Momisms

Things y our mom should have taught you.

Published in ESP, October 2000

- Return to articles list
- Return to TGG's home page

Momisms

Or,

Lessons Your Mom Should Have Taught You

Momism (môm iz' em) n. ***1.*** *A brief statement of a principle passed maternally.* ***2.*** *A tersely worded statement of an observation of truth: APHORISM*

The image of mom gently guiding her young ones down paths of righteousness, teaching them the basic elements of being civilized, helping with school work, is a powerful one indeed. Yet we still leave home ill-equipped for real life; college itself does but a poor job in preparing us for careers and adulthood. Perhaps mom should have taught us a few more lessons. Here's a few thoughts.

Interrupts

Keep ISRs Short – Debugging interrupt service routines is tough and in some cases almost impossible. Too often those expensive tools work poorly or not at all inside an ISR. Breakpoints fail since they operate at human speeds, while the interrupts come much faster. The old standby of many developers, single stepping, just won't work where interrupts arrive at any sort of reasonable rate. Single step in a section of code where interrupts are reenabled, and you'll likely debug different instantiations of the ISR with each step. An emulator with trace will capture the service routine's execution, but even the largest trace buffers fill quickly when loops and recursion

A very wise friend taught me the fundamental rule of debugging ISRs: don't. Keep the routine so short, so simple, that you can debug by inspection. A good rule of thumb is to limit ISRs to a dozen or so lines. Worst case, keep them shorter than a page. If the ISR really must do a lot of work, why not spawn a task that handles the complexity?

Avoid NMI – Non Maskable Interrupt, aka Trap, level 7, or any of a number of different monikers, can't be shut off, ever. Other interrupt inputs succumb to the "disable" instruction, and generally turn off automatically when a hardware-initiated interrupt occurs. Till you explicitly turn the interrupt back on, the unavoidable non-reentrant parts of the ISR are safe. An NMI handler, however, is never safe. Non-reentrant code will be destroyed if the interrupt reoccurs. Many CPUs use an edge-sensitive input for this beast, so the slightest bit of noise can create multiple, false, NMIs over the course of a few microseconds. And debugging tools, like emulators, often couple small bits of spurious noise into the target system. Reserve NMI for one-time events like the apocalypse or power failure.

Fill unused vectors – Though a CPU might support hundreds of interrupt sources, each one defined via an entry in the dispatch table, we rarely use more than a handful. Leave those unused dispatch table entries blank and any weird vectoring will crash the application horribly… leaving no trail of evidence to the root cause.

Why would spurious interrupts occur? Maybe the hardware is defective or glitchy – it is a prototype during development, isn't it? Perhaps you've misprogrammed one of the hundreds of registers inside of today's too complex peripherals.

Better: fill all unused vectors with a pointer to a debug routine that either logs the erroneous interrupt, or where your debug tools leave a lurking breakpoint.

Listen – don't interrupt others. You'll learn far more listening than talking, and the listener never puts his foot in his mouth.

Bugs

Inspect rather than debug – Bottom line: code inspections find bugs some 20 times more efficiently than debugging by test. Inspect the code, design, specs, and all relevant design documents to find problems before writing/debugging/testing and then chucking a lot of expensive firmware. Inspections won't find all of the problems, but a well-implemented inspection process wrings out 70-80% of the defects for a faction of the cost.

Studies indicate that in too many systems some 50% of the code never gets tested. It's difficult at best to devise test conditions for every error condition/exception handler, and IFs nested 5 deep. Since post-compile error rates run around 5% (5 bugs per 100 lines of code), even a small 10,000 line of code system might have 250 lurking bugs after it's completely "tested". Though devising better tests is surely a good idea, inspections will bring most of these hidden problems to light.

Inspect at 100-200 lines of code per hour – There's a sweet spot at 150 lines of code/hour where inspections proceed very efficiently yet unveil most of the defects. Monitor the inspection rate. These numbers suggest that inspections cost (assuming there's no benefit!) about $2 per line of code, or about 10% of usual $15-30/line cost for most commercial firmware.

Track debugging time – It sure is fun to crank code. But in many organizations some 50% of a project's time gets consumed in debugging. This is a certain sign of dysfunctional development, and indicates the developers either write the code carelessly or spend too little time on specification and design.

Measure bug rates – A few functions or modules typically exhibit most of the product's defects. We've all been there, worked on a function which is so complex, so poorly understood, and so badly coded that we're terrified of opening it in the editor. Change a single character in a comment and it seems the code stops working. Barry Boehm, the guru of software estimation, has shown that these error-prone functions, which typically represent a few percent of the total code base, contain 60-80% of the errors. More compellingly, his data indicates these problem-functions eat up 4 times more effort than their well-behaved brethren. It behooves us take data to quantitatively figure out which are bad, and then toss the code and start again.

Debug proactively – Face it – you're gonna have problems. The code will be far from perfect. *Plan* for bugs and instrument your code to find them quickly. Does your RTOS include a stack-overflow checker? Leave it enabled whilst debugging. Or seed the stack with a pattern, then stop the debugger from time to time to see if stacks are too big or too small.

Why not fill unused ROM/Flash with nasty instructions, like software interrupts, that vector off to a debug routine? When code crashes it often just wanders off, perhaps wandering into your carefully seeded ROM area. The software interrupts and associated handler will capture the crash quickly, and in safety-critical systems can bring the system to a known harmless state.

