Coding Conventions

For programming in C

Introduction

Many organisations have coding guidelines, so why define yet another set of coding guidelines?
Well, first of all, no two sets of coding guidelines are the same; my intention is to collect all issues I think useful into one article. Being a free-lance consultant, I may have seen more different coding guidelines in my professional career than some. And then again, maybe I haven't...
Secondly, voicing ideas concerning coding guidelines I am often asked to provide a document of this sort. Until now I haven't really bothered to really sit down and describe these ideas and practical tips. So I couldn't hand over anything in writing. This has now changed.
And lastly, I have seen many coding guidelines listing the "do's" and "don'ts", but without any explanation. This makes it difficult to reassess the guidelines from time to time, but may also be detrimental as far as acceptance among the developers is regarded.
Since most of my development work is done in assignment for different customers, I hardly ever get the chance to live by my own rules exclusively. However, if my customer's rules permit, I always add the rules I have collected here to my requirements. I do not expect anyone to take this document verbatim, and use it as a coding guideline. The document structure has never been conceived for this usage (see also the next section).
But if someone reading this document gets a few ideas for his/her own guidelines (or company guidelines), this document has fulfilled its purpose.

Document structure

The main reason for this document is the explanation why a certain guideline has been created. For that reason I have chosen a top-down approach. I describe a certain way of structuring your software and from that I more or less derive the guidelines: From components and sub-components to files to functions to statements…
If you have a different way of organizing your development, your guidelines may come out different. But that's exactly the reason why I emphasize the reasoning behind a certain set of rules. You may follow the reasoning given your own structure, and derive a somewhat different set of rules, depending on your own prerequisites.

What about C++?

Good question. Most of the top-level guidelines contain nothing specific to C, only bound to the way C and C++ (and many other languages) organize software into files and files into directories. Additionally, C++ uses many of the same structures C offers, so the same reasoning may apply, and the same guidelines could be given.
The main reason not to concentrate on C++ is twofold: First, if your projects have reached the state where C++ is the main implementation language, your projects will likely be somewhat larger and more developed. In my experience, such projects already have a set of guidelines, even if usually concentrating on naming and the C aspects of the language. Related to that, many smaller embedded projects (not necessarily less complex, just smaller; less than 500 kLOC) live without explicit conventions, and then grow to a size where the need for a set of guidelines is felt. In such situations a certain amount of guidance can be most welcome.
That said, this article has not been written to include C++, but if you're mainly involved in C++ projects you still might benefit from reading this.

What about C99?

The same argument applies here. C99 has not yet arrived in the embedded development circles I visit regularly. Yes, C++-style comments have been adopted quickly, but full C99 implies a lot more. As for C++, this article has not been written with C99 in mind, but you may very well benefit from reading.

Definitions

In discussions about coding guidelines one issue reappears every time: Definitions!
What is meant by a component? Is a directory the same as a component? What is meant by a module? Is it the same as a component? These are the definitions I have used in this document, unless noted otherwise.

Component – a component is a software unit built around one or more interfaces. The component defines and implements the interface, and offers a self-contained declaration of the interface for use in other components. A component may be as simple as a single header file, for instance if the component only defines some inline functions, or as complex as containing a set of implementing C files in addition to several sub-components.

Sub-component – a sub-component is a component with all properties a component has. However, in the context used, the sub-component is considered a child of the parent component. The interface published by the sub-component is used only by its sibling sub-components and their common parent-component. If discussed in isolation, a sub-component may be called a component again.

Module – a module is an undefined entity in this document. Because of the different associations used in the industries, the idea of a module may mean different things to different people. For that reason I prefer to talk about components, which to you might mean the same.

Translation unit – a translation unit is a C/C++ source file recursively combined with all necessary header files. A translation unit is what a compiler gets to see when (successfully) compiling a source file.

Component structure

I tend to refrain from using the term module because it seems to mean different things to different people. Some use it like I define component, some use it at a hierarchical level below component, and others use it to mean just a source file together with its associated header file.
To me a component is an entity with one or more clearly defined interfaces to the world outside the component. Outside entities can interact with the component only through one of its interfaces, whereas everything not exported through such an interface is only visible from within the component, independent from the fact whether the language actually forbids usage of such elements.

