S Software Style Guide

Contents

Introduction

This document gives an overview of the software style to be used when programming in C for an embedded system.

S.1 Quality Programming

Software engineering is like other fields of engineering. Engineering is about implementing the solutions to important problems by *creatively* applying methods from sound bodies of scientific theory and by experimentation – and doing it for the benefit of members of society (with minimal impact in terms of economic, environmental, and societal cost). The fact that engineers need to be *creative* to find solutions to problems leads to many interesting and novel ideas – which generally advances the state-of-the-art. This creativity is the very reason that engineering is carried out by humans (at the present).

Creativity leads to one major problem: there are a very large number of ways to implement a solution to a problem. We need to be guided by theory, practice and past experience in seeking out whether a particular solution meets the specifications.

Large systems, especially software systems, tend to involve a level of complexity that is beyond the capability of one person. The only way to build and maintain large, complex systems is by following well-defined procedures during the engineering development cycle. One of those procedures is *engineering design*. There are many procedures we can use for software engineering design – block diagrams, data flow graphs, UML etc.

The most important phase of the engineering development cycle is *implementation*. In an embedded system this involves both hardware and software. The ultimate goal of an embedded system is to meet the stated objectives such as functionality, input/output relationships, stability and accuracy. Nevertheless it is appropriate to separately evaluate the individual components of a system. Software quality is one key area that needs to be evaluated.

There are two categories of performance criteria with which we evaluate software. *Quantitative* criteria include static efficiency (e.g., memory requirements), dynamic efficiency (e.g., speed of execution), and accuracy of the results. *Qualitative* criteria centre around ease of *understanding*. If your software is easy to understand then it will be:

- Easy to debug (fix mistakes)
- Easy to verify (prove correctness)
- Easy to maintain (add features)

Since there is no "best way" to write software, this document simply outlines techniques, based on experience, that you should try to adopt when forming your own software style. In particular, the style of writing software presented leads to code that is: self-documenting, modular, and layered.

You can tell if you write good software if:
1) you can understand your own code one year later
2) others can make changes to your code.

(S.1) The two tests of writing good software

S.2 Naming Conventions

1. *Names should have meaning*

   If we observe a name out of the context of the program in which it exists, the meaning of the object should be obvious. The object `TxFIFO` is clearly the transmit first-in first-out circular buffer. The function `LCD_OutString` will output a string to the LCD display.

2. *Avoid ambiguities*

   Don't use variable names that are vague or have more than one meaning. For example, it is vague to use `temp`, because there are many possibilities for temporary data, in fact, it might even mean temperature. Don't use two names that look similar, but have different meanings.

3. *Give hints about the type*

   We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, `dataPtr`, `timePtr`, `putPtr` are pointers. Similarly, `voltageBuf`, `timeBuf`, `pressureBuf`, are data buffers. Other good phrases include `Flag`, `Mode`, `U`, `L`, `Index`, `Nb`, which refer to boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a number (counter) respectively. Don't use the abbreviation `No` for number, as you might create variables like `NoItems`, which can be misread. Instead, use the abbreviation `Nb` – the variable then becomes `NbItems`.

4. *Use the same name to refer to the same type of object*

   For example, everywhere we need a local variable to store an ASCII character we could use the name `letter`. Another common example is to use the names `i`, `j`, `k` for indices into arrays. The names `V1` and `R1` might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just the fact that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.

5. *Use a prefix to identify public objects*

An underline character should separate the module name from the function name. As an exception to this rule, we can use the underline to delimit words in an all upper-case name (e.g., `#define MIN_PRESSURE 10`). Functions that can be accessed outside the scope of a module should begin with a prefix specifying the module to which it belongs. It is poor style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the file name containing the object. For example, if we see a function call, `LCD_OutString("Hello world");` we know the public function belongs to the LCD module, where the policies are defined in `LCD.h` and the implementation in `LCD.c`. Notice the similarity between this syntax (e.g., `LCD_Init()`) and the corresponding syntax we would use if programming the module as a class in C++ (e.g., `LCD.Init()`). Using this convention, we can easily distinguish public and private objects. If the variable is public, because the name has an underline, then the first letter of the name after the underline should be capitalized (e.g., `Logger_Count` is a public variable belonging to the module "Logger" and defined in the header file `Logger.h`).