You'd be surprised how many embedded systems, those that are "done" and shipping, access memory in bizarre ways. Writing to ROM. Reading from unused memory. Though perhaps harmless, these odd behaviors indicate lurking software problems. Consider setting up unused/extra chip selects to trigger on any errant memory access, or expand your PLD decode logic to signal, via an extra output, such problems. Code that behaves unexpectedly, even when the symptoms seem benign, is flawed.

Clean your room – bugs and insects prefer messy places.

Performance and Size

Sometimes engineering costs more than fast hardware – If you're building a million of something, production costs overwhelm development costs. That's much less true for small production runs.

Never forget that one of the "production" costs is that of the amortized engineering. If a design decision adds a month to the project, at perhaps a cost of $20,000, then the product's price must include this additional cost: $20,000 divided by the number of units made. Does an 8051 really make sense for your low-run application? Would a bigger CPU dramatically reduce development costs? Will shoehorning bytes into an undersized code space eat weeks of expensive developers' time?

A tiny CPU is absolutely the perfect choice for a huge range of applications. You just can't beat them for minimizing PCB real estate, recurring costs, and power consumption. And I've long been a proponent of distributing small CPUs around a board to handle small chores like I/O processing. But do understand the very real costs of working in a confined address space with perhaps under-powered tools and languages. Make CPU tradeoffs that minimize total system cost, from engineering through production.

Overload a CPU at your peril – A 90% loaded processor doubles development time. At 95% figure on tripling the effort. This is hardly surprising to our intuitive understanding of programming. With experience we've all battled a performance-bound system by tuning every bit of code it contains, instead of the 20% typically responsible for most of the real time problems. Margins minimize engineering costs by allowing us to be a little sloppy, letting us deliver a product which is not quite tuned to perfection, with the extreme costs that entails.

Create a dynamic model of code size – Few of us really create a meaningful estimate of ROM/Flash needs. Instead we tend to ask for as much ROM as possible, or perhaps double the amount used on the previous project. It's tough to estimate binary sizes when starting a large project.

But we can't abdicate our responsibility to monitor code growth. Telling the boss a month before delivery – when the hardware design is cast in PCBs - that we need more ROM is a sure path to career stagnation.

It's a simple matter to build a spreadsheet that lists all of the modules the system will contain with estimates of their size (in lines of code, function points, or any other reasonable measure). Edit in the real source line and object size of each module as it's completed. Over time you'll find a reasonable approximation to the number of bytes of code per line of C; have the model apply this to the as-yet-uncompleted portions of the code to predict final system size. Odds are you'll spot ROM shortages early on, when there's still time to take design action.

Size doesn't matter – be content with yourself and who you are.

Reuse and Maintenance

Be realistic about reuse – Reuse is hard. Good rules of thumb: Before you can develop code for reuse you must have developed it at least 3 times. Before you can reap the benefits of reuse you must have reused it 3 times. One proposal for Regan's version of the Star Wars missile defense system, which was pegged at 100 million lines of code, was that every module had been used 3 times before being included in the system. Not a bad

that every module had been used 3 times before being included in the system. Not a bad idea, especially for a system that was so hard to test.

Avoid dependencies – Global variables are responsible for most of the evil in the world. A global-infested program becomes non-maintainable, buggy, and a nightmare for all team members. Globals also make reuse all but impossible.

Embedded systems suffer from another dependency problem: code that talks to hardware. Encapsulate all I/O operations.

Self documenting code does not exist – Long variable names do not self documenting code make. Judicious name selection is just a part of good code.

Comment aggressively – Any idiot can write code. Even teenaged hackers manage to crank out working software. Professionals create beautiful code, that which is a joy to maintain and is crystal clear. Accurate, clear comments are an important ingredient of well-written firmware. Code is nothing more than the computereeze description of what's going on; comments are the human description.

Use active voice. Capitalize using standard English rules. Check your spelling. Describe concisely the goes-intas and goes-outas, as well as what happens and why. Some enlightened programmers write all of the comments first, and then fill in the C at their leisure. The hard part, after all, is creating an accurate, documented design. The code is nothing more than a simple translation of a good design to computer-lingo.

Keep compiles clean – Don't come to the dinner table with dirty hands, and don't deliver code reeking of unpleasant warnings.

Why do we tolerate warning messages from our compiler? Firmware lives forever. When someone else opens your code 5 years from now for an upgrade, and finds hundreds of warnings scrolling off the screen, he'll have no idea if the messages are expected or are an effect of the way he's reinstalled the tools. Maintenance is an unavoidable aspect of the software development process; he who programs without maintenance in mind is an amateur.

Keep the code strictly ANSI compliant to minimize warnings and maximize portability. Segment unavoidable deviations from the standard to separate modules which document expected unusual compiler behaviors.

Encapsulate – The OOP folks chant "encapsulation, polymorphism and inheritance"; of those three, encapsulation is the easiest and most powerful tool for building well-written, easy to understand code. It's equally effective in assembly, C, or C++. Bind "methods" (code that accesses a device or data structure) with the data itself.

Floss – you'll miss your teeth when they're gone.

Back to home page.

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Tel: 410-504-6660, Fax: 647-439-1454
Email info@ganssle.com