RETURN

by STEVE DREVIK

How To Comment Code

Good code is maintainable and reusable both by the original programmer as well as other programmers—that means it's well documented. This article presents guidelines for effective documentation.

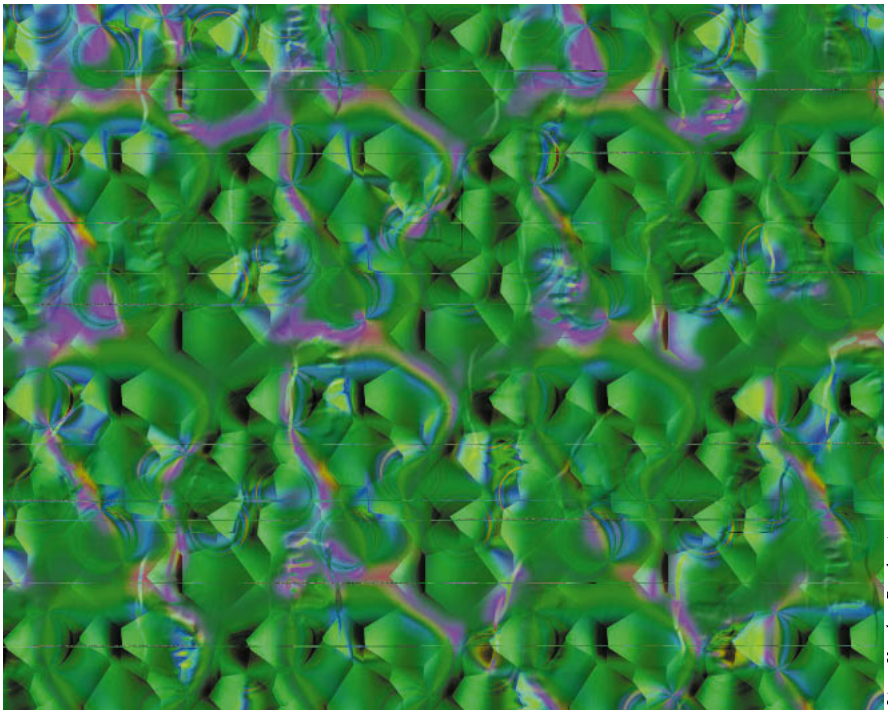
The title of this article should be enough to arouse the interest or ruffle the feathers of any programmer. Most programmers' responses to articles on how to comment code are:

- I already know how to comment code.
- The author of such an article does not know the policies of my organization and therefore is not qualified to teach me anything.

The first response is easy to both accept and dismiss. Everyone does know how to comment code. In C, you put a `/*` in front of your comment and a `/*` at the end. In BASIC, you start the line with the keyword `REM`. But it is fallacious for programmers to say they know everything about the proper way to comment code in every situation.

The second response, commenting methods change based on "policy," is a statement of what is true but not what is correct. These policies are based on certain factors:

- What is the degree of time/schedule pressure on the programmer?
- Are there any other programmers on the project? If so, to what degree do they interact on the same modules of code?
- What is the personnel turnover rate among the programming staff?
- Is the organization under any internal (corporate) or external (contractual) requirements for code maintainability?
- What are the personal politics among the programmers or groups of programmers?


Lime Voodoo Productions

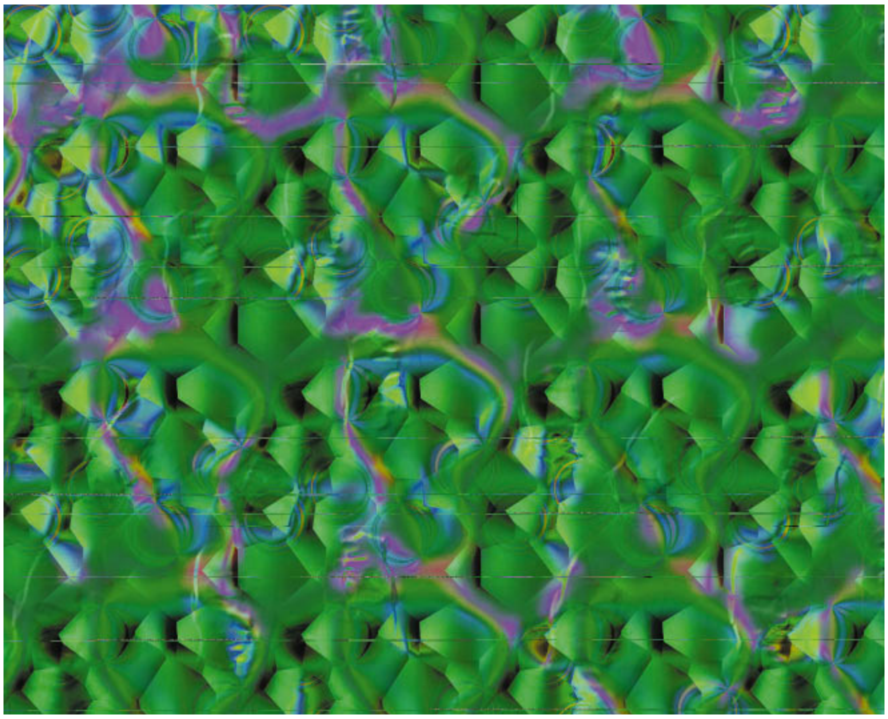
Time pressure is the most obvious force mitigating against good documentation. If the programmer is required to generate too many lines of code in a given time period, and the programmer is aware of it, there is no time to spend writing comments. Although lack of time is the most commonly used excuse, several other reasons are usually hiding under this rock.

