be useful to receive.
- To enable further warnings, you have to specify them individually. So you have to know the name of each potential warning flag, and decide whether it might be useful to you. And then, when a new version of the compiler is released, you have to do it all again, because there are new ones (which have not been included in `-Wall` or `-Wextra`).

To really find problems, I use the following gcc options (as at gcc version 4.9.3)

-Wall	-Wextra	
-Wbad-function-cast	-Wcast-qual	-Wcomment
-Wconversion	-Wdisabled-optimization	
-Wfloat-equal	-Wformat-nonliteral	-Wformat-security
-Winit-self	-Wlogical-op	-Wmissing-declarations
-Wmissing-format-attribute	-Wmissing-include-dirs	-Wmissing-prototypes
-Wold-style-definition	-Wpointer-arith	-Wredundant-decls
-Wshadow	-Wsign-conversion	-Wstrict-prototypes
-Wswitch-default	-Wswitch-enum	
-Wundef	-Wunused-macros	-Wwrite-strings

If you apply all these options to code written by someone else, it is likely that you will drown in warnings. But if you apply them to code you are writing from scratch, they will save you a lot of grief. The warnings that are likely to annoy you the most are the `-Wconversion` and `-Wsign-conversion` warnings, but these are a cue for you to consider what you really intend. See sub-section D for ways to make conscious rather than unconscious decisions about these conversions.

D Beware of type casts

In C, type casts are a can of worms. For example, consider the following

```
foo = (int16_t)bar;
```

What happens if the value of `bar` is out-of-range for the `int16_t` type? This depends on what type of variable `bar` is. If `bar` is an integer type, the result is well defined, if perhaps surprising. For example, if

```
int bar = 110000;
```

then `foo` will be -21072 (i.e. its sign will have changed), assuming your machine uses 2's complement arithmetic.

But if bar is a floating point type, e.g.

```
float bar = 110000.0f;
```

then the result of the type cast is actually undefined. You may get -21072, or you may not. Or, what if bar is a NaN? What should foo be set to?

Unfortunately, many uses of type casts seem to assume that “it’ll never happen”. The trouble is, many problems in software occur when things that “can’t happen”, do happen.

Here is a suggestion for dealing with the problem. Define type conversion functions, and call them rather than using a type cast directly. The idea is that you get in the habit of thinking about

(a) what, if any, range checking you want, and

(b) what, if any, error status information you want to get back from the conversion function.

Here is a macro and a couple of sample inline functions that I have used.

```
#define myTC_CLAMP(to_type_, lo_, hi_, val_, res_, err_)  do {   \
    int         stat__;                         \
                                                \
    stat__ = ERANGE;                            \
    if ((val_) > (hi_)) {                       \
        res_ = (hi_);                           \
    } else if ((val_) < (lo_)) {                \
        res_ = (lo_);                           \
    } else {                                    \
        stat__ = 0;                             \
        res_ = (to_type_)(val_);                \
    }                                           \
    if (err_ != NULL) { *err_ = stat__; }   \
} while (0)
```

Note that its first argument is not actually a value, but a type name. This is unorthodox, but it works. Here are a couple of the inline functions that may use this macro.

```
static inline int16_t tc_itoi16Unsafe(int value)     { return (int16_t)value; }
static inline int16_t tc_i32toi16Unsafe(int32_t value)
                                                     { return (int16_t)value; }

static inline int16_t tc_itoi16(int value, int *error)
{
    int16_t     result;

    myTC_CLAMP(int16_t, -32768, 32767, value, result, error);
    return result;
}

static inline int16_t tc_i32toi16(int32_t value, int *error)
{
    int16_t     result;

    myTC_CLAMP(int16_t, -32768, 32767, value, result, error);
    return result;
}
```

So the idea is that if I need to cast an `int` value to an `int16_t`, I don’t use the cast directly, but consider what I want to happen if the value is out of range. If I really don’t care, I call `tc_itoi16Unsafe()`, which does the cast. If I do care, I call `tc_itoi16()`. I can separately decide whether I care about the error status or not. I find this a useful discipline.

There is a very similar (but slightly simpler) macro for unsigned values.

6 The general.h header file

This file provides a number of definitions that are of general use. These include the `bool` and size-specific integer data types, the `static_assert()` macro, and other useful macros.

