The Ganssle Group

Perfecting the Art of Building Embedded Systems

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Email info@ganssle.com

Keep It Small

The importance of partitioning. Originally in Embedded Systems Programming, September and October, 1998.

The most important rule of software engineering is also the least known: *Complexity does not scale linearly with size.*

For "complexity" substitute any difficult parameter, like time required to implement the project, bugs, or how well the final product meets design specifications (unhappily, meeting design specs is all too often uncorrected with meeting customer requirements…)

So a 2000 line program requires *more* than twice as much development time as one half the size.

A bit of thought confirms this. Surely, any competent programmer can write an utterly perfect five line program in ten minutes. Multiply the five lines and the 10 minutes by a hundred; those of us with an honest assessment of our own skills will have to admit the chances of writing a perfect 500 line program in 16 hours are slim at best.

Data collected on hundreds of IBM projects confirm this. As systems become more complex they take longer to produce, both because of the extra size and *because productivity falls dramatically*:

(man-yrs)	*Lines of Code produced per month*
1	439
10	220
100	110
1000	55

Look closely at this data. Notice there's an order of magnitude increase in delivery time simply due to the reduced productivity as the project's magnitude swells.

COCOMO Data

Barry Boehm codified this concept in his Constructive Cost Model (COCOMO). He found that:

Effort to create a project = C * KLOC^{M}

(KLOC means “thousands of lines of code”)

Though the exact values of C and M vary depending on a number of factors (e.g., real time code is harder than that for the user interface) both are always greater than one.

A bit of algebra shows that, since M>1, effort grows much faster than the size of the program.

For real time projects managed with the very best practices C is typically 3.6 and M around 1.2. In embedded systems, which combine the worst problems of real time with hardware-dependencies, these coefficients are higher. Toss in the typical poor software practices of the embedded industries and the M exponent can climb well above 1.4.

The human analogy of this phenomena is the one so colorfully illustrated by Fred Brooks in his *The Mythical Man-Month* (a must read for all software folks). As projects grow, adding people has a diminishing return. One of the most interesting reasons is the increased number of communications channels. Two people can only talk to each other; there’s only a single comm path. Three workers have three communications paths; 4 have 6. In fact, the growth of links is exponential: given **n** workers, there are $(n^2-n)/2$ links between team members.

In other words, add one worker and suddenly he’s interfacing in n^2 ways with the others. Pretty soon memos and meetings eat up the entire work day.

The solution is clear: break teams into smaller, autonomous and independent units to reduce these communications links.

Similarly, cut programs into smaller units. Since a large part of the problem stems from dependencies (global variables, data passed between functions, shared hardware, etc.), find a way to partition the program to eliminate - or minimize - the dependencies between units.

Traditional computer science would have us believe the solution is top down decomposition of the problem, perhaps then encapsulating each element into an OOP object. In fact, “top down design”, “structured programming”, and “OOP” are the holy words of the computer vocabulary; like fairy dust, if we sprinkle enough of this magic on our software all of the problems will disappear.

I think this model is one of the most outrageous scams ever perpetrated on the embedded community. Top down design and OOP are wonderful concepts, but are nothing more than a subset of our arsenal of tools.

Top down decomposition and OOP design are merely screwdrivers or hammers in the toolbox of *partitioning* concepts.

Our goal in firmware design is to cheat the exponential in the COCOMO model. We need to use every conceivable technique to flatten the curve, to move the M factor closer to unity.

Partition with Encapsulation

The OOP advocates correctly and profoundly point out the benefit of encapsulation, to my mind the most important of the tripartite mantra *encapsulation, inheritance and polymorphism*.

Above all, encapsulation means binding functions together with the data the functions operate on. It means hiding the data so no other part of the program can monkey with it. All access to the data takes place through function calls, not through global variables.

Instead of reading a status word, your code calls a status function. Rather than diddle a hardware port, you insulate the hardware from the code with a driver.

Encapsulation works equally well in assembly language or in C++. It requires a *will to bind data with functions* rather than any particular language feature. C++ will not save the firmware world; encapsulation, though, is surely part of the solution.