If no other programmers are working on the task, code will typically lie uncommented, regardless of the time constraint. Why spend time explaining the obvious to an audience that does not exist? This argument is valid only if the code must be written once and never reworked or debugged after the original methodology has left the forefront of the programmer's mind—a situation that never happens. I have found, both through personal experience and through watching others, that programmers can waste a considerable amount of time trying to sort through their own recently-written code.

If several programmers are on the staff, there is usually more incentive to comment code. The level of incentive depends on how much the programmers interact with each other on the same code modules. Modern programming techniques allow dozens of programmers to work simultaneously on the same project without ever really seeing each other's code. In this case, programmers eventually fall into the single-programmer train of thought. Peer reviews can actually prevent this problem; the most common comment that comes out of a peer review is "I don't understand this. Why don't you add a comment here?"

A high rate of personnel turnover can affect the level of commenting in either direction. Some programmers will comment their code in great detail, knowing that someone else will have to take it over soon. More often, however, programmers will stop commenting, knowing it doesn't matter and wanting to get their project done—or perhaps wanting to build in a little job security in an unstable environment. Programmers document their code most religiously when the turnover is due to movement within the same company, rather than people exiting the company. When programmers move off a project but stay within the same organization, they know the next programmer is only a phone call away. If the threat of a new programmer close enough to ask questions doesn't exist, commenting work comes to a screeching halt.

No matter how well-written a module is, it cannot be reused if another programmer cannot quickly determine how to use it.



In all my career, I have seen a hundred companies ask for access to source code when contracting for complex customized software. But I have only seen two of those companies provide a written specification of how that supplied code should be commented. I can't imagine what those other 98 companies would have done if they actually brought up the source code they paid so dearly for, only to find out it was unintelligible. If my audience here includes anyone whose line of work includes writing software specifications, I would highly recommend adding a dozen or so pages describing a minimal system of code commenting. If my audience includes software development managers, I would highly recommend putting in place similar minimal guidelines about how code should be commented. For either party to fail to put minimal code commenting guidelines in this day and age is short-sighted stupidity. Enough said.

Finally, we come to the issue of personal politics. An organization can dictate documentation procedures, require peer reviews, and give the programmers enough time to follow those procedures. But, while you can lead a horse to water, but you can't make it document its code. The most common stumbling block is the Not Invented Here (NIH) syndrome loop. I call it a loop because failure to comment code properly causes other programmers to be reluctant to re-use that code, and a low level of code reuse causes programmers to slack off on their commenting. The blame lies not with the programmers but with the development managers for failing to enact and enforce strict commenting guidelines and failing to encourage re-use.

Notice that I referred to "strict" standards. To prevent NIH syndrome and assure a high level of code re-use, commenting must be of the utmost quality. No matter how well-written a module is, it cannot be reused if another programmer cannot quickly determine how to use it. A lot of good code is thrown away because of poor documentation. It is the responsibility of the programmer to follow those guidelines to the letter. If the guidelines are poorly written, the programmer should suggest improvements.

Other stumbling blocks include simple pecking-order disputes. There is no better way to humble programmers than to force them to come to you, begging you to explain the incredible ingenuity of your top-rate code. Such positioning can also make you appear more like a guru than a simple programmer to your boss. I think the recent team management paradigms have done a lot to defuse this problem.

There are plenty of reasons why people do not comment code properly. The software development managers are now well-armed and should be able to determine those reasons, defuse them, and convince the programmers of the benefits of well-commented code. Now what? Is there a right and a wrong way to comment code? Absolutely. Let's look at some of the common mistakes.