If your compiler is a C89/90 compiler, the size-specific integers types are defined manually with typedefs, but a form of the `static_assert()` macro is defined and used to check those definitions for correctness. If you get a compilation error from these macros, you will need to edit the typedefs to match your compiler's integer sizes. For C99 and later compilers, this is not required.

```
#ifndef GENERAL_H
#define GENERAL_H
/*============================================================================*/
/*!\file
 * \brief general.h - definitions of general use to (standard) C programs
 *
 * \author  David Hoadley <vcrumble@westnet.com.au>
 *
 * \details
 *      Definitions that almost all of our C programs can find useful. Defines
 *      bool and fixed size int data types, if the compiler standard does not
 *      (i.e. if this is a pre-C99 compiler). Defines a static_assert() macro
 *      if this is a pre-C11 compiler. Some other useful constants and macros
 *      are defined, including macros for assertions in an embedded context.
 *
 *============================================================================
 */
/* Which C standard(s) does this compiler support here? */
#if defined(__STDC__)
#  define PREDEF_STANDARD_C_1989
#  if defined(__STDC_VERSION__)
#    define PREDEF_STANDARD_C_1990
#    if (__STDC_VERSION__ >= 199409L)
#      define PREDEF_STANDARD_C_1994
#    endif
#    if (__STDC_VERSION__ >= 199901L)
#      define PREDEF_STANDARD_C_1999
#    endif
#    if (__STDC_VERSION__ >= 201112L)
#      define PREDEF_STANDARD_C_2011
#    endif
#    if (__STDC_VERSION__ >= 201710L)
#      define PREDEF_STANDARD_C_2018
#    endif
#  endif
#endif

/*----------------------------------------------------------------------------
 *
 * Possibly missing type definitions
 *
 *----------------------------------------------------------------------------*/
#ifndef PREDEF_STANDARD_C_1999

typedef int     bool;           /* C89/90 lacks a proper Boolean data type */
#define false   (bool)(0)
#define true    (bool)(1)

/* In embedded systems, we are implementing code on processors where the word
 * length is critical to the maths we implement. The size of char, int and long
 * int varies from processor to processor (and/or compiler to compiler). In
 * order to get 8, 16 and 32-bit numbers when we want them, declare variables
 * using the following six typedefs. Change the definitions of these typedefs
```

```
 * here, according to your compiler.
 *   If our compiler supports the C99 standard and we include the <stdint.h>
 * header file, these types are already defined. */
typedef signed char         int8_t;
typedef unsigned char       uint8_t;
typedef signed short int    int16_t;
typedef unsigned short int  uint16_t;
typedef int                 int32_t;
typedef unsigned int        uint32_t;

#else
#include <stdbool.h>        /* for bool, true and false */
#include <stdint.h>         /* for intX_t and uintX_t types */
#endif

/* How do we know that our typedef's above are correct? We need a way of
 * checking this at compile time with a test that will cause a compilation error
 * if it fails. This is a case for a static assertion.
 *     If our compiler supports the C11 standard, we have access to the
 * _Static_assert(expression, message) built-in function to do exactly this, and
 * if we include <assert.h>, there is a macro "static_assert()" already defined
 * (= _Static_assert()).
 *     But of course, if this is a C89/90 compiler, this macro does not exist.
 * So it is useful to define one here. Similarly a C99 program can use it.
 * Here is that definition. Unfortunately it is not quite as smooth as the real
 * static_assert() introduced in C11, because it generates two irrelevant error
 * messages:
 *  "warning: division by zero [-Wdiv-by-zero]"
 *  "error: enumerator value for ‘static_assert_nnn’ is not an integer constant"
 * but the location of the error will tell you what the problem is.
 *     If you are using gcc, you might find that you are drowning in "note:"
 * messages each time a static assertion fails. If so, compile with the option
 * -ftrack-macro-expansion=0 */
#ifndef PREDEF_STANDARD_C_2011

/* Strange token-pasting magic here */
#define CONCAT(a, b) a ## b
#define EXPCONCAT(a, b) CONCAT(a, b)
#define static_assert(e, msg) \
        enum { EXPCONCAT(static_assert_, __LINE__) = 1/!!(e) }

#else
#include <assert.h>
#endif

/* Now that we have defined static_assert(), use it to make sure that our
 * typedefs of fixed size integers are actually correct for this machine
 * architecture. (This is only a problem with C89/90 where we have typedef'ed
 *  them ourselves) */
#ifndef PREDEF_STANDARD_C_1999
static_assert(sizeof(int16_t) == 2,  "typedef of int16_t is wrong");
static_assert(sizeof(uint16_t) == 2, "typedef of uint16_t is wrong");
static_assert(sizeof(int32_t) == 4,  "typedef of int32_t is wrong");
static_assert(sizeof(uint32_t) == 4, "typedef of uint32_t is wrong");
#endif


/*----------------------------------------------------------------------------
 *
 * Peculiar design decisions in the C language
