The Ganssle Group

Perfecting the Art of Building Embedded Systems

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Email info@ganssle.com
© 2000,2001, 2002 The Ganssle Group

Prototyping - Part 1

Prototyping is a critical skill - in hardware and software.
This is part 1 of a two part series on the skill.

Published in Embedded Systems Programming, September 1996

A quiet "cool!" slipped from my lips as the room filled with smoke. I watched the engineer first turn pale, then lean heavily on this elbows. Head hung low, slowly swinging slowly back and forth, he seemed on the verge of a breakdown.

All of 16 years old, I had no understanding of the poor man's torment. With kids, a mortgage, and multiple car loans no doubt his family lived on the American-standard brink of bankruptcy. Now, due to a simple mistake, he had smoked a $40,000 system due for delivery in a week. Later I learned he offered his resignation to the boss, who wisely refused it and sent him back to the lab, with orders to "fix that unit now!"

The company was in the business of selling prototypes (always glamorized by a moniker reflecting a more perfect product) to NASA. Everything was a one-off design; everything was delivered to the customer. The frenetic pace of Apollo brooked no delays.

My undercapitalized employer always spent tomorrow's money on today's problems. There was no spare cash to cover risks. In the case of the smoking ground support box we had no spare system, nor even spare boards. As is so often the case, business issues overrode the laws of physics and human missteps: the prototype simply had to work - period.

Years ago I carried this same dysfunctional approach to my own business. We prototyped products, of course, but did so leaving no room for failure. Schedules had no slack; spare parts were scarce, and people heroically overcame resource problems. In retrospect this seems silly, since by definition we create prototypes simply because we expect mistakes, problems, and, well… failure.

Can you imagine being a civil engineer? Their creations - a bridge, a building, a major interchange - are all one-off designs that simply must work correctly the first time. We digital folks have the wonderful luxury of building and discarding trial systems.

Software, though, looks a lot like the civil engineer's bridge. Costs and time pressures means code prototypes are all too rare. We write the code and knock out most of the bugs. Version 1.0 is no more than a first draft, minus most of the problems.

Though many authors suggest developing version 1.0 of the software, then chucking it and doing it again, now correctly based on what was learned from the first goaround, I doubt that many of us will often have that opportunity. The 90s are just too frantic, workforces too thin, and time-to-market pressures too intense. The old engineering adage "If the damn thing works at all, ship it", once only a joke, now seems to be the industry's mantra.

Besides - who wants to redo a project? Most of us love the challenge of making something work, but want to move on to bigger and better things, not repeat our earlier efforts.

Even hardware is moving away from conventional prototypes. Reprogrammable logic means the hardware is nothing more than software. Slap some smart chips on the board and build the first production run. You can (hopefully) tune the equations to make the system work despite interconnect problems.

Yet some level of hardware and software prototyping remains. Don't believe all of the hardware hype - an awful lot of us still build complicated designs using low integration PALs and logic. Even with the fancy FPGAs and large PLDs we still have to connect a lot of other components (like, a microcomputer!) to get a working system. And, many embedded systems still exploit design tricks to keep costs low; tricks that are inherently risky, which may take a bit of fiddling on the bench to get right.

Software Prototypes

We're paid to develop firmware that is correct - or at least correct enough - to form a final product, first time, every time. We're the high tech civil engineers, though at least we have the luxury of fixing mistakes in our creations before releasing the product to the cruel world of users.

Though we're supposed to build the system right the first time, we're caught in a struggle between the computer's need for perfect instructions, and marketing's less than clear product definitions. The B-schools are woefully deficient in teaching their students - the future product definers - about the harsh realities of working in today's technological environment. Vague hand waving and white-board sketches are not a product spec. They need to understand that programmers must be unfailingly precise and complete in designing the code. Without a clear spec, the programmers themselves, by default, must create the spec.

Most of us have heard the "but that's not what I wanted" response from management when we demo our latest creation. All too often the customer - management, your boss, or the end-user - doesn't really know what's they want until they see a working system. It's clearly a Catch-22 situation.