6. *Use upper- and lower-case to specify the scope of an object*

We will define I/O ports, internal registers and constants using upper-case letters. In other words, names with upper-case letters refer to objects with fixed addresses or values. `TRUE`, `FALSE` and `NULL` are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words will use an underline character to delimit the individual words, e.g., `MAX_VOLTAGE`, `UPPER_BOUND`, `FIFO_SIZE`. Global objects will begin with a capital letter, but include some lower-case letters. Local variables will begin with a lower-case letter, and may or may not include upper-case letters. Since all functions are global, we can start function names with either an upper-case or lower-case letter. Using this convention, we can distinguish constants, globals and locals.

7. *Use capitalization to delimit words*

Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether it is local or global. Some programmers use the *underline* as a word-delimiter, but except for constants we will reserve *underline* to separate the module name from the variable name.

type	**Examples**
constant	`CR`, `SAFE_TO_RUN`, `PORTA`, `STACK_SIZE`, `START_OF_RAM`
local variable	`maxTemperature`, `lastCharTyped`, `errorCount`
private global variable	`MaxTemperature`, `LastCharTyped`, `ErrorCount`
public global variable	`DAC_MaxTemperature`, `Key_LastCharTyped`, `Network_ErrorCount`, `File_OpenFlag`
private function	`ClearTime`, `wrapPointer`, `InChar`
public function	`Timer_ClearTime`, `FIFO_Put`, `Key_InChar`

Table S.1 – Examples of naming conventions

S.3 Code Style Structure (the .c file)

Maintaining a consistent style will help us locate and understand the different components of our software, as well as prevent us from forgetting to include a component (or worse, including it twice). The following regions should occur in this order in every code file (e.g., `file.c`).

1. *Opening comments*

   The opening comments will be duplicated in the corresponding header file (e.g., `file.h`) and are intended to be read by the client, the one who will use these functions. If major portions of the software are copied from copyrighted sources, then we must satisfy the copyright requirements of those sources. The opening comments should include:

   - a reference to the file name
   - the overall purpose of the software module,
   - the names of the programmers,
   - the creation (optional) and last update dates,
   - the hardware/software configuration required to use the module, and
   - any copyright information.

2. *Including .h files*

   Next, we will place the `#include` statements that add the necessary header files. Normally the order doesn't matter, so we will list the include files in a hierarchical fashion starting with the lowest level and ending at the highest. If the order of these statements is important, then write a comment describing both what the proper order is and why the order is important. Putting them together at the top will help us draw a call-graph, which will show us how our modules are connected. In particular, if we consider each code file to be a separate module, then the list of `#include` statements specifies which other modules can be called from this module. Of course one header file is allowed to include other header files. Be careful to include only those files that are absolutely necessary. Adding unnecessary include statements will make our system seem more complex than it actually is.

3. **`#define`** and **`const`** *statements*

   Next, we place the `#define` macros and constants. Since these definitions are located in the code file (e.g., `file.c`), they will be private. This means they are available within this file only. If the client does not need to use or change the macro or constant, then it should be made private by placing it here in the code file. Conversely, if we wish to create a public constant or macro, then we place it in the header file for this module.

4. **`struct, union, enum`** *statements*

   After the `#define` statements, we should create the necessary data structures using **`struct`**, **`union`** and **`enum`**. Again, since these definitions are located in the code file (e.g., `file.c`), they will be private.

