Theta Engineering Firmware
Coding Conventions

Best Practices

What constitutes "best practice" in software development is an ongoing topic of debate in industry and academia. Nevertheless, certain principles have emerged over the years as being sound and beneficial. In taking a conservative stance on this topic, we will avoid the most recent and contentious ideas and stick with the ones that have withstood the test of time. The principles we will use in the development of firmware are:

- **Object oriented design** – Even though we are not programming in an "object oriented language" per se, the principles of object oriented design are still applicable. We will use a C module to correspond to an "object", meaning a body of code that deals with a specific item or conceptually small zone of functionality, and encapsulates the code and data into that module.
- **Separation of interface and implementation** – Each module will have a .c file that comprises the implementation and a .h file specifying the interface. Coding details and documentation pertaining to the implementation should be confined to the .c file, while items pertaining to the interface should be in the .h file.
- **Encapsulation –** Each module should *encapsulate* all code and data pertaining to its zone of responsibility. Each module will be self contained and complete. Access to internal variables if necessary will be provided through published methods as described in the header file for the module. A module may use other appropriate modules in order to do its job, but may do so only through the published interface of those modules. It should not rely upon any undocumented features or side effects. The use of globals is discouraged. All functions in a module not intended to be part of its public interface must be declared *static.* All module-level data items should be declared *static.*
- **One-to-one correspondence –** An implication of object oriented design and its principle of encapsulation is that there is a one-to-one correspondence between objects and "items of work". In other words, one does not design the system to have two different objects that might be doing the same job, nor does one have the same object doing two different jobs. Carrying this concept further, we should strive to maintain one-to-one correspondence at all levels, not just at the "object" (i.e., module) level. If one finds the same snippet of code occurring more than once in a module, he should consider consolidating those occurrences into a single function. If one finds he has a single data item that is representing too many things, it may be advisable to create a separate data item for each thing. A numerical constant that is used several times in the code should be defined as a symbolic constant at one place in the code.
- **Naming Conventions –** It is generally agreed that naming conventions are a good idea. What naming convention is used is not terribly important. What is important is that *some* naming convention is used and agreed upon. The convention we will use is a mild form of *Hungarian Notation* with a few variations specific to the field of embedded programming and bit twiddling:

  Functions and variable names are written with the upper-lower case convention, where multiple words are strung together as necessary to create a descriptive name, and each word starts with an upper case letter.

  All-upper case is used for defined constants and symbols, such as are created with #define or typedef statements, with underbar being used when needed to separate words for readability. All-upper case with a lower-case "e" prefix is used for the values of an enum.

  All-upper case symbols used to define a bit mask should have a lower-case "b" prefix. For example:

```
#define SOME_BIT     (7)
#define bSOME_BIT    (0x80)
```

  The first example gives the ordinal bit position whereas the second provides the same information in the form of a bit mask.

An underbar preceding a bit name indicates it is active low, for example:

```
#define b_SOME_BIT   (0x80)
```

A simple prefixing system is used for data items. It is a single lower case letter that represents the data type:

PREFIX	TYPE	SIZE
b	Bit	1 bit
c	Char or Byte	8 bit
n	Integer or Short	16 bit
l	Long	32 bit
e	Enum	8, 16, or 32 bit
u	User Defined	
p	Pointer	16 bit
a	Array	

If an item is a pointer or an array name, the "p" or "a" prefix, respectively, should be used rather than the type being pointed to or the type of the array elements.

Another little twist we use is:

Functions that are used to indicate a true/false condition (i.e., functions that return a boolean) have the word "Is" in their name. This makes them very readable in if statements, for example: "if (MotorIsRunning())" or "if (IsValid())".

Brace-matching and indenting style is at the discretion of the code writer. However, whatever style is used should be consistent at least to the function level. In other words:

Don't mix indenting styles within the same function.

To accommodate file sharing amongst multiple developers, the following rule is adopted for tab settings:

All indenting is done with spaces, not tab characters. The IAR development environment used at Theta Engineering has a setting for converting tabs to spaces, as do most programmer's editors.

Documentation

The following authors do a good job of stating the rationale and philosophy of good documentation:

"Coding standards are a vital part of writing software in any language if one intends the code that has been written to survive being handed over to other personnel than the original authors. Once a standard is known and used, one can quickly understand such code (if the standard is itself reasonably good).

