10 Tricks for Documenting Embedded Software

Jacob Beningo - October 08, 2013

There are few things more challenging in software development than acquiring a code base with little to no documentation and being required to maintain it. Documentation doesn't just tell an engineer what a particular function or variable does but it also demonstrates and conveys why the software was implemented in a particular fashion. There are millions of decisions that are made when implementing software and it can be critical for the maintaining engineer or even the future you to retain as much of that decision making process as possible.

Part of the problem with documenting code boils down to delivery pressures, improper design and the fact that documenting how something works just isn't as fun or exciting as developing it! Many engineers (including myself) hate having to document code but it is such an essential part of what embedded engineers do that we can't skip it or half-heartedly create it. However, there are some tricks that can be kept in mind during the software development stage that will ensure future developers maintain their hairline.

**Trick 1 - Document as you go not afterwards**
The pressure to ship a product often leads to wild-west style coding where code is flung here and there in an effort to just get something working so that it can be pushed out the door. During the frantic coding little thought is ever given to writing down what the code is actually doing. Once the product ships the design team goes back through the code in a "documentation" phase. The problem with this is that it can be weeks or even months after the code was written! It can be a challenge for some engineers to remember what they had for breakfast yesterday let alone what a piece of code from two weeks ago does. The result is inaccurate documentation that later leads to misunderstandings and bugs.

The trick of course is to document as you go while the decisions are being made. A formalized process with external documentation would definitely slow a developer down but adding comments into the code base really doesn't take any more time. The first thing the developer can do is write a few comment lines on what the code is going to do and then write away. If a change in implementation is made the developer can update the comment immediately. In any case, developing the comments as you code can only save time and increase clarity that results in fewer mistakes and quicker delivery.

**Trick 2 - Automate the document generation**
Despite documenting the code in great detail there is always a demand that external documentation be generated that anyone can see without looking at the code. This usually results in double documentation efforts. Thankfully, there are tools available that can read in code comments and then generate interface and other documentation details of the code! Saving the engineer from having to do the same job twice! One free example of a tool like this is Doxygen. While a developer is
writing their code, they format their comments in a specific way with details that they would want in external documentation. Then, they can run doxygen and generate either html, rtf or pdf documents that exactly reflect the comments within the software. The nice thing about this is that if you keep your comments up to date then the external documentation will be too! Tutorials on how to get this tool up and running can be found at www.beningo.com under Design Articles -> Design Cycle. **Trick 3 - Write non-obvious comments**

It is fantastic when a developer documents their code but extremely annoying when the documentation is nothing more than a repeat of a variable or function name. Comments should be descriptive and provide additional details beyond the obvious! Provide as much information as you can and don’t forget to mention relevant and related variables or functions. The developer should be able to gain insights into how the software behaves just from reading the comments. An example can be seen in Figure 1 where a simple mapping array is documented.

```c
/**
 * The Map_StateToRaw array is used to map a position lever state to the raw sensor state. They don't map 1:1, but instead there are additional raw states in order to provide finer tuned control. This requires a conversion from the user entered desired state and the internal target state.
 *
 * Example:
 * UserInput = STATE_2;
 * Target = Map_StateToRaw[UserInput];
 */

const uint8_t Map_StateToRaw[] =
{
    RAW_STATE_1,
    RAW_STATE_2,
    RAW_STATE_3,
    RAW_STATE_4,
    RAW_STATE_5,
};
```

**Figure 1 - Mapping Array**

**Trick 4 - Provide example uses to increase clarity**

It can be very helpful when functions or variable comments include an example of how they should be used. It is one thing to say how it should be used but much clearer to SHOW how it is used. This can give a much clearer picture in addition to decreasing the chances that the object is used the wrong way. Figure 2 shows an example of how a function can be documented to show a developer how it should be used in order to remove the guess work and give a clear picture.
Trick 5 - Create a documentation standard
Just like with writing code, there should be a standard that is associated with the development of comments and documentation for the code. Since there are not likely to be nearly as many items in a documentation standard, it is highly recommended that it be rolled up into the coding standard. This is meant to make sure that everyone on a team comments the same way and documents the same way in order to ensure ease of development. A developer should be focused on the problem at hand not trying to wade through the comments.

Trick 6 - Use a documentation template
The easiest way to ensure that comments follow a standard is to create a template for header, source and supporting documents. When a new module is created, it is created from the template and then relevant information is added. This will help ensure that file info blocks, code segments, functions and variables are all commented in the same format. The best part about this approach is that it also saves a ton of time and can help reduce copy and paste errors of copying one module to be the pretend template of another. In order to make life easier, I’ve developed templates that can be downloaded here.

Trick 7 - Pull/Push from Diagrams
Trick 7 - Pull/Push from Diagrams
Before the software implementation phase of a project begins, there should have been a software design phase. This design phase undoubtedly produced many pretty diagrams such as flowcharts and state machines that are used during the actual implementation. While these documents act as the road map for the software, during development and testing there are always deviations! Unfortunately these changes rarely find their way back into the diagrams. The result is design documents and software that don't match! During the implementation and testing phase keep these diagrams on hand so that if deviations from them arise, the diagrams can be updated on the spot. Leaving them to be updated later is never the right answer. After all, we always have the best intention to go back and update or fix something but there just never is time to do it.

Trick 8 - Consistent use of commenting brackets
As strange as it sounds, many holy wars have been fought over when, where and what type of comment bracketing should be used! In all seriousness though, no matter what your religion, it all boils down to consistency. If a team has decided that only the /* ... */ style of comments will be used then only use that style. If the // style has been decided then use that. Personally, the author prefers to use /* ... */ for functions and module level declarations whereas // is used for function code documentation. Whatever the preference, make sure that it is done the same way each time. It will help make life just a little bit easier.

Trick 9 - Keep it readable (i.e. formatted nicely)
Keeping code structured and easy to read is very important in order to ensure that misunderstandings and hence bugs do not find their way into the code. Comments are no different. Sporadically structured comments make it difficult for the eye to catch the comment and more importantly to catch something that is out of place. Comments should be formatted such that if the code is printed out (pretty rare these days but I still occasionally print my code) the comments do not run across multiple pages. In large block of comments such as a file header or function comment if you are using block indicators, do not include any trailing character such as # or *. This will only make updating the documentation more difficult.

Trick 10 - Embed Images and Diagrams
With the use of automated tools, it is possible to have coding standards, abbreviations, project details, requirements and a plethora of other items included in the built documentation. The very design diagrams such as flowcharts can even be included! Utilizing this type of capability allows for a code base to not only contain the executed code and logic but also contain anything and everything you might want to know about a project all in one place!

Jacob Beningo is a lecturer and consultant on embedded system design. He works with companies to develop quality and robust products and overcome their embedded design challenges. Feel free to contact him at jacob@beningo.com, at his website www.beningo.com or on twitter @Jacob_Beningo.