10 Tips Make Embedded-System Code Easy to Maintain

Timothy Stapko, Digi International

Ecnmag.com - January 01, 2008

In the rush to get a product out the door, programmers often ignore code maintenance — a key aspect of application development. For applications with short lives, this rush may not pose a significant problem because once deployed, no one will touch the code again. Embedded systems applications, however, may have lives that span decades, and early coding mistakes can result in significant bug-fix and update costs later on.

You must consider code maintenance during design and implementation of software for an embedded application that will have a long life. The following ten tips do not constitute a complete list, but they address common issues that can give the team that maintains your application cause to curse your name. And you may be part of that team:

1. Avoid assembly code.

On a low-end PIC microcontroller (MCU), you have no choice but to use assembly language, and on a high-end ARM processor you probably do not need it. Between those processor extremes, many programmers may use assembly language to increase performance and to reduce code size. But using assembly-language code can derail your project and set it back months.

Assembly language lets you directly access a machine's functions, but the difficulty of understanding just what happens in assembly language code can overshadow the performance gains you hope to achieve. For this reason, people developed higher-level languages such as C and Java. Always treat assembly language code with suspicion, because it can easily violate the "safety" features built into higher-level languages.

If you must use assembly language, include verbose comments that will save time and reduce frustration when someone examines your code. Use comment blocks of assembly-language code that include no more than five or six instructions. Ideally, use pseudo-code in comments to describe the operation of an algorithm. (see "Keep all documentation with the code," in the online version of this article at www.ecnmag.com).

2. Avoid comment creep.

This general programming tip becomes especially important in long-lifetime applications: Keep comments with the code they document. As others update a program, comments can "migrate" within a block of code, which makes a listing difficult to understand. The following example shows the result of comment creep:

```
// This function adds two numbers and returns the result
#if __DEBUG
void printNumber(int num) {
 printf("Output: %d ", num);
}
#endif
// This function multiplies two numbers and returns the result
int multiply(int a, int b) {
 return a*b;
}


int add(int a, int b) {
#if __DEBUG
 // Debugging output
    printNumber(a+b);
#endif
 return a+b;
}
```

Note that the comment for the function add appears at the top of the source code. By leaving a space between a comment and a function, other developers might squeeze in their code. In this case, they added the printNumber function between the add function and its associated comment. Later, someone saw the addition function and thought it logical to put a multiply function nearby. In the process, he or she separated the add function and its comment. To fight this problem, keep code inside the function it documents or make a comment block obvious by putting lines above and below it.

3. Do not optimize prematurely.

Time constraints, sloppy coding or overzealous engineers often force premature optimization — a cardinal sin for any programmer. Any program you write should start as simply as possible yet still provide the desired functionality. If you have optimum performance as a system requirement, continue to aim for a simple program even if it does not meet final performance goals. Once the complete unit has been tested and debugged — a program or a component of a larger system — go back and optimize. Haphazardly optimizing code can lead to a maintenance nightmare because optimized code is usually hard to understand, and you still might not get the required performance results. Ideally, use a profiler such as Intel's VTune or gprof, which works with GCC, to locate bottlenecks you can attack.

4. Keep ISRs simple.

Simple interrupt service routines (ISRs) improve performance and ease maintenance. Because ISRs operate asynchronously, they are inherently difficult to debug; so keep their tasks to a minimum. Try to move any data processing or housekeeping tasks out of an ISR and into the main program. Then, the ISR will only grab data, say, from hardware, place it in a buffer, raise a data-ready flag and re-enable the interrupt.

5. Leave debugging code in source files.

During development, you will likely add code designed for debugging — verbose output, assertions, LED blinking and so on. At the end of a project, it may be tempting to remove debugging code to "clean up" the source code. However, removing the debugging code can create problems later. Anyone attempting to maintain the application code may have to recreate many of the debugging steps you included in the original code. So, keep the debugging code in the source file to simplify maintenance. If you must remove the debugging code in production builds, use conditional compilation or put the debugging code in a central module or library. Do not link it into production builds. Development of an application should include time to document and clean up debugging code. This step will be well worth the effort.

6. Separate low-level I/O routines from the higher-level program logic.

Hardware-specific operations can make it difficult to port to a new platform. So, to simplify programming and application maintenance, create your own "wrapper" interfaces for hardware APIs or direct hardware control. A wrapper can be as simple as a macro that offers a standard way for your application to control hardware. When porting to a new application within a wrapper, simply change code that defines the hardware-specific operations or API function calls. Then, the application code remains consistent, and all code changes take place in one central location.

7. Break up functionality as needed.

Unlike PC software, embedded-application code works with specialized and unique hardware. So, you may choose to divide functional units of code into small pieces, but do not split code more than necessary. Aim to keep the number of function calls in a single scope (function) to fewer than five or six, and make functional units in the software correspond to specific hardware functions. Do not split up the functionality further. If a program is brokin into too many small pieces, the call graph will look like a spider web and the code will be difficult to debug and comprehend.

8. Keep all documentation with the code.

When you document an application, try to put as much of the design and application "model" directly into the source code. If you must keep documentation separate, put it in a source file as a giant comment and link it into the program. At the least, if you use a version control system such as CVS or Microsoft's Source Safe, check the documentation into the same directory as the source code. It is too easy to lose documentation if you do not locate it with the source code.

Ideally, put all the documentation and source code for a complete application on a CD, USB memory stick or other portable storage device. Seal it in a package with the hardware and development tools you have used, and put that package in a safe place that your colleagues know about. Your successors will thank you for it.

9. Avoid clever techniques.

Clever programmers can come up with extremely compact and elegant ways to use tools such as templates, inheritance, the goto statement and the ternary operator (the "?"). But clever coding can lead to trouble. Usually only the original programmer understands the clever solution, and later he or she will likely forget how it works. So, avoid cleverness and keep the use of esoteric language features to a minimum. Do not rely on short-circuit evaluation in C statements, and do not use the ternary operator for program control. Use an if-then statement instead.

10. Put all definitions in one place.

If your code includes many constant definitions or conditional defines, keep them in a central location such as a single file or a source code directory. If you bury a definition deep within your code, it will come back to haunt you.

For further information. . .
For information about gprof, the GNU Profiler, visit www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html and www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

For information about Intel's VTune, visit www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

Timothy Stapko is lead software engineer for Rabbit, a brand of Digi International. He has more than eight years of experience in the software industry, and he is the author of "Practical Embedded Security," published by Newnes. For more information, contact Rabbit Semiconductor, 2900 Spafford St., Davis, CA 95618-6809; 530-757-8400; www.rabbit.com.