5. *Global variables and constants*

   After the structure definitions, we should include the global variables and constants. If we specify the global as `static` then it will be private, and can only be accessed by functions in this file. If we do not specify the global as `static` then it will be public, which means it could be accessed by any other module (that other module defines it as `extern` and the linker will resolve the reference). Therefore, we should use `static` declarations as the norm.

   We put all the globals together before any function definitions to symbolize the fact that any function in this file has access to these globals. If we have a permanent variable that is only accessed by one function, then it should be defined as a `static` local. The scope of a variable includes all the software in the system that can access it. In general, we wish to minimize the scope of our data.

declaration	accessibility
`short file_PublicGlobal;`	by any function via **`extern`** declaration
`static short PrivateGlobal;`	in this file only

`void function()`
`{`
`  static short staticLocal;`
`}`
	by this function only, but persistent
	(not on stack and initialised once only)

Table S.2 – Accessibility of variables

6. *Prototypes of private functions*

After the globals, we should add any necessary prototypes. Just like global variables, we can restrict access to private functions by defining them as `static`. Prototypes for the public functions will be included in the corresponding header file. In general, we will arrange the code implementations in a bottom-up fashion. Although not necessary, we will include the parameter names with the prototypes. Descriptive parameter names will help document the usage of the function.

7. *Implementations of the functions*

The heart of the implementation file will be, of course, the implementations. Again, private functions should be defined as `static`. The functions should be sequenced in a logical manner. The most typical sequence is bottom-up, meaning we begin with the lowest level and finish with the highest level. Another appropriate sequence mirrors the manner in which the functions will be used. For example, start with the initialization functions, followed by the operations, and end with the shutdown functions. For example:

```
open();
input();
output();
close();
```

S.4 Header Style Structure (the .h file)

Once again, maintaining a consistent style facilitates understanding and helps to avoid errors of omission. Definitions made in the header file will be public, i.e., accessible by all modules. As stated earlier, it is better to make global variables private rather than placing them in the header file. Similarly, we should avoid placing actual code in a header file.

There are two types of header files. The first type of header file has no corresponding code file. In other words, there is a `file.h`, but no `file.c`. In this type of header, we can list global constants and helper macros. Examples of global constants are I/O port addresses (e.g., `MK70F12.h`) and calibration coefficients. Debugging macros could be grouped together and placed in a `debug.h` file. We will not consider software in these types of header files as belonging to a particular module.

The second type of header file does have a corresponding code file. The two files, e.g., `file.h`, and `file.c`, form a software module. In this type of header, we define the prototypes for the public functions of the module. The `file.h` contains the policies (behaviour or what it does) and the `file.c` file contains the mechanisms (functions or how it works.) The following regions should occur in order in every header file (e.g., `file.h`).

1. *Opening comments*

   The opening comments will be duplicated in the corresponding code file (e.g., `file.c`) and are intended to be read by the client, the one who will use the functions and variables. We should repeat copyright information as appropriate. The opening comments should include:

   - a reference to the file name
   - the overall purpose of the software module,
   - the names of the programmers,
   - the creation (optional) and last update dates,
   - the hardware/software configuration required to use the module, and
   - any copyright information.

2. *Including .h files*

Nested includes in the header file should be avoided. Nested includes obscure the manner in which the modules are interconnected. The only exception is if data structures or functions depend on definitions made in other modules, such as `typedefs`.

3. **`#define`** and **`const`** *statements*

Public constants and macros are next. Special care is required to determine if a definition should be made private or public. One approach to this question is to begin with everything defined as private, and then shift definitions into the public category only when deemed necessary for the client to access in order to use the module. If the parameter relates to what the module does or how to use the module, then it should probably be public. On the other hand, if it relates to how it works or how it is implemented, it should probably be private.

4. **`struct, union, enum`** *statements*

The definitions of public structures allow the client software to create data structures specific for this module.

5. *Global variables and constants*

If at all possible, public global variables should be avoided. Public constants follow the same rules as public definitions. If the client must have access to a constant to use the module, then it could be placed in the header file.