One of the greatest evils in the universe, an evil in part responsible for global warming, ozone depletion and male pattern baldness, is the use of global variables.

What’s wrong with globals? A partial list includes:

- ? Any function, anywhere in the program, can change a global variable at will. This makes finding why a global changes a nightmare. Without the very best of tools you’ll spend too much time chasing simple bugs; even with top-of-the line debuggers time invested chasing problems will be all out of proportion to value received.

- ? Globals create tremendous reentrancy problems

- ? While distance may make the heart grow fonder, it also clouds our memories. A huge source of bugs is assigning data to variables defined in a remote module with the wrong type, or over- and under-running buffers as we lose track of their size, or forgetting to null-terminate strings. If a variable is defined in its referring code it’s awfully hard to forget type and size info.

Every firmware standard - backed up by the rigorous checks of code inspections - must set rules about global use. Though we’d like to ban them entirely, the truth is that in real time systems they are sometimes unavoidable. Nothing is faster than a global flag; when speed is truly an issue, a few, a very few, globals may indeed be required. Restrict their use to only a few critical areas. I feel that defining a global is such a source of problems that every one used should be approved by the team leader.

There’s yet one more waffle on my anti-global crusade: device handlers sometimes must share data stored in common buffers and the like. We do not write a serial receive routine in isolation. It’s part of a fabric of handlers that include input, output, initialization and one or more ISRs.

This implies something profound about module design. Write programs with lots and lots of modules! Don’t lump code into a handful of five thousand line files. Assign one module per logical function: for example, have a single module (file) that includes all of the serial devices handlers - *and nothing else.* Structurally it looks like:

```
     public     serial_in, serial_out, serial_init

serial_in:     code

serial_out:     code

serial_init:     code

serial_isr:     code

      private     data

buffer:     data

status:     data
```

The data items are filescopics - global to the module but private to the rest of the system. I feel this tradeoff is needed in embedded systems to reduce performance penalties of the noble but not-always-possible anti-global tack.

Partition with CPUs

Given that firmware is the most expensive thing in the universe, given that the code will always be the most expensive part of the development effort, given that we're under fire to deliver more complex systems to market faster than ever, it makes sense in all but the most cost sensitive systems have the hardware design fall out of software considerations. That is, design the hardware in a way to minimize the cost of software development.

It's time to reverse the conventional design approach, and *let the software drive the hardware design.*

Consider the typical modern embedded system. A single CPU has the metaphorical role of a mainframe computer: it handles all of the inputs and outputs, runs application code, and services interrupts. Like the mainframe, one CPU, one program, is doing many disparate activities that only eventually serves a common goal.

Not enough horsepower? Toss in a 32 bitter. Crank up the clock rate. Cut out wait states.

Why do we continue to emulate the antiquated notion of "big iron" - even if the central machine is only an 8051? Mainframes were long ago replaced by distributed workstations.

A single big CPU running the entire application implies that there's a huge program handing everything. We know that big programs are bad as they cost too much to develop.

It's usually cheaper to add more CPUs merely for the sake of simplifying the software.

Run the math! Using the COCOMO model, assuming C=1 and M=1.4 (not unreasonable for real time, embedded code), a 10,000 line program has a relative effort of 25. Change this to one 6000 line "main" program, plus two little 2500 line programs running on separate microcontrollers, and the effort falls by 22% to 19. That's a 22% improvement in time to market.

If we reduce M to 1.2 (Boehm's real time number) for the main code, since we've hopefully partitioned much of the real time madness to the smaller controller programs, and keep their values at 1.4, the time to market improvement jumps to 37%.

Bigger programs get more dramatic improvements as we split the code across multiple CPUs.

Follow these guidelines to be successful in simplifying software through multiple CPUs:

? Break out nasty real time hardware functions into independent CPUs. Do interrupts come at 1000/second from a device? Partition it to a controller and to offload all of that ISR overhead from the main processor.

? Think microcontrollers, not microprocessors. Controllers are inherently limited in address space which helps keep firmware size under control. Controllers are cheap (some cost less than 40 cents in quantity). Controllers have everything you need on one chip - RAM, ROM, I/O, etc.