"It is all too common for the new programmer, having received any significant amount of code written by others, to soon thereafter call for a necessary complete rewrite of the code. In some cases this may be justified by other development requirements, but far more often this is really due to the fact that the programmer feels he could rewrite the code in less time than it would take to figure out how the inherited code works! And so the programmer can get "productive" sooner.

"The problem with this (in addition to the earlier problem of poorly written code) is that rarely does this new programmer then rewrite the code so as to survive being taken over *from him*! And the cycle repeats. This is really a major source of software development cost over the life of some products, and is entirely unjustifiable in the light of proper coding practices.

"Coding practices include standards for commenting code, naming objects, and general layout of source code. It can go beyond any single module of code and include standards for the elements of an entire development project or family of projects.

"The inexperienced, lazy, or self-interested developer, hearing that coding standards are in place and will be enforced, may protest that performing all that "extra work" just "slows me down". But a quarter-century of experience in the area, both with and without coding standards, over hundreds of projects and many millions of lines of code, have demonstrated the practicality of even moderately loose standards in enhancing the survivability of code across projects and personnel changes.

"I've had developers, on turning in non-compliant code, such as with no comments, lament that they intended to "do it later when I was all done". But that "later" never arrives. The developer gets other assignments, and, more importantly, even if time does exist for the task, it will take much longer at the end of a project, where the developer who wrote it now has to do a certain amount of reverse engineering to figure out what was done.

"The rule is:

"THE RIGHT THING TO DO IS APPLY ALL NECESSARY CODING PRACTICES AND STANDARDS AS THE WORK IS BEING DONE, INCLUDING ALL COMMENTING AND NAMING CONVENTIONS. NEVER WAIT UNTIL 'LATER'.

Comments — Purpose For And Use Of

"One very valuable point is to consider when and why to comment the code. Let me provide you a stable datum for comments:

"COMMENTS SHOULD NOT SAY WHAT THE CODE IS DOING. COMMENTS SHOULD EXPLAIN HOW THE APPLICATION FUNCTIONS ARE BEING DONE AND WHY.

"And this:

"COMMENTS ARE RATIONALE.

"In other words, don't do this:

```
"A = A + 1  ' Add one to A.
```

"It is little more than a distraction. Somewhere, however, what "A" is counting and how that gets some part of the application's functions done would be documented. It is common to occasionally write extended comments — even a page or two, to describe a complex procedure. But remember to always explain *why* this is being done and how it gets the job done.

"One purpose of a comment is to communicate to *others*. All too often the programmer writes just enough to remind himself (he thinks) so that when he visits the code later he will jog his memory about it. But this can fail even himself, and almost never communicates to others. So whenever you write comments, you must assume the person reading them (a) needs to understand it; (b) knows little more than the immediate code and its surrounding comments.

"Now, in the heat of coding one often gets an insight into how some algorithm or program activity should be done. *That is the time to write comments*. Not code. Write them all down as comments, including what factors makes it necessary, why this solution is workable or improves something, how it fits with other parts of the system, etc. All the thoughts and considerations that make up the idea. Yes, this is "designing on the fly", and it even creates the possibility that you will have to update the documentation if you change the code. But such comments are of great value, both to yourself and others.

"It is actually better to write this all as a comment block rather than breaking out the word processor and making a formal design. Formal designs are good for larger activities, or where there must be review of a design, but where you've figured out how several parts come together and work to get some functionality accomplished, use a comment block. The reason is that too much reference to other documentation will result in effectively no comments. Developers can't find the document or won't take the time to look it up. But in-line comments are available at the right time and place, always. So where there would be several paragraphs or a page or two, comments are quite suitable.

"So another maxim:

"WRITE IT WHEN YOU THINK IT UP."

(from *Coding Practices for Visual Basic* by Richard Berman).

"Note that comment lines require as much review as code lines. Misspellings, lousy grammar, and poor communication of ideas are as deadly in comments as outright bugs in code. Firmware must do two things to be acceptable: it must work, and it must communicate its meaning to a future version of yourself - and to others. The comments are a critical part of this and deserve as much attention as the code itself."

(from *A Guide to Code Inspections* by Jack Ganssle. http://www.ganssle.com/Inspections.pdf).

"My standard for commenting is that someone versed in the functionality of the product, but not the software, should be able to follow the program flow by reading the comments without reference to the code itself. Code implements an algorithm; the comments communicate the code's operation to yourself and others, maybe even to a future version of yourself performing maintenance years from now.

