Defensive Programming

Programming style is just as important as any other parameter that goes into a coding task. Here are a few ideas, rants, and raves.

Published in Embedded Systems Programming, January 1993

My New Year's wishes are for world peace, environmental cleanup, a balanced budget, and an end to crummy embedded code. There's not a lot I can do about the first 3 items (hey - I voted for Perot!), but each of us can and should resolve to improve our coding styles.

Industrial strength programming, while a technical activity, is done solely to satisfy a business need. Why does the big boss pay your salary? Her goals are to get a product to market quickly, keep it viable, and (if she is truly enlightened) re-use what is learned and written on the next generation of products.

Those of us writing firmware or managing engineering teams are really the company's technical custodians. Most high level managers don't know or care enough to translate those overriding goals into objectives that the software team can work towards. Personally, I feel a software group's coding-style objectives should be to find a style leading to:

- Fast coding and debugging
- Few bugs over life cycle
- Easy maintenance
- Generate reusable libraries
- Prolong the code's life

The first three items have been beaten to death in the trade press, though far too many programmers don't live and breathe these needs.

My objective of "prolonging the code's life" is meant to avoid the all-too-familiar catastrophe of giving up on the code in a project, and re-writing it from scratch because it is too hacked to be understandable or maintainable. God knows I've made the mistake of permitting code to evolve to chaos over a development or maintenance cycle. How many times have you heard someone critique another's code, only to say "I'll just have to re-code that". This is truly unacceptable. Sometimes this might be the Not Invented Here syndrome, but sometimes the code is indeed so bad that recoding is required. Code that cannot be maintained is worse than no code.

Managers: commit to reading your people's code. Simply delete that which is unreadable or poorly structured. The employees will get the message or move on, solving the problem either way. The cost is initially high and schedules might collapse, but unless you take strong, consistent action the situation will never improve.

With OOP being the buzz-acronym of the 90's, generating reusable libraries is a hot item, but so far not too many companies are really benefiting from it. I'm really impressed by Microsoft and Borland's reusability strategies. I have no idea how they manage projects internally, or how well they do at preserving individual functions. Both outfits sure do a great job of reusing major software components. Look at Borland's TurboVision and Object Windows, that preserve their editors and user interface across a wide range of products. Microsoft goes one better: their Basic is available customized as a word-processor Basic, a spreadsheet Basic, etc. The Windows DRAW is built in to a lot of applications. Microsoft's Word interface looks a lot like Microsoft Works or Microsoft Powerpoint. It's clear some clever folks are snipping off (at least) the major parts of code and applying them to the entire product line. This is the direction of the future.

Variables and Functions

During the last Ice Age Basic and Fortran limited us to 2 to 6 character variable names. Read about it in the

history books. Yet today people still code using classic integers like "I" or "II" or (my personal favorite) "III". If your compiler or assembler forces you into this straitjacket, then buy some decent tools!

Of course, the other extreme is the undisciplined use of long variable names - 60 to 70 characters built up like some Germanic compound word, that takes a lot of head scratching to differentiate from the other 100 similarly constructed variables in that module.

There are two reasons to give a variable a name. 1) to make the damn thing work, and 2) to convey information about how the program works. We can satisfy (1) by using a random number generator to create identifiers. (2) requires lots of common sense.

ReadFromBuffer() seems like a pretty descriptive function name. But from what buffer? Read with a wait? What form does the returned data take?

Maybe ReadByteFromUARTBuffer is better. The names can start to get awkward if you get too descriptive.

Use acronyms and abbreviations, being careful to apply them to common situations only. Generate a glossary of the definitions of every abbreviation and acronym you use. I like PTR for pointer, BUF for buffer, REG for register, etc. You want to convey information quickly.

Consider using Hungarian Notation. Hungarian notation uses lower case characters that indicate the type of a variable as the name's prefix. Then, you can look at any variable and instantly know it's type, crucial in a language like C that lets you get away with murder in parameter passing and creating expressions. Form the rest of the name by capitalizing the first letter of each word.

For example, iGetUartStatus() returns an integer. cOutLcd puts a character to an LCD. sOutLcd outputs a string to the same device.

Success lies in defining a set of standard types and the abbreviation for each. Note that Hungarian even supports structures and other complex types. Just be sure to add the abbreviation for each (like "dow" for DayOfWeek) to the glossary.

Underscores are not really needed with Hungarian Notation.

Global Variables

Don't use globals. Tears come to my eyes looking at the average embedded program, which is invariably riddled with hundreds or thousands of globals. Every little function seems to access this vast hoard of data, changing a bit here or complementing a word there. None of us are smart enough to effectively deal with code written like that.

Globals are a problem because any routine, at any time, can alter the state of any global. There are just too many chances for a careless reference to a variable creating havoc in some remote part of the code.

Perhaps a few globals might be needed, to maintain major state parameters of the system. Limit the number of these.