6. *Prototypes of public functions*

The prototypes for the public functions are last. Just like the implementation file, we will arrange the functions in a sequence which mirrors the manner in which the functions will be used. Comments should be directed to the client, and these comments should clarify what the function does and how the function can be used.

S.5 Formatting

Formatting is a matter of personal preference, but the following section lists techniques that can make your software easier to understand, debug and change.

1. *Make the software easy to read*

   We should develop and debug software by observing it on the computer screen. In order to eliminate horizontal scrolling, no line of code should be wider than the window or pane that it resides in.

2. *Indentation should be set at 2 spaces*

   When transporting code from one computer to another, the tab settings may be different. So, what looks good on one computer may look ugly on another. For this reason, we should avoid tabs and use just spaces. Function parameters can go on the same line as the function definition, or aligned under the function's opening parenthesis "(" when it makes sense to list the parameters on separate lines (if there are many of them).

3. *Be consistent about where we put spaces*

   Similar to English punctuation, there should be no space before a comma or a semicolon, but there should be at least one space or a carriage return after a comma or a semicolon. There should be no space before or after open or close parentheses. Assignment and comparison operations should have a single space before and after the operation. One exception to the single space rule is if there are multiple assignment statement. In this case we can line up the operators and values. For example:

```
Data     = 1;
pressure = 100;
voltage  = 5;
```

4. *Be consistent about where we put braces* { }

Misplaced braces cause both syntax and semantic errors, so it is critical to maintain a consistent style. Place the opening brace on a new line directly underneath the code that opens the scope of the compound statement. Placing the open brace at the beginning of a new line provides a visual clue that a new code block has started. Place the closing brace on a separate line to give a vertical separation showing the end of the compound statement. The horizontal placement of the close brace should line up with the opening brace, giving a visual clue that the enclosed code is a compound statement. For example

```
void main(void)
{
  int i, j, k;

  j = 1;

  if (sub0(j))
  {
    for (i = 0; i < 6; i++)
      sub1(i);
    k = sub2(i, j);
  }
  else
    k = sub3();
}
```

S.14

Use braces after all **`if-else`**, **`for`**, **`do-while`**, **`case`** and **`switch`** commands where the following statement is a *compound* statement. For the case of *single* statements, it is acceptable to leave out the braces, but we must be careful when editing and adding statements. For example, assume we start with the following code:

```
if (flag)
  n = 0;
```

Now, we add a second statement that we also want to execute if the flag is true. The following error might occur if we just add the new statement.

```
if (flag)
  n = 0;
  c = 0;
```

We get the correct software if we enclose the two statements in braces:

```
if (flag)
{
  n = 0;
  c = 0;
}
```

Leaving out braces for single statements increases our code density and is much more readable. For example:

```
if (flag)
  n = 0;
else
  n = 1;
```

is better than:

```
if (flag)
{
  n = 0;
}
else
{
  n = 1;
}
```

S.6 Code Structure

1. *Make the presentation easy to read*

We define presentation as the *look* and *feel* of our software as displayed on the screen. If at all possible, the size of our functions should be small enough so the majority of the code fits on a single computer screen. We must consider the presentation as a two-dimensional object. Consequently, we can reduce the 2-D area of our functions by encapsulating components and defining them as private functions.

Do not list multiple statements on the same line. The compiler often places debugging information on each line of code. Breakpoints in some systems can only be placed at the beginning of a line.

Consider the following two presentations. Since the compiler generates exactly the same code in each case, the computer execution will be identical. Therefore, we will focus on the differences in style.