 *
 *----------------------------------------------------------------------------*/

/*! C has some very silly default behaviours. One of these is to define all
 *  symbols declared at file level as global in scope (i.e. visible to the
 *  linker). This is a recipe for trouble. Secondly, C overloads the 'static'
 *  storage class (which defines how data is allocated to memory) with a second
 *  completely independent meaning - that the scope of the symbol name will be
```

```
 *  restricted to the current C source file only. While this is behaviour that
 *  we want, the use of the word 'static' is odd, to say the least. So, use the
 *  following definitions to discipline yourself to sensible behaviour - define
 *  truly global variables and functions as 'GLOBAL', and those which are to be
 *  visible only in the current C source file as 'LOCAL'.
 *  [Each global function definition should be matched with a corresponding
 *   function prototype in the corresponding header file, and each global
 *   variable definition should be matched with a corresponding 'extern'
 *   declaration in that same header file. There should be no 'extern'
 *   declarations outside of header files.] */
#define LOCAL   static
#define GLOBAL                  /*!< See above */

/* Another odd design decision was the requirement to insert the keyword "break"
 * after every element of a switch-case construct if you did not want to run on
 * and execute the code of the next element. That is, most of the time you must
 * insert the word "break", and only occasionally do you write code in which you
 * don't want it. So on average, if a "break" is missing in a case statement, it
 * is a bug. So, on those rare occasions when we do intend to run on, we want to
 * let people know that, this time, it isn't a bug. */
#define NOBREAK     /*!< Show reader that 'break' was not omitted by accident */
#define FALLTHROUGH /*!< or if you prefer, use this word instead. */


/*---------------------------------------------------------------------------
 *
 * Useful arithmetic constants
 *
 *---------------------------------------------------------------------------*/

/* A few definitions below are followed by an empty comment containing "!< . "
   and nothing else. This is just to get Doxygen to include the definitions in
   its lists and cross-references. But you don't need me to explain what each
   one is, do you? */
/*      Double precision constants */
#define PI          3.14159265358979323846264338327950288419 71  /*!< , */
#define HALFPI      (PI / 2.0)                                    /*!< , */
#define TWOPI       (2.0 * PI)                                    /*!< , */
#define SQRT2       1.41421356237309504880168872420969807856 97  /*!< , */
#define SQRT3       1.73205080756887729352744634150587236694 28  /*!< , */

/*      Angle conversions */
#define DEG2RAD     (PI / 180.0)        /*!< degrees to radians */
#define RAD2DEG     (180.0 / PI)        /*!< radians to degrees */

/*!     A very small number, used to avoid divide by 0 errors */
#define SFA         1E-10

/*---------------------------------------------------------------------------
 *
 * Other useful macros and inline functions
 *
 *---------------------------------------------------------------------------*/
/*      Convert angle from one unit to another */
#ifdef PREDEF_STANDARD_C_1999
/*          Compiler supports inline functions */
/*! Returns \a angle_deg converted from degrees to radians */
static inline double degToRad(double angle_deg) { return angle_deg * DEG2RAD; }
/*! Returns \a angle_rad converted from radians to degrees */
static inline double radToDeg(double angle_rad) { return angle_rad * RAD2DEG; }

#else
/*          C89/C90 compiler only - no inline functions. Need macros instead */
#define degToRad(angle_deg__)       ((angle_deg__) * DEG2RAD)
#define radToDeg(angle_rad__)       ((angle_rad__) * RAD2DEG)

#endif
```

```
/*! Because C passes arrays to functions by passing only a pointer to the
 *  zero'th element of the array, we often need to pass the array size in a
 *  separate argument to the function. Make that easier. */
#define ARRAY_SIZE(x__)             (int)(sizeof(x__)/sizeof(x__[0]))

/*! There are times when we want to declare arrays or buffers whose dimension is
 *  a power of 2, so that the index value can be bit-masked to enforce wrap-
 *  around. Code based on this assumption will fail if the array size is changed
 *  by someone who doesn't notice this requirement.
 *  Here is a check you can use to help prevent that happening */
#define ISPOWER2(x__)               (!((x__)&((x__)-1)))

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/*!     Useful macros for supporting a "Design by Contract" approach to
        programming embedded systems. These are heavily influenced by the
        article "Design by Contract (DbC) for Embedded Software" by Miro Samek,
        to be found on the Barr Group website, at
        https://barrgroup.com/Embedded-Systems/How-To/Design-by-Contract-for-
Embedded-Software

        To use,