COMMON MISTAKES

In the following example, the start-up code of an embedded control product is initializing several serial ports:

```
/* Loop through MAX_SIO_PORTS-1 */
for(port = 0; port < MAX_SIO_PORTS-1;
   port++)
        {
        /* Initialize port 'x' */
        initialize_port(port);
        }
```

Notice the top comment, which tells what the loop does. The only problem is that, as a programmer, I already know C. I don't need another programmer to teach it to me again, nor do I need the same lesson repeated every time I see a `for()` statement. What I would rather know is why the programmer is looping through `MAX_SIO_PORTS-1`, instead of `MAX_SIO_PORTS`. As it turns out, the last serial port is reserved for remote debugging and is not initialized in this section of code. Instead, it is initialized in another section of code. And what exactly does `initialize_port()` do? A better way to comment this case would be:

```
/* Initialize all ports except the last
   port, which is reserved for remote
   debugging.
      The last port is initialized in the
   function check_for_debugger() */
for(port = 0; port < MAX_SIO_PORTS-1;
   port++)
      {
      /* Initialize port 'x', reset the
   UART, set baud rate, data/stop bits, and
   state of RTS control line */
      initialize_port(port);
      }
```

This is probably the most common and the most serious programming sin. Your comments should explain why you are doing something, not what you are doing. Comments should not teach language structure.

The second sin is improper documentation of a module's interaction with the rest of the world. What does the module do? How should it be called? What are its arguments? Does it access any global variables? Simply prototyping is not enough, as we see here:

```
void init_port
(
         int num,
         int speed,
         int db,
         int par,
         int status
);
```

Isn't it perfectly clear what the module does? Granted, this case is an extreme one. A more realistic case might be:

```
void init_port      /* initialize port X */
(
      int num,
      int speed,    /* Rate */
      int db,       /* Data bits */
      int par,      /* Parity */
      int status    /* RTS */
);
```

This code looks like a serial port initialization routine. If we want to initialize the first serial port to 9600 baud, 8 data bits, one stop bit, RTS off, we apparently call it this way:

```
init_port (1, 9600, 8, 0, 0);
```

Or do we send `0` for the port number? As it turns out, we are wrong on several counts. The first serial port is index `0`, the logic on the RTS is reversed because of the UART logic, and the baud rate is supposed to be an index into another array:

```
int baud_rates[300, 1200, 2400, 4800, 9600,
   19200, 38400];
```

The proper way to call it is:

```
init_port(0, 5, 8, 0, 1);
```

Or, using `#typedefs` to make it more legible:

```
init_port(PORT_0, BAUD_9600, 8, NONE,
   RTS_OFF);
```

(In fact, use of `typedefs` in this way can be considered part of proper documentation procedures.)

We had no way of knowing all this just by looking at a poorly-commented prototype. How about the following?

```
void init_port      /* Initialize a serial
   port, reset the UART, set baud rate and
   RTS output */
(
  int port_num,     /* Port index 0...
   MAX_SIO_PORTS-1. See 'sio.h' for
   typedefs */
  int baud_rate,    /* Index into global
   'baud_rates' array. See 'sio.h' for
   typedefs */
  int data_bits,    /* Number of data bits
   (7 or 8 allowed) */
  int parity,       /* Parity bits (see
   ´sio.h´ for typedefs) */
  int rts_status    /* RTS status. See
   ´sio.h´ for typedefs */
);
```

Notice the careful choice of names for the function arguments. Not all comments are found between slashes and asterisks! Choosing good, reasonably long variable names is part of the commenting process. In more archaic languages (say, Fortran), when variable names had a limited number of characters, it was standard practice to document each variable in the function header. A language that allows long variable names makes this documentation unnecessary. However, if you do not like to use long variable names or you are working in assembly language, you should continue this practice.

COMMENTING CHANGES / BUG FIXES

Need I say that no code ever works right the first time? Sometimes I feel as though I spend more time revising than writing. Usually the revisions consist of fixing typos, initializing variables that weren't initialized, and so on. On occasion, I find myself making somewhat unusual modifications to make things work correctly. Sometimes, I have to do some odd things to work around a compiler bug or to satisfy some hardware timing requirement. When I finish editing that file, I am always left with the ugly feeling that I, or someone else, will come back to that code later and wonder why the module was coded that way, or worse yet, change it back to something like the original code.