The first example has an horrific style.

```
void testFilter(short start, short stop, short step)
  { short x, y; initFilter(); UART_OutString("x(n) y(n)");
  UART_OutChar(CR); for(x=start; x<=stop; x=x+step)
  { y=filter(x); UART_OutUDec(x); UART_OutChar(SP);
  UART_OutUDec(y); UART_OutChar(CR);} }
```

The second example places each statement on a separate line.

```
void testFilter(short start, short stop, short step)
{
  short x, y;

  initFilter();
  UART_OutString("x(n) y(n)");
  UART_OutChar(CR);

  for (x = start; x <= stop; x += step)
  {
    y = filter(x);
    UART_OutUDec(x);
    UART_OutChar(SP);
    UART_OutUDec(y);
    UART_OutChar(CR);
  }
}
```

2. *Employ modular programming techniques*

Complex functions should be broken into simple components, so that the details of the lower-level operations are hidden from the overall algorithms at the higher levels.

3. *Minimize scope*

In general, we hide the implementation of our software from its usage. The scope of a variable should be consistent with how the variable is used. In a military sense, we ask the question, "Which software has the need to know?" Global variables should be used only when the lifetime of the data is permanent, or when data needs to be passed from one thread to another. Otherwise, we should use local variables. When one module calls another, we should pass data using the normal parameter-passing mechanisms. As mentioned earlier, we consider I/O ports in a manner similar to global variables. There is no syntactic mechanism to prevent a module from accessing an I/O port, since the ports are at fixed and known absolute addresses. Some processors do have a complex hardware system known as a memory protection unit (MPU) to prevent unauthorized software from accessing I/O ports, but the details are beyond the scope of this document. For embedded software, we must rely on the *does-access* rather than the *can-access* method. In other words, we must have the discipline to restrict I/O port access to the module that is designed to access it.

For similar reasons, we should consider each interrupt service routine separately, grouping it with the corresponding I/O module if possible. In particular, rather than having one long list of interrupt service routines for the entire system, each interrupt service routine should be separately defined along with the software that supports the other I/O hardware of the module. For example, the serial port interrupt service routine should be specified in the same file that handles setting up and using the serial port.

4. *Use types*

Using a `typedef` will clarify the format of a variable. It is another example of the separation of mechanism and policy. New data types and structures will begin with an upper case letter, or end in `_t`. The `typedef` allows us to hide the representation of the object and use an abstract concept instead. For example

```
typedef short Temperature;
void main(void)
{
  Temperature lowT, highT;
}
```

This allows us to change the representation of temperature without having to find all the temperature variables in our software. Not every data type requires a **`typedef`**. We will use types for those objects of fundamental importance to our software, and for those objects for which a change in implementation is anticipated. As always, the goal is to clarify. If it doesn't make it easier to understand, easier to debug, or easier to change, don't do it.

5. *Minimise function prototypes*

Public functions obviously require a prototype in the header file. In the implementation file, we will organize the software in a bottom-up or by-use hierarchical fashion. Since the highest level functions go last, prototypes for the lower-level private functions will *not* be required.

If we need to make a function prototype, then we include both the type and name of the input parameters. Specify the function as (**`void`**) if it has no parameters.

These prototypes are easy to understand:

```
void start(unsigned short period, void(*functionPt)(void));
short divide(short dividend, short divisor);
void UART_Init(void);
```

These prototypes are harder to understand:

```
start(unsigned short, (*)());
short divide(short, short);
UART_Init();
```

6. *Declare function return types explicitly*

In general, we can remove ambiguities by clarifying exactly what we want. Unless the number of parameters is large, we will place the return type, the function name, and the input parameters on a single line. The following are good examples of the first line of several functions.

```
void main(void)
void UART_OutUDec(unsigned short number)
unsigned short UART_InUHex(void)
int RxFIFO_Put(char data)
```

7. *Declare data and parameters as* **`const`** *whenever possible*

Declaring an object as **`const`** has two advantages. The compiler can produce more efficient code when dealing with parameters that don't change. The second advantage is to catch software bugs, i.e., situations where the program incorrectly attempts to modify data that it should not modify.