        1. define an implementation of onAssert__() function as specified below.
           For example, it could write the first n chars of the filename and the
           line number to a bit of memory that won't be erased on a reboot, and
           then force a reboot.
        2. spread REQUIRE(), ENSURE(), and INVARIANT() macros throughout your
           code. (In brief, REQUIRE() macros are generally put at the beginning
           of functions to check the validity of passed parameters. ENSURE()
           macros are generally placed at the end of functions to check that the
           functions have not calculated invalid results.)
        3. put the DEFINE_THIS_FILE statement near the top of each C file that
           contains any REQUIRE(), ENSURE(), or INVARIANT() macros.
        4. choose the ASSERTION_LEVEL that you want.

        (If you define USE_STANDARD_ASSERT as described below, you don't really
        need to do steps 1 and 3. They are really for embedded systems.)
*/
//#define ASSERTION_LEVEL     0       /* DbC assertions are all disabled */
//#define ASSERTION_LEVEL     1       /* Pre-conditions checked (only) */
#define ASSERTION_LEVEL     2       /* Pre & post-conditions checked */
//#define ASSERTION_LEVEL     3       /* Everything checked */

/* Uncomment the following line if you want to use the standard assert() macro,
 * and NOT use your own onAssert__() function to implement these DbC macros.
 * Comment it out if you want to create an onAssert__() function, or if assert.h
 * is not available.  */
#define USE_STANDARD_ASSERT

#if (ASSERTION_LEVEL == 0)
/*!    Leave DEFINE_THIS_FILE undefined. But we get a -Wextra-semi warning
 *     message if it is defined as nothing at all, so use this dummy
 *     declaration instead. */
#  define DEFINE_THIS_FILE    typedef int dummy_t
#  define ASSERT(ignore_)     ((void)0)     /*!< all assertions disabled */

#else

#  if defined(USE_STANDARD_ASSERT)
#    include <assert.h>
     /*!    Leave DEFINE_THIS_FILE undefined. But we get a -Wextra-semi warning
      *     message if it is defined as nothing at all, so use this dummy
      *     declaration instead. */
#    define DEFINE_THIS_FILE    typedef int dummy_t
#    define ASSERT(test_)       assert(test_)   /*!< Uses standard assert */
#  else
#    ifdef __cplusplus
     extern "C"
     {
```

```
#    endif
        /*! callback invoked in case of assertion failure */
        void onAssert__(const char *file, unsigned line);
#    ifdef __cplusplus
     }
#    endif

/*!       Ensure exe image contains one copy of file name, not one per ASSERT */
#    define DEFINE_THIS_FILE    static char const THIS_FILE__[] = __FILE__
#    define ASSERT(test_)       ((test_) ? (void)0  \
                                         : onAssert__(THIS_FILE__, __LINE__))
#  endif
#endif

#if (ASSERTION_LEVEL >= 1)
#  define REQUIRE(test_)      ASSERT(test_) /*!< Check preconditions */
#else
#  define REQUIRE(test_)      ((void)0)     /*!< Pre-condition checks disabled*/
#endif

#if (ASSERTION_LEVEL >= 2)
#  define ENSURE(test_)       ASSERT(test_) /*!< Check post-conditions */
#else
#  define ENSURE(test_)       ((void)0)     /*!<Post-condition checks disabled*/
#endif

#if (ASSERTION_LEVEL == 3)
#  define INVARIANT(test_)    ASSERT(test_) /*!< Check invariants */
#else
#  define INVARIANT(test_)    ((void)0)     /*!< Invariant checks disabled */
#endif

#if !defined(ASSERTION_LEVEL) || (ASSERTION_LEVEL < 0) || (ASSERTION_LEVEL > 3)
#  error "ASSERTION_LEVEL must be set to one of 0, 1, 2, or 3"
#endif

/* Finally, one more piece of assertion checking that is particularly relevant
 * to C code. Unlike C++, C does not allow passing parameters to functions by
 * reference. It allows passing by address/by pointer, but this is not exactly
 * the same thing. It is quite possible to pass NULL to any parameter in C that
 * is expecting an address or pointer, and the compiler will not object. After
 * all, sometimes functions check for NULL pointers and take different actions
 * if one is passed. But all too often, functions cannot handle getting a NULL
 * pointer, and will crash if one is passed. An assertion check helps here.
 *      Uncomment the following line if you want to perform NULL checking.
 * Comment it out if you want to suppress NULL checking. */
#define ENABLE_NULL_CHECKING

#ifdef ENABLE_NULL_CHECKING
#  include <stddef.h>                       /* Make sure NULL is defined */
#  define REQUIRE_NOT_NULL(pointer_)  ASSERT((pointer_) != NULL)    /*!< , */
#else
#  define REQUIRE_NOT_NULL(pointer_)  ((void)0) /*!< Pointer checking disabled*/
#endif

#endif /*GENERAL_H*/
```

7 References

[1] MISRA (2004) MISRA-C:2004 – Guidelines for the use of the C language in critical systems, edition 2. Motor Industry Software Reliability Association, UK.

[2] Barr Style Guide, https://barrgroup.com/embedded-systems/books/embedded-c-coding-standard

[3] Google Java Style Guide, https://google.github.io/styleguide/javaguide.html#s5-naming

[4] C++ Programming Style Guidelines. http://geosoft.no/development/cppstyle.html