Use OOP-like packaging (even in C or assembly), keeping the method and data in a module with a bulletproof shell around it. Access to the function's local variables (which can be static) is though a call to the function. Where a major data structure underlies the operation of the entire system, protect that structure by surrounding it with GET and PUT routines. Does it need to be sorted? Protect it with a SORT function.

Files

I'm a DOS person. UNIX is great and all that, but I just don't use it. Therefore, I'm saddled with DOS's crummy file structure.

8 character file names are simply not adequate, but that's all we have to work with. Some friends swear by Norton's cute tricks that associate more information with files, extending names considerably. I consider this a toy, because few applications support the extended naming convention. Who cares if you can get a nice directory listing, but are still stuck with 8 character names in your editor, compiler, and debugger?

Some people come up with elegant naming conventions that use an acronym for the project as the first few characters of each filename - like COM_MAIN, COM_IO, etc. I don't care for this; it unnecessarily wastes a good chunk of the limited name asset.

Still, 8 characters is all we have. Use directories as name extensions. Store all of the file for the COMM-BUFFER project in directory COMM-BUF.

Don't use linked directories. That is, store all of the components of a project in one directory, not in a dozen scattered all over the disk. Using multiple directories seems to create multiple problems. No one is really sure

what file belongs to what project, and backing the project up becomes a nightmare.

However, on a large project it's just not realistic to store hundreds of possibly unrelated files in a single directory. Use the project as a root directory, and store your code in sub directories off of this project node. This concentrates all of the project's assets in one, central location. A single XCOPY transfers the entire project to a backup.

Sharing code is a noble and important goal, but works well only if a certain discipline is enforced. If you plan to use a file of code in several applications, then compile it to a library, strip out all debugging information, and lock the source code in a safe. Get it off the system. Either you plan to share this code or you don't; if shared, any alterations made to the code might have deleterious effects on many projects.

If you cannot bear the thought of turning a function into a true library that no one may modify, realize that you have made the implicit decision to use many versions of the code. You might as well, then, just copy the source to each project directory, and live in peace with the sure knowledge that each copy will evolve in a different direction.

If you write good code, than a goal should be to get rid of the source. Write it, test it, and then recompile it without debugging information. Save the objects in a working directory, and isolate the C code in a source-only directory or on tape. Tested, compiled, and archived objects are useful. Source in the process of being modified, or that is kept around "in case we need to make some changes" invites chaos.

Commenting

I think the oft expressed notion "I don't need comments - I use long variable names" is total hogwash. Comments serve two functions: they describe what is going on in the code and how it works, and (more importantly) they indicate what a function does, and what its parameters are.

Everyone complains about comments, so I'll mostly leave this subject alone. However, the most important programming manual on your desk should be The Elements of Style, by William Strunk and revised by E. B. White.

Make your comments *clear, concise*, and *well organized* . Commenting is a literary experience - you are telling readers a story about your code. Unlike prose from a novel, your comments must also be *correct.*

Avoid long paragraphs. Use simple sentences: noun, verb, object. Use active voice: "iStartMotor actuates the induction relay after a 4 second pause". Be complete.

If writing the code is harder than writing the comments, you are doing something wrong. I generally write all of the comments first and then fill in the code. Even better, write the comments first and hire a low-paid entry-level programmer to pound out some code. Programming is really two jobs: figuring out how to do (ITALICS ON do) something, which is represented in the comments, and converting the comments/method to code.

Backups

12 years of Jesuits and nuns beating "correct-think" into my brain left me with only a single religious belief: back up your work constantly. If backup isn't the first thing you think of in the morning and the last thing before leaving, then get out of this field. Try sheep farming in New Zealand.

Backing up is perhaps not part of a programming style, but is equally a part of the culture and philosophy of being in this business.

Anything can destroy your carefully written code. A wild pointer might wipe 100 MB from your disk; a computer crash can leave you with no way to recover valuable data. Windows 3.0 had a nasty habit of crashing - I found that it often lost a bunch of clusters on my computer, some containing crucial code.

Unfortunately it is rather a pain to back up code, so come up with a procedure that makes it easy. On DOS machines the backup tools are pretty good.

I keep a 3.5" disk in my B: drive all of the time. Being in B: it doesn't bother booting. Several times a day I copy the stuff I'm working on to it. If you change a lot of things, use a utility that flags changed files and does the copy automatically. Once a week I roll the entire disk to tape.

Invest in a lot of tapes and disks. You may not even know that a file is corrupt for months or years, so save the backups for long periods of time.

Get the backups out of your building. Fires, malicious magnets, or other problems can destroy the data. I keep copies at home and in a safe deposit box at the local bank.

Some people back up to a nice big server. This is great if someone copies the server itself to tape frequently,

and copies of the tapes are stored off-site.

Conclusion

"If architects designed buildings the way programmers write code, the first woodpecker that came along would destroy civilization". This is an old joke, but there may be a bit of truth in it. Strive towards perfection; accept only that which is first class. Ross Perot said in one of his interviews that he couldn't tolerate non-performance. I wonder what working for him at EDS was like?