? Think OTP - One Time Programmable - or EEROM memory. Both let you build and test the application without going to expensive masked ROM. Quick to build, quick to burn, and quick to test.

? Keep the size of the code in the microcontrollers small. A few thousands lines is a nice, tractable size that even a single programmer working in isolation can create.

? Limit dependencies. One beautiful benefit of partitioning code into controllers is that you're pin-limited - the handful of pins on the chips act as a natural barrier to complex communications and interaction between processors. Don't defeat this but layering a hideous communications scheme on top of an elegant design.

Communications is always a headache in multiple-processor applications. Building a reliable parallel comm scheme beats Freddy Krueger for a nightmare any day. Instead, use a standard, simple, protocol like I^2C. This is a two wire serial protocol supported directly by many controllers. It's multi-master and multi-slave so you can hang many processors on one pair of I^2C wires. With rates to 1 Mb/sec there's enough speed for most applications. Even better: you can steal the code from Microchip's and National Semiconductor's web sites.

The hardware designers will object to adding processors, of course. Just as firmware folks take pride in producing optimum code, our hardware brethren, too, want an elegant, minimalist creation where there's enough logic to make the thing work, but nothing more. Adding hardware - which has a cost! - just to simplify the code seems like a terrible waste of resources.

Yet we've been designing systems with extra hardware for decades. There's no reason we couldn't build a software implementation of a UART. "Bit banging" software has been around for years. Instead, most of the time we'll add the UART device to eliminate the nasty, inefficient software solution.

Partition by Features

Carpenters think in terms of studs and nails, hammers and saws. Their vision is limited to throwing up a wall or a roof. An architect, on the other hand has a vision that encompasses the entire structure - but more importantly, one that includes a focus on the customer. The only meaningful measure of the architect's success is his customer's satisfaction.

We embedded folks too often distance ourselves from the customer's wants and needs. A focus on cranking schematics and code will thwart us from making the thousands of little decisions that transcend even the most detailed specification. *The only view of the product that is meaningful is the customer's.* Unless we think like the customer we'll be unable to satisfy him. A hundred lines of beautiful C or 100k of assembly - it's all invisible to the people who matter most.

Instead of analyzing a problem entirely in terms of functions and modules, look at the product in the feature domain, since features are the customer's view of the widget. Manage the software using a matrix of features.

List every feature in a spreadsheet, and assign complexity values (in lines of code, function points, or a similar measure) and priority to each. Priority is an honest appraisal of the feature's importance to the success of the product.

In a printer, for example, the first items in the matrix won't really be features; they're things like the shell, the RTOS, perhaps a keyboard handler and communications routines that are basic, low level functions required just to get the thing to start up.

Beyond these, though, are features used to differentiate the product from competitive offerings. Downloadable fonts might be important, but do not effect the unit's ability to just put ink on paper. Image rotation sure is cool but may not always be required.

> The feature matrix insures we're all working on the right part of the project. Build the important things first! Focus on the basic system structure - get all of it working, perfectly - before worrying about less important features. I see project after project is trouble because the due date looms with virtually nothing complete. Perhaps hundreds of functions work, but the unit cannot do anything a customer would find useful. Developers' efforts are scattered all over the project so that until everything is done, nothing is done.

The feature matrix is a scorecard. If we adopt the view that we're working on the important stuff first, and that until a feature works perfectly we do not move on, then any idiot - including those warming seats in marketing - can see and understand the project's exact status.

(I suggested measuring complexity in estimated lines of code. LOC as a unit of measure is constantly assailed by the software community. Some push function points - unfortunately there are a dozen variants of this - as a better metric. Most often people who rail against LOC as a measure in fact measure nothing at all. I figure it's important to measure something, something easy to count, and LOC gives a useful if less than perfect assessment of complexity.)

Most projects are in jeopardy from the outset, as they're beset by a triad of conflicting demands: infinite features, zero time to market, and perfect quality. Meeting the schedule, with a high quality product, that does everything the 24 year old product manager in marketing wants, is usually next to impossible.

80% of all embedded systems are delivered late. Lots and lots of elements contribute to this, but we too often forget that when developing a product we're balancing the schedule/quality/features mix. Cut enough features and you can ship today. Set the quality bar to near zero and you can neglect the hard problems. Extend the schedule to infinity and the product can be perfect and complete.