"Write every bit of the documentation (in the U.S. at least) in English. Noun, verb. Use active voice. Be concise; don't write the great American novel. Be explicit and complete; assume your reader has not the slightest insight into the solution to the problem. In most cases, I prefer to incorporate an algorithm description in a function's header, even for well-known approaches like Newton's Method. A description that uses your variable names makes a lot more sense than "see any calculus book for a description." And let's face it: once a program is carefully thought out in the comments, it's almost trivial to implement.

"Begin every module and function with a header in a standard format. The format may vary a lot between organizations, but should be consistent within a team. Every module (source file) must start off with a general description of what's in the file, the company name, a copyright message if appropriate, and dates. Start every function with a header that describes what the routine does and how, goes-intas and goes-outas (i.e., parameters), the author's name, date, version, a record of changes with dates, and the name of the programmer who made the change.

"Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines that work together to perform a larger function.

"Explain the meaning and function of every variable declaration. Long variable names are merely an aid to understanding; accompany the descriptive name with a meaningful, prose description.

"One of the perils of good comments-which is frequently used as an excuse for sloppy work-is that over time the comments no longer reflect the truth of the code. Comment drift is intolerable. Pride in Workmanship means we change the docs as we change the code. The two things happen in parallel. Never defer fixing comments until later, as it just won't happen. Better to edit the descriptions first, and then fix the code.

"If you use code inspections (and please do; they are the cheapest known way to get rid of bugs) review the comments as well as the code. Both are equally important.

"Finally, consider changing the way you write a function. I have learned to write all of the comments first, including the header and those buried in the code. Then it's simple, even trivial,

to fill in the C or C++. Any idiot can write software following a decent design; inventing the design, reflected in well-written comments, is the really creative part of our jobs."

(from *Comments on Comments* by Jack Ganssle, Embedded Systems Programming, March 2002).

To sum up, the objective we are striving for is not merely "code that works"; it is:

"Code that communicates how it works."

Commenting Conventions

All source files must include the customer's copyright banner at the top.

Next is a revision history which summarizes, dates and identifies the author of all changes made after the initial release of the module.

Each .h file should start with a comment block that explains the purpose and overall usage of the module, from the perspective of someone who wants to use the module but doesn't necessarily know (or care) anything about its internal workings. The revision history entries in the .h file pertain to changes which impact the module's interface or functionality visible from the interface.

Each .c file should have a comment block following the revision history that gives an overview of the module's implementation. Many times, simply referring the reader to the .h file will provide an adequate overview, but in some cases there may be a tricky bit in the implementation that needs to be explained or verbiage is needed to tie it all together when it is not obvious from the descriptions of the individual functions. This revision history for the module is like in the .h file, only this would include *all* changes to the module.

Each function declaration in the .h file should be followed by a comment block that explains the function's purpose and usage, again from the viewpoint of someone who doesn't know or care about the function's internal workings. Headings typically used in the comment block are:

```
PURPOSE:
GIVEN:
ASSUMES:
RETURNS:
SIDE EFFECTS:
USAGE:
ADDITIONAL INFO:
```

It is expected every function would have a PURPOSE. Any function that took one or more arguments would have a GIVEN section to explain what the arguments are and what they do. Any restriction on the range of values for an argument would be noted here. If the function depends on any pre-established conditions in order to function properly, these conditions should be listed in ASSUMES. If the function is not *void*, its return value would be explained in the RETURNS section. Any side effects created by the function would be listed under SIDE EFFECTS. An init function for a module that interfaced to external hardware would document here any configuration of the I/O pins it uses, for example. Any configuration of an on-chip peripheral would be documented here. It is expected a function would have either a RETURNS section or a SIDE EFFECTS section, otherwise it wouldn't have an effect on anything! Any issues on using the function would be explained under USAGE. A heading such as ADDITIONAL INFO can be used as a catch-all for anything that should be explained but doesn't fit elsewhere.

Each function in the .c file not already covered in the .h file should have a comment block at the beginning of the body of the function definition using the same format just described. Sometimes it may be necessary to have a comment block for a function even though it is already covered in the .h file because of some tricky bit about its *implementation* or because further explanation is needed to see how it actually works.

Of course, within the body of each function there should be comments as needed to explain the code, as covered above in **"Documentation"**.

Source file text should be kept within 80 columns if possible to avoid line wrap problems when making hardcopy printouts of the source files.