8. **`goto`** *statements are not allowed*

Debugging is hard enough without adding the complexity generated when using **`goto`**. When developing assembly language software, we should restrict the branching operations to the simple structures allowed in C.

9. ++ *and* −− *should appear* once *in complex statements*

These operations should only appear as commands by themselves. Also, the compiler tends to generate more efficient code when they are separated. More importantly, the issue is readability. The statement:

```
*(--pt) = buffer[n++];
```

should be written as:

```
--pt;
*(pt) = buffer[n++];
```

or as:

```
*(--pt) = buffer[n];
n++;
```

10. *Be a parenthesis zealot*

When mixing arithmetic, logical, and conditional operations, explicitly specify the order of operations. Do not rely on the order of precedence. As always, the major issue is clarity. Even if the following code were actually to perform the intended operation (which in fact it does not),

```
if ( x + 1 & 0x0F == y | 0x04)
```

the programmer assigned to modify it in the future will have a better chance if we had written:

```
if (((x + 1) & 0x0F) == (y | 0x04))
```

11. *Use* **`enum`** *instead of* `#define` *or* **`const`**.

The use of **`enum`** allows for consistency checking during compilation, and provides for easy to read software. A good optimizing compiler will create the same object code for the following four examples. So once again, we focus on style.

In the first example, we need comments to explain the operations:

```
int Mode;
void function1(void)
{
  Mode = 1; // no error
}

void function2(void)
{
  if (Mode == 0)
    // error?
    UART_OutString("error");
  Mode = 3;   // This line will compile
  Mode = 256; // This line will compile
}
```

In the second example, no comments are needed:

```
#define NOERROR 1
#define ERROR 0

int Mode;

void function1(void)
{
  Mode = NOERROR;
}

void function2(void)
{
  if (Mode == ERROR)
    UART_OutString("error");
  Mode = 3;   // This line will compile
  Mode = 256; // This line will compile
}
```

In the third example, the compiler performs a type-match, making sure `Mode`, `NOERROR`, and `ERROR` are the same type:

```
const unsigned char NOERROR = 1;
const unsigned char ERROR = 0;

unsigned char Mode;

void function1(void)
{
  Mode = NOERROR;
}

void function2(void)
{
  if (Mode == ERROR)
    UART_OutString("error");
  Mode = 3;   // This will compile
  Mode = 256; // This line will *NOT* compile (Mode>255)
}
```

Enumeration provides a check of both type and value, if the compiler supports it. Standard C compilers do **NOT** support it, but C++ compilers do. However, in C it is good programming practice to see the type required by a variable or function parameter, and provide the necessary enumerated type. We can explicitly set the values of the enumerated types if needed.

```
enum ModeState {ERROR, NOERROR};
enum ModeState Mode;

void function1(void)
{
  Mode = NOERROR;
}

void function2(void)
{
  if (Mode == ERROR)
    UART_OutString("error");
  Mode = 3; // This will *NOT* compile in C++ (out of range)
  Mode = 256; // This line will *NOT* compile (out of range)
}
```

12. *Don't use bit-shift for arithmetic operations*

Microprocessor architectures and compilers used to be so limited that it made sense to perform multiply/divide by 2 using a shift operation. For example, when multiplying a number by 4, we might be tempted to write `data << 2`. This is wrong; if the operation is multiply, we should write `data * 4`. Compiler optimization has developed to the point where the compiler can choose to implement `data * 4` as either a shift or multiply depending on the instruction set of the computer. When we use `data * 4`, we have code that is easier to understand than `data << 2`.

S.7 Comments

Comments are an important aspect of writing quality software. They often tell the reader *why* something is done, rather than *how* (which should be apparent from reading the actual code).

The beginning of every file should include the file name, purpose, author, date, and copyright.

The beginning of every function should include a comment block outlining the purpose, input parameters, output parameters, and special conditions that apply. The comments at the beginning of the function explain the policies (e.g., how to use the function.)