Too many computer-based products are junk. Companies die or lose megabucks as a result of prematurely shipping something that just does not work. Consumers are frustrated by the constant need to reset their gadgets and by products that suffer the baffling maladies of the binary age.

We're also amused by the constant stream of announced-but-unavailable products. Firms do quite exquisite PR dances to explain away the latest delay; Microsoft' renaming of a late Windows upgrade to "95" bought them an extra year and the jeers of the world. Studies show that getting to market early reaps huge benefits; couple this with the extreme costs of engineering and it's clear that "ship the damn thing" is a cry we'll never cease to hear.

Long term success will surely result from shipping a quality product on-time. Features are the only one leg of the triad left to fiddle. Understand that cutting a few of the less important features will help us get a first class device to market fast.

The computer age has brought the advent of the feature-rich product that no one understands or uses. My cell phone's "Function" key takes a two digit argument - one hundred user selectable functions/features built into this little marvel. Never use them, of course. I wish the silly thing could reliably establish a connection! The design team's vision was clearly skewed in term of features over quality, to the consumers' loss.

If we're unwilling to partition the product by features, and to build the firmware in a clear, high-priority features-first hierarchy, we'll be forever trapped in an impossible balance that will yield either low quality or late shipment. Probably both.

Use a feature matrix, implementing each in a logical order, and *make each one perfect before you move on.* Then at any time management can make a reasonable decision: ship a quality product now, with this feature mix, or extend the schedule till more features are complete.

This means you must break down the code by feature, and only then apply top-down decomposition to the components of each feature. It means you'll manage by feature, getting each done before moving on, to keep the project's status crystal clear and shipping options always open.

Management may complain that this approach to development is, in a sense, planning for failure. They want it all: schedule, quality and features. *This is an impossible dream!* Good software practices will certainly help hit all elements of the triad, but we've got to be prepared for problems.

Management uses the same strategy in making their projections. No wise CEO creates a cash flow plan that the company must hit to survive; there's always a backup plan, a fall-back position in case something unexpected happens.

So, while partitioning by features will not reduce complexity, it leads to an earlier shipment with less panic as a workable portion of the product is complete at all times.

In fact, this approach suggests a development strategy that maximizes the visibility of the product's quality and schedule.

Develop Firmware Incrementally

Demming showed the world that it's impossible to test quality into a product. Software studies further demonstrate the futility of expecting test to uncover huge numbers of defects in reasonable times - in fact, some studies show that up to 50% of the code may never be exercised under a typical test regime.

Yet test is a necessary part of software development.

Firmware testing is dysfunctional and unlikely to be successful when postponed till the end of the project. The panic to ship overwhelms common sense; items at the end of the schedule are cut or glossed over. Test is usually a victim of the panic.

Another weak point of all too many schedules is that nasty line item known as “integration”. Integration, too, gets deferred to the point where it’s poorly done.

Yet integration shouldn’t even exist as a line item. Integration implies we’re only fiddling with bits and pieces of the application, ignoring the problem’s gestalt, until very late in the schedule when an unexpected problem (unexpected only by people who don’t realize that the reason for test is to unearth unexpected issues) will be a disaster.

The only reasonable way to build an embedded system is to start integrating today, now, on the day you first crank a line of code. The biggest schedule killers are unknowns; only testing and actually running code and hardware will reveal the existence of these unknowns.

A unique idiosyncrasy of embedded systems is that the code always interfaces in unusual and often scary ways with custom hardware. You’d think that, since this is almost a defining characteristic of embedded systems, we’d be masters at dealing with these issues; in fact, virtually every project I look at falls flat on its face as soon as hardware and software come together.

We’re rather like bankers who are aghast and surprised when a customer wants to make a deposit. In fact, bankers know and expect depositors so build systems to insure their customers’ needs are taken care of. We embedded folks know and expect troubles integrating hardware and software - so should build systems and methods to minimize the problems.