As an example, I had a version of compiler that would not do `+=` and `-=` operations correctly on huge pointers. If the fix had not been commented as follows:

```
huge char * add_huge_char_ptr
(
  huge char * a,
  huge char * b
)
{
 /*  a += b;  <-- compiler bug: += and -=
    operations don´t work right */
         a = a + b;
         return(a);
}
```

Another programmer (or even the original programmer, if forgetful) might have been tempted to change this version back for aesthetic reasons, causing the system program to go badly awry.

In some cases, I have found myself writing half-page or full-page comments about what I have done. In these situations, more is better. Explain what you did, why you did it that way, what other approaches you tried, and the repercussions of doing it another way. If you think something might be done more elegantly another way, but you don't have the time to code it now, say so. Otherwise, another programmer may be afraid to touch this section of code when the time is right.

If you don't have a source code control system (SCCS) that can document and date changes, I recommend that all bug fixes other than typos should also be accompanied with a short comment listing the programmer and the date modified. If the programmer's initials are used, this shouldn't take any more than 15 keystrokes. Adding the version number is also a great help in backtracking problems later, as well as answering the boss's query "which version has a fix for that?"

DOCUMENTING STANDARD LIBRARIES

Programmers for desktop and workstation targets benefit greatly from the availability of large standard libraries for C++ object classes and graphic windowing tools. Embedded system programmers typically don't enjoy having such libraries available because of the very nature of the target or project. The C++ object class code may take too much program memory, and windowing tools are useless unless you are working with a VGA resolution display. Speed is often another consideration—standard library routines can be slower than need-specific routines the programmer could write. Most embedded programmers find themselves creating their own standard portable library functions for things like linked list/queue management, sorting operations, and string operations (parsers, or routines to strip spaces from a string, for example).

As mentioned previously, these routines are useless to another programmer unless the documentation is written such that another programmer can figure out how to use it in a short period of time. I suggest a target time of 15-30 seconds, the average patience level of a typical programmer for another person's code. After reading for 30 seconds, most programmers lose interest.

I strongly suggest documenting your standard library routines the same way major software companies document their standard libraries: write a document using an entire page dedicated to one function. Use the same format the major companies do—the format itself isn't copyrighted! Show the function name, the arguments, discuss possible return values, and so on. Put all the information in a three-ring binder you can add to, and give everyone on the project a copy for reference. Have everyone put their copy next to their standard library reference. Not only does the documentation need to be good, it needs to be easily available. I have seen similar functions appear in a project three or four times, because the function was either never put into the standard library, or the fact it was there was forgotten.

A good programmer is productive in the long term, writing code that is maintainable and reusable for the original programmers as well as others.

HOW TO DOCUMENT A PROJECT

Documents in source files serve programmers already familiar with the project and its development environment. However, if the project is handed over to another programmer or to another company (or if the customer has to take over code development or fixes), a batch of commented source files may prove close to meaningless. Equally important is the development environment that weaves all the pieces together—the compiler/linker tools, the source code control system (SCCS), the file system, and so on. I guarantee that if the client tries to take those source files, load on your average compiler/linker, and tries to compile and link those files with default arguments, the client will fail spectacularly. This situation is especially true in the embedded systems environment.

Consider a project being developed in a command-line environment on a DOS platform, using a source code control system and MAKE files. What are all the components that have an effect on the system? I can think of several:

- Environment variables in the `AUTOEXEC.BAT` file
- The `.MAK` files themselves
- Potential links from the `.MAK` files to the SCCS software (to check for updates from the library server)
- Potential `.BAT` files that call the compiler/linker with appropriate arguments
- File locations for sources/compiler/linker are "hard-coded" into `.BAT` or `.MAK` files

Of equal importance to commenting code is that of commenting the development environment and all its components. All project areas on disk should include a "Documents" directory, which should contain the following:

- Explanation of file system layout and directory structure—location of source files, target (`.obj`, `.exe`, `.hex`) files, header files, linked libraries, and so on
- Does the compiler/linker need to go in a certain file location?
- Description of all `.BAT` files and `.MAK` files—what do they do?
- Machine requirements for build environment—memory required, environment variables, O/S requirements, brand of compiler/linker, and so on
- Role of and links to the source code control system (SCCS), if one is being used

A GOOD PROGRAMMER

A good programmer is not a programmer who is productive in the short term, churning out code, but is productive in the long term, writing code that is maintainable and reusable for the original programmers as well as other programmers. The likelihood of reusability lies less with the actual code as with its level of documentation. ESP

Steve Drevik is field applications manager for Celerity Systems, Inc. in Knoxville, TN, which develops high-performance digital video servers for interactive video applications in overseas and domestic markets. He has a Master's degree in electrical engineering from the University of Tennessee and has been working in embedded systems since 1988. He can be reached electronically at thedrev@aol.com.