1. A COMPONENT OFFERS AND IMPLEMENTS ONE OR MORE INTERFACES

The basic idea behind a component is to have a set of requirements that are to be fulfilled by a certain entity. This set of requirements defines a certain amount of input data as well as a set of output data. The link between the two is the requirements themselves.
In order for a component to perform the so defined functionality, interfaces link the component to the outside world. So a component offers and implements an interface.
This definition doesn't tell us anything about decomposition, but it puts quite a strict harness of encapsulation on our code.
Why would we allow a component to offer more than one interface? Having a requirement that just one interface is to be exported by a component seems to create a simpler world. However, in some cases a component supervises a set of data with different meaning to different clients: An HR department needs to know more about employees than a colleague shall have access to; an operator at a nuclear power plant shall have less access to the controlling equipment than a lead engineer may need.
In such cases encapsulating all data with just one public interface may seem like a bad idea. Of course it may make sense to have one small general interface with one or more specialized ones, where the specialized ones contain the general interface. But at least we'll have more than one interface.

2. EACH COMPONENT HAS ITS OWN DIRECTORY

This is an issue of what I call "physical architecture". Of course one can disperse the files belonging to one component, but finding your way becomes cumbersome.
You can also advocate the use of one directory for more than one component. For small projects this may be feasible, and in many cases the build systems do not support multiple directories well, if at all.
However, the scalability of such projects is severely limited. If you then create a project with one source directory with 150 components and more than 700 files, limits are clearly behind you. And converting such a project into a directory-based structure might be more than you bargained for.

3. EACH INTERFACE IS EXPORTED USING ONE SELF-CONTAINED HEADER FILE

Here are a few mixed requirements. If I am to be the user of one component's interface, I want to include the corresponding definitions, data structures and interface functions. If there are multiple header files involved because of a certain partitioning on the side of the component's creators, it is easy enough to create a small header file at a higher level, including all the header files belonging together. The component's author is in a better position to decide what belongs together and what not.
Additionally, I want to interface to be self-contained. If the component's interface needs to use other interfaces I may or may not yet be a user of, I make it the component's duty to indirectly provide those interfaces to me by including those interfaces in its interface. If I do not need those interfaces directly, I may not know that those interfaces have been made available, and I will not use them. In C++ such interfaces may even remain at least partially hidden. It would be more than cumbersome (doable, but cumbersome) if I had to read some extra description, and include the interfaces of one or more components myself, possibly even in a certain order!

4. INTERFACE HEADER FILES ARE COLLECTED IN A DIRECTORY 'INCLUDE' ONE LEVEL UP FROM THE COMPONENT'S DIRECTORY

This again is just a convention from someone who's been tasked with the conception of build systems more than once. The name for the header directory is irrelevant, as long as a convention is used and adhered to. Also the position within a directory hierarchy can be debated. But the principle to collect all interfaces at a certain hierarchical level into one directory is not to be debated. Neither is the use of an identical name for each hierarchical level. That way, the basic include path for compilation can be constructed (semi-) automatically in a Makefile.
If we do not want such a directory to be created, in the extreme each component may have to adapt its build configuration as soon as a new interface is to be used.
An extra issue to address is the "one directory per component" requirement. In this requirement we seem to violate one of our own requirements, and yes, we do: The interface file belongs to a component but does not reside in the component's directory. If such an exception is not to be tolerated (only for the interface files!), a solution can be to put stub header files into the header directory, only pointing to the real interface header file residing in the component's directory.

5. EACH COMPONENT SHALL HAVE A 4-LETTER IDENTIFIER DESCRIBED IN THE SOFTWARE PROJECT DOCUMENTATION

This is a matter of debate. The idea is to augment globally visible elements (macros, types, variables, functions) with such an identifier, to be able to see where the element is coming from. If you enforce a clear project structure, such an augment may not be necessary. Especially since the modern IDEs offer enormously powerful features assisting in finding the elements of your code, definitions, declarations and all references, it might be detrimental to use prefixes.
I have seen one occasion where such identifiers were used as postfixes, in order not to hamper the code-completion feature of the used IDE.

6. Each component's files start with the component-ID followed by an underscore

I am no fan of such a rule. If your software is decently distributed across a set of directories, such a file-naming convention seems superfluous to me. But if it is unavoidable to have all or at least many source files in one directory, such a naming convention may help you find the relevant sources more quickly.

7. Components can be grouped in further directory levels

Architectural decomposition of a system usually results in several hierarchical levels. These levels shall be reflected in the directory depth of the respective components. Of course the interfaces of the various levels shall be made to match the decomposition; and if a sub-component offers an interface that is intended to also be used at other (parent) levels, such interfaces automatically become interfaces of the parent level. Whether an interface translation is necessary, is up to the structure and the naming conventions used, but at least the interface header file shall be made available at the appropriate level.