Comments can be added to a variable or constant definition to clarify the usage. In particular, comments can specify the units of the variable or constant. For complicated situations, we can use additional lines and include examples. For example,

```
short V1;             // voltage at node 1 in mV,
                      // range -5000 mV to +5000 mV

unsigned short Fs;    // sampling rate in Hz

int FoundFlag;        // 0 if keyword not yet found,
                      // 1 if found

enum TMode            // system states for the serial port
{
  IDLE,
  RECEIVE,
  TRANSMIT
};

enum TMode Mode;      // determines serial port action
```

Comments can be used to describe complex algorithms. These types of comments are intended to be read by our coworkers. The purpose of these comments is to assist in changing the code in the future, or applying this code to a similar but slightly different application. Comments that restate the function provide no additional information, and actually make the code harder to read. Examples of bad comments include:

```
time++;      // add one to time
mode = IDLE; // set mode to IDLE
```

Good comments explain why the operation is performed, and what it means:

```
time++;      // maintain elapsed time in msec
mode = IDLE; // switch to idle mode
             // because no more data is available
```

We can add spaces so the comment fields line up. As stated earlier, we avoid tabs because they often do not translate from one system to another. In this way, the software is on the left and the comments can be read on the right.

Alternatively, comments can appear on the line before the actual code: It is also good practice to separate blocks of code by blank lines, making "paragraphs" of code.

```
void main(void)
{
  // Initialise Tower
  Tower_Init();

  // Loop forever (embedded software never ends!)
  while (1)
  {
    // Debug pulse on entry
    if (~DEBUG)
      PTT |= 0x80;

    // Receive commands from the PC
    Tower_ReceiveFromPC();

    // Send status to the PC
    Tower_SendToPC();

    // Update motors in background
    Motors_Update();

    // Debug pulse on exit
    if (~DEBUG)
      PTT &= ~0x80;
  }
}
```

S.8 Doxygen

Doxygen is a widely used and free documentation generator:

http://www.stack.nl/~dimitri/doxygen/

The documentation is written within code, and is thus relatively easy to keep up to date. Doxygen can cross reference documentation and code, so that the reader of a document can easily refer to the actual code.

Doxygen can generate an on-line documentation browser (in HTML) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

You can also configure Doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

S.8.1 Special Comment Blocks

A special comment block is a C or C++ style comment block with some additional markings, so doxygen knows it is a piece of structured text that needs to end up in the generated documentation. There are various styles of special comment block, we will use the following style:

```
/*! @file
 *
 *  @brief I/O routines for UART communications on the TWR-K70F120M.
 *
 *  This contains the functions for operating the UART.
 *
 *  @author PMcL
 *  @date 2015-07-23
 */
```

Note the exclamation mark (!) after the usual C opening comment marker "/*". This tells doxygen that it is the start of a special comment block. We use the style of beginning each line with a space-asterisk-space (" * ").

Variables

If you want to document the variables of a file, or members of a struct, union, or enum, it is sometimes desired to place the documentation block after the declaration instead of before. For this purpose you have to put an additional < marker in the comment block. Note that this also works for the parameters of a function. For example:

```
uint8_t Packet_Command,    /*!< The packet's command */
        Packet_Parameter1, /*!< The packet's 1st parameter */
        Packet_Parameter2, /*!< The packet's 2nd parameter */
        Packet_Parameter3, /*!< The packet's 3rd parameter */
        Packet_Checksum;   /*!< The packet's checksum */
```

Functions

To document a function, we use something like:

```
/*! @brief Sets up the UART interface before first use.
 *
 *  @param baudRate The desired baud rate in bits/sec.
 *  @param moduleClk The module clock rate in Hz
 *  @return BOOL - TRUE if the UART was successfully initialized.
 */
BOOL UART_Init(const uint32_t baudRate, const uint32_t moduleClk);
```

Modules

Modules are a way to group things together on a separate page. Members of a group can be files, functions, variables, enums, typedefs, and defines, and also other groups.