Both hardware and real time integration always cause trouble. Though firmware in many ways is getting divorced from bit twiddling as products become more complex, hardware requirements too are spiraling as we add DSPs, Internet-connectivity, and other the like to the products.

Tackle the hardware issues first! Get a prototype early. Use your emulator or other debugger as an interactive tool to fiddle with the I/O and understand how it operates. Even off-the-shelf peripherals suffer from too many documentation flaws, such as inverted and transposed bits. Play with the hardware to understand it; then, write and test drivers before dealing with the application code.

Real time integration means building and testing a software design that operates properly in the time domain. As you work with the hardware, bring up an RTOS and the ISRs needed to support the OS and the hardware.

One reader recently complained about commercial RTOSes. His point: every time he changes processors he fights a terrible battle with the new OS. The learning curve to switch between products is steep. Worse, as a user of sometimes poorly-supported exotic CPUs he wastes far too much time working with the vendor in porting the RTOS to the new architecture. A simple, basic multitasker he wrote years ago satisfies most of his requirements, with no learning curve and minimal porting overhead.

He’s right: it takes time and effort to master a new RTOS. Your first design with a multitasking OS will be a nightmare as you deal with both the intricacies of the particular operating system, and the new paradigm of building a tasking application. No matter how expert one becomes, every succeeding RTOS experience burns weeks of time leanring the nuances of the particular OS. This suggest to me that it’s important for companies to make a commitment to a particular processor architecture and toolchain.

Changing either at will costs plenty. The time to integrate a new OS into the design and into your engineering group’s collective brains will be essentially wasted.

I also can’t help but think that using a processor with weak software support is a bad idea. A development process that allows the hardware to crowd make chip selections with no input from software engineering is seriously flawed. The fundamental rule of embedded software is that *firmware is the most expensive thing in the universe.* Clearly, hardware decisions must reflect this truism and be skewed towards selecting components that ease development, not hinder it.

We learned from the PC wars years ago that unsupported OSes and architectures fail. Take that lesson to heart and pick parts with wide support, decent tools, and great off-the-shelf libraries. Take any other path and watch engineering time skyrocket.

Always buy off the shelf products (e.g., the RTOS) to minimize the amount of code you've got to write. Recognize that buying code minimizes rather than eliminates development pain. Figure on dancing with a vendor or three till you find a product that satisfies your needs - and then commit to it.

Real time and hardware integration are the hardest parts of embedded systems. Do them first, before worrying about the application itself.

Conquer The Impossible

Firmware people are too often treated as the scum of the earth, because their development efforts tend to trail that of everyone else. When the code can't be tested till the hardware is ready - and we know the hardware schedule is bound to slip - then the software, already starting late, will appear to doom the ship date.

Engineering is all about solving problems, yet sometimes we're immobilized like deer in headlights by the problems that litter our path. *We simply have to invent a solution to this dysfunctional cycle of starting firmware testing late because of unavailable hardware!*

And there are a lot of options.

One of the cheapest and most available tools around is the desktop PC. Use it! Here's a few ways to conquer the "I can't proceed because the hardware ain't ready" complaint. All require a clear understanding that this is inevitable, so should be dealt with before writing a line of code or drawing the first wire on the schematic.

? One compelling reason to use an embedded PC in non-cost-sensitive applications is that you can do much of the development on a standard PC. If your project permits, consider embedding a PC and plan on writing the code using standard desktop compilers and other tools.

? Write in C or C++. Cross develop the code on a PC until hardware comes on line. It's amazing how much of the code you can get working on a different platform. Using an processor-specific timer or serial channel? Include conditional compilation switches to disable the target I/O and enable the PC's equivalent devices. One developer I know tests over 95% of his code on the PC this way - and he's using a PIC processor, about as dissimilar from a PC as you can get.

? Regardless of processor, build an I/O board that contains your target-specific devices, like A/Ds, etc. There's an up-front time penalty incurred in creating the board; but the advantage is faster code delivery with more of the bugs rung out.

Summary

You'll never flatten the complexity/size curve unless you use every conceivable way to partition the code into independent chunks with no or few dependencies. Partition, partition, partition, by encapsulation, adding CPUs, using an RTOS, by feature management, and then, finally, by top down decomposition.