8. Groups are to be considered "super-components" and have their own specific ID

A grouping of components at a certain hierarchical level shall always result in a new component, even if the so-created component only offers its constituent's interfaces one-to-one. If the naming conventions require it, this includes adding additional (renamed) interfaces. If this is not the intention, consider to ungroup the components.

Header file handling

9. No two files shall have identical names

In really large, distributed systems this may present a problem. It may also be a problem if multiple libraries are being used. But in normal cases the reasoning is valid.
When all header files have differing names one can be sure that an include directive will either find the intended header file or produce a compiler error.
When all source files have differing names, the linker will have a chance to link all intended components into the intended executable.
As said, exceptions may be necessary and unavoidable, but then these issues must be handled differently (carefully maintaining include paths, partitioning the executable into multiple libraries before submitting them to the linker), thereby increasing the complexity.

10. Use internal include-guards

The reasoning behind include-guards is most likely undisputed. The intention is to prevent multiple inclusions of individual header files. This is necessary to maintain a valid translation unit, since the language does not in all circumstances allow double definitions or even declarations of certain entities, e.g. type definitions or macros.
In certain cases, this rule must be violated. In such cases extensive comments are required to illustrate the exact intended usage.
The creation of such include-guards is partly an issue of language standards, partly of project dependencies and partly of taste.
I normally use the following format:

```
#ifdef  _included_<basename>_<extension>_
#define _included_<basename>_<extension>_ 1
...
#endif /* _included_<basename>_<extension>_ */
```

Macros starting with double underscores or with one underscore and a capital letter are reserved by the language standard. Macros starting with a capital letter are customarily used in regular code. And in order to identify all macros used for include-guards with a regular expression, I tend to use the form presented above.
As an example, for a file called *MyIncludeFile.h* I would create the include-guard `_included_MyIncludeFile_h_`.

Please note the '1' in the definition. This is to avoid problems when someone forgets to use `#ifdef` and uses `#if` instead. You may consider using an undefined macro, something like `_h_file_included_`. A clever definition of that macro may even produce a compiler error in such cases.

11. USE EXTERNAL INCLUDE-GUARDS

This again is a controversial topic. In my experience it makes sense to bracket each include directive with an external include-guard as used in the header file to be included. To take up the above example again:

```
#                           if !defined(_included_MyIncludeFile_h_)
#include "MyIncludeFile.h"
#                           endif /* _included_MyIncludeFile_h_ */
```

The indentation added here does not quite match the indentation I recommend to use (I normally use 40 spaces), but the intention is not to clutter a whole list of include directives with the external include-guards. In this way the include-guards are located to the right of the column with include directives, so you can still get an easy overview of the header files used.
The rationale for these external include-guards is to limit the number an include file is read. Internal include-guards prevent the language content of a file to be processed multiple times. However, the compiler has to process the file each time it is included, look for and process `#ifdef` and `#endif` constructs, only to find that the relevant content of the file is zero. With the external include-guards the compiler will not even process the include directive, and doesn't have to open, process and close the file. On some platforms opening and closing a file for reading, especially on a network, is a rather expensive operation, so this may save quite a lot of build time.
As an alternative, the Microsoft-initiated `#pragma once` directive is used more and more often. And yes, it basically fulfils the same purpose. However, not all compilers support this pragma, it is a non-standard (albeit standard-conforming) extension, and the implementation of its functionality is up to the tool authors. For me it is a question, whether a second opening and closing of the file is really prevented whenever the second include directive is seen, or if the file's processing is halted only after reading the `#pragma once` for the second time. Also, when the first include directive is using a (relative) pathname to specify the file, whereas the second include directive uses no or a different pathname, is the implementation intelligent enough to discover that and act accordingly, or does it process both variants as if they were different files.
Since I have no time to test these kinds of implementation variants for each and every compiler and -version I am required to use, I prefer the external include-guards.

12. EACH C-FILE SHALL DEFINE A MACRO TO IDENTIFY ITSELF TO INCLUDED FILES

This is a rule that by itself has no rationale. It only makes sense in combination with other rules. The idea behind it is for header files to be able to differentiate between its inclusion as an interface to be used by other components, or as a header file for its own implementation. What you do with this knowledge is up to you, but do have a look at rule 13 below for an example.
The principle is simple: Each C file provides at the top of the file (just below the initial comment) a macro definition that is unique within the project. Its composition follows the same rules as for include-guards, refer to rule 10, `_file_<basename>_<extension>_`.

```
#define _file_MyCSourceFile_c_ 1
```

Again note the '1' for the definition. Here too you might use a definition like `_c_file_being_processed_` or something along those lines. For a rationale again refer to rule 10.