To define a group, you should put the `\addtogroup` command in a special comment block. The first argument of the command is a label that should uniquely identify the group. The second argument is the name or title of the group as it should appear in the documentation.

For example, to document a `.c` file as a module we use something like:

```
/*!
 *  @addtogroup UART_module UART module documentation
 *  @{
*/

...

/*!
 * @}
*/
```

You can group members together by the open marker "`@{`" before the group and the closing marker "`@}`" after the group. The markers can be put in the documentation of the group definition or in a separate documentation block. So in another file you could use:

```
/*!
 *  @addtogroup UART_module
 *  @{
*/

...

/*!
 * @}
*/
```

to add additional members to the `UART_module` group.

S.8.2 Doxygen Commands

All doxygen commands in the comment block start with an at-sign (@). Some of the more common doxygen commands are given below:

Command	**Usage**
@addtogroup <module_name> [(title)]	A mechanism to group documentation together into a single page called a module. The module name appears after the command. The title is optional. The open marker `@{` and close marker `@}` are used to define the group.
@author	Starts a paragraph where one or more author names may be entered.
@brief	Starts a paragraph that serves as a brief description.
@date	Starts a paragraph where one or more dates may be entered.
@file <name>	Indicates that a comment block contains documentation for a source or header file, with an optional <name>. **Note**: This command must be present to document global objects (functions, typedefs, enum, macros, etc.).
@param <name>	Starts a parameter description for a function parameter with <name>, followed by a description of the parameter.
@return	Starts a return value description for a function.

For a complete list of doxygen commands, see:

http://www.stack.nl/~dimitri/doxygen/manual/commands.html

S.9 Compiler Specific Coding Style

A compiler's default settings may generate a warning on many instances of correct code. Rather than turning off the warnings, which are designed to catch our programming errors, we will adopt a coding style which circumvents the creation of these warnings.

S.9.1 Result of function-call is ignored

This warning occurs because we should be using the result of a function-call. If we do not wish to use the result of a function-call (for example, a function returns an error number that we don't wish to handle), then we can adopt one of two strategies.

The first strategy is to typecast the function result to **`void`**. For example:

```
void main(void)
{
  // Initialise Tower
  (void)Tower_Init();
  ...
```

The second strategy is to set up your code structure in a form where you can easily handle the expected result:

```
void main(void)
{
  // Initialise Tower
  if (!Tower_Init())
  {
    // Error handling goes here
  }
  ...
```

S.9.2 Condition always TRUE

This warning occurs when we implement our “loop forever” **`while`** loop with an argument which is constant:

```
void main(void)
{
  // Loop forever (embedded software never ends!)
  while (1)
  {
    // Do our stuff...
  }
}
```

The way around this is to implement a **`for`** loop with no initialization, condition and post-processing parts:

```
void main(void)
{
  // Loop forever (embedded software never ends!)
  for (;;)
  {
    // Do our stuff...
  }
}
```

S.9.3 C99

A subtle but important consideration in adopting a coding style is the version of the C language used. Modern compilers support two main options – ANSI C (commonly known as C89, and published as ISO/IEC 9899:1990) and C99 (ISO/IEC 9899:1999). C99 was recently withdrawn by the International Standards Organisation in favour of C11 (a new standard ratified in 2011), but compiler support for this standard is minimal at this time.

C99 brings valuable features to the C language, such as:

- The ability to mix declarations and code, i.e. to declare a variable at any point in a function.
- The first expression in a for loop may be a declaration, as in C++.
- The **`inline`** keyword, which hints to the compiler that a function should be included inline rather than called.
- Support for a Boolean type called **`bool`**, with values **`true`** and **`false`**, which are defined in **`<stdbool.h>`**.
- C++ style **`//`** one line comments.
- More flexible initialisation of arrays and **`struct`**s.

The utility of these features is seen as sufficient justification for the use of C99.