13. USE _LOCAL_EXPORT_ TO DEFINE INTERFACE ELEMENTS

Here we must take a look at the keyword `extern` and its uses.
We have to differentiate between variables and functions, so let's start with functions.
Many environments require all functions to have proper prototypes (e.g. MISRA); in C++ prototypes are even required. But C90 also know the concept of external declarations, basically performing the same functionality. Even the syntax of both is identical except for the storage-class specifier `extern`. Here's an example:

```
int MyFunction(char cMyChar);   /* prototype */
extern
int MyFunction(char cMyChar);   /* external declaration */
```

From a semantic point of view, these two are different: The prototype serves to declare the name `MyFunction` to signify a function taking one `char` as a parameter and returning an `int`, as does the external declaration. However, the `extern` keyword explicitly signifies external linkage. To comply with the understanding of several different compilers I therefore try to use the keyword `extern` whenever declaring a function defined outside of the current compilation unit and to provide a prototype without the keyword when compiling the defining compilation unit.[1]
To ensure consistent definitions, both are generated using one declaration, where the `extern` keyword is provided or not based on the file identification from rule 12. Later we'll see how.
For variables the situation is similar but not identical. The issue here is the concept of tentative definitions. For this reason and to accommodate older compilers I prefer to also make the storage-class of interface variables explicit using the extern keyword in foreign translation units only.
But for variables another issue arises here: Not all compilers accept the practice of providing an initializer with the definition of a variable with explicit external linkage. To accommodate initialization of such variables we will need another macro eliminating the initialization when included by a foreign compilation unit.
This is how it can be done: First I'll show an interface header file:

```
/**
 * Top level command of MyInterface.h
 */

#if !defined(_included_MyInterface_h_)
#define      _included_MyInterface_h_ _h_file_included_

/* Put all include directives here */

#                       if !defined(_INC_STDIO)
#include <stdio.h>
#                       endif /* _INC_STDIO */
#                       if !defined(_included_MyOtherInterface_h_)
#include "MyOtherInterface.h"
#                       endif /* _included_MyOtherInterface_h_ */

/* From here onwards, no include directives shall be used! */
/* Differentiate extern declarations and prototypes */
#ifdef _file_MyInterface_c_
#define _local_export_
#define _local_initialize_(_parm) = _parm
#else
#define _local_export_ extern
#define _local_initialize_(_parm)
#endif /* _cfile_MyInterface_h_ */

_local_export_
unsigned char cMyExternal
  _local_initialize_(5);

_local_export_
int MyFunction(char c);

/* Make sure _local_export_ is not carried over into the next
include */
#undef _local_export_
#undef _local_initialize_

#endif /* _included_MyInterface_h_ */
```

If now the C file ***MyInterface.c*** defines "its own" macro `_file_MyInterface_c_`, the component will have a prototype and an initialized global variable thanks to its corresponding

[1] Most current compilers do not need this distinction, since prototypes within file-scope are defined to have external linking.

include file. If a foreign compilation unit includes this interface, both the function `MyFunction` and the variable `cMyExternal` are declared to be introduced from a different component.
OK, this is quite an elaborate scheme. Yes, I have used it (slightly modified) to write libraries for publication. No, I wouldn't use it for casual programs to be written.
But if you're serious about a compartmentalized architecture, if you cannot foresee which, possibly older, compilers you will have to use in the future, at least give it serious consideration.

Layout

Layout discussions are fun. For about 5 seconds.
Opinions differ, especially on style. Therefore I have no rules on layout, period.
If you take over a piece of software from a colleague, use a beautifier with your settings and the code is yours.
If you do a quick fix in a component of someone else, stick to her/his layout. My editor can do that automatically, but even by hand a few lines of code in a different style will not hurt much.
I have only one rule concerning maintenance of a given layout:

14. NEVER USE TAB CHARACTERS FOR ANYTHING IN SOURCE CODE

OK, GNU Makefiles require the use of tab characters, but C sources do not. Agree or disagree on the number of spaces to use for indentation, use a beautifier in other cases, but don't use tabs!

Naming conventions

I'm no fan of naming conventions. Most naming conventions I've seen are far more elaborate than I would recommend. The problem with such rules is not the rules themselves, but the effort it takes each and every developer to adhere to such rules. If adherence is cumbersome, the rules are ignored. Code reviews may be a means to enforce such rules, but when a delivery is due, no project manager will accept a delay just for renaming a set of variables. Somehow that is understandable too.
Let's talk about the benefits. Naming conventions are there to help reading source code. And yes, it makes sense to be able to see at a glance if an identifier is used for a function, an integer or a pointer. But do I need to know if the loop variable for a loop from 1 to 16 is `unsigned`, `char`, `short`, `int` or `long`? Or do I need to differentiate between `uint8_t`, `uint16_t`, and `uint32_t` in such cases? Yes, sometimes I do need that information, but not often. And if we are talking state-of-the-art development, we all have a modern IDE. And most modern IDEs are offering that information for every identifier if you just hold your mouse-pointer above it for a few milliseconds; at maximum you need just one key-press.
OK, I am truly sorry for all those developing large C++ programs with thousands of files and millions of lines using just Windows ***`notepad.exe`***. Yes, no kidding, I've personally met such people, also using the old original ***`vi`***. But they have themselves to blame.
Even if you don't want to spend any money, Eclipse is free software; Visual Studio Express 2008 doesn't cost any money either. I'm not in favour of either – since I'm hooked on SlickEdit – but they cost no money and they are far better than anything from the eighties.
So I suggest keeping naming conventions to an absolute minimum, and then play by the rules at all times.

15. USE CAPITAL LETTERS FOR ALL MACROS USED IN CODE

You start with a capital letter, and then you use only capital letters, digits and underscores. Minimize the use of macros in your code anyway, most can be done away with using constant variables and inline functions anyway. But if you use them, make them stick out.
Example:

```
#define MY_MACRO_CONSTANT 12
```

16. USE A POSTFIX _T TO INDICATE A TYPE DEFINITION

Differentiating types makes a lot of sense. Because of completion functions in modern editors I prefer to use a postfix `'_t'` for such types.
Example:

```
typedef enum eLight_tag { eON, eOFF } light_state_t;
```

17. START FUNCTION NAMES WITH A CAPITAL LETTER

After the initial capital letter use small letters, and either underscores or again capital letters to separate the words.
Example:

```
Void MyVoidFuntion(int param);
```

18. START VARIABLES WITH A SMALL LETTER INDICATING THE GENERAL TYPE

I differentiate between the following general types:

- Unsigned integers indicated by `'u'`;
- Signed integers indicated by `'i'`;
- Characters indicated by `'c'`;
- Floating point definitions indicated by `'f'`;
- `enum`-definitions indicated by `'e'`;
- Structure definitions indicated by `'s'`;
- Pointers by `'p'`.

If you really want to see what a pointer is pointing at, use `'pu'`, `'pi'`, `'pf'`, `'pe'`, and `'ps'` respectively. And then use `'pv'` for pointers to void, and `'pfn'` for pointers to functions. But in my opinion that already is too much.
Examples:

```
unsigned short uSpeed = 5;
uint64_t uDistance = 32000000UL;
enum eLight_tag { eON, eOFF } eLightState;
struct sPerson_tag * (*pfnGetPersonData)(uint8_t * pName);
```

Yes, I've used a few more conventions I usually adhere to. But would you really have trouble understanding my names, if I were somewhat inconsistent? See?

And finally...

19. USE THE BEST DEVELOPMENT ENVIRONMENT YOU CAN AFFORD

Wrap-up

Now you've seen the conventions I use and when. Most of my recommendations go into a good physical architecture, not into the names or layout of things. You are allowed to disagree.
If you will, consider my reasoning. Whether you want to use each and every rule stated in this document is only of secondary importance to me, if at all.
I'm perfectly willing to discuss the rationale behind every rule. However, I'm not prepared to discuss items of style and taste. Yes, your style is better than mine, your editor does more than mine and your naming conventions are far more usable than mine. I will not discuss that. If you only have to use Microsoft VS2008, your guidelines may differ greatly from mine, which is fine for me. And yes, whether your rationale for differing rules is perfectly valid, that really matters! And only that to me is a topic for discussion.
For that discussion, you are invited to visit my blog at http://www.bezem.de/blog/, look for the entry introducing the latest version of this document and post a comment with your remarks. Please be aware that all comments are moderated to avoid comment-spam, so be patient. I will not delete or edit any serious discussion elements.