The solution is a prototype of the system's software, running a minimal subset of the application's functionality. This is not a skeleton of the final code, waiting to be fleshed out after management puts in their two cents. I'm talking about truly disposable code.

Most embedded systems do posses some sort of look and feel, despite the absence of a GUI. Even the light-up sneakers kids wear (which, I'm told, use a microcontroller from Microchip) have at least a "look". How long should the light be on? Is it a function of acceleration? If I were designing such a product, I'd run a cable from the sneaker to a development system so I could change the LED's parameters in seconds while the MBAs argue over the correct settings.

Ground Rules

"Wait" you say, "we can't do that here! We always ship our code!" Though this is the norm, I'm running into more and more embedded developers who have been so badly burned by inadequate/incorrect specifications that even management grudgingly backs up their rapid prototyping efforts. However, any prototype will fail unless the goals are clearly spelled out.

The best prototype spec is one that models risk factors in the final product. Risk comes in far too many flavors: user interface (human interaction with the unit, response speed), development problems (tools, code speed, code size, people skill sets), "science" issues (algorithms, data reduction, sampling intervals), final system cost (some complex sum of engineering and manufacturing costs), time to market, and probably other items as well.

A prototype may not be the appropriate vehicle for dealing with all risk factors. For example, without building the real system it'll be tough to extrapolate code speed and size from any prototype.

The first ground rule is to define the result you're looking for. Is it to perfect a data reduction algorithm? To get consensus on a user interface? Focus with unerring intensity on just that result. Ignore all side issues. Build just enough code to get the desired result. Real systems need a spec that defines what the product does; a rapid prototype needs a spec that spells out what won't be in it.

More than anything you need a boss who shields you from creeping featurism. We know that a changing spec is the bane of real systems; surely it's even more of a problem in a quick-turn model system.

Then you'll need an understanding of what decisions will be made as a result of the prototype. If the user interface will be pretty much constant no matter what turns up in the modeling phase, hey - just jump into final product development. If you know the answer, don't ask the question!

Define the deadline. Get a prototype up and running at warp speed. Six months or a year of fiddling around on a model is simply too long. The raison d'être for the prototype is to identify problems and make changes. Get these decisions made early by producing something in days or weeks. Develop a schedule with many milestones where non-developers get a chance to look at the product and fiddle with it a bit.

The Code

Last month I slammed Java and even took a few swipes at C++, but this doesn't mean I think high level languages are bad. In fact, for a prototype where speed and code size is not a problem I'm inclined to use really high level "languages" like Basic. Excel. Word macros. The goal is to get something going now. Use every tool, no matter how much it offends your sensibilities, to accomplish that mission.

Does your product have a GUI? Maybe a control panel? Look at products like those available from National Instruments (Austin, TX 800-433-3488) and IoTech (Cleveland, OH 216-439-4091). These companies provide software that lets you produce "virtual instruments" by clicking and dragging knobs, displays, and switches around on a PC's screen. Coupled to standard data acquisition boards and a bit of code in Basic or C you can produce models of many sorts of embedded systems in hours.

This is a chance to jump up on my favorite soapbox and rant a bit about the biggest scandal in our industry: software reuse. It ain't happening. Don't think that a particular language will automatically create objects that you'll reuse forever. More than anything, this industry needs a commitment to reuse code, and a sea change away from the Not Invented Here syndrome. Packages like those from National Instruments and IoTech are a cheap, fast, and effective way to leverage someone else's code to get a prototype out in no time flat.

The cost of creating a virtual model of your product, using purchased components, is immeasurable compared to designing, building, and troubleshooting real hardware and software. Though there's no way to avoid building hardware at some point, count on adding months to a project when a new board design is required.

Another nice feature of doing a virtual model of the product is the certainty of creating worthless code. You'll focus on the real issues - the ones identified in your prototyping goals - and not the problems of creating documented, portable, well structured software. The code will be no more than the means to the end. You'll toss the code as casually as the hardware folks toss prototype PC boards.

I mentioned using Excel. Spreadsheets are wonderful tools for evaluating the product's science. Unsure about the behavior of a data smoothing algorithm? Fiddling with a fuzzy logic design? Wondering how much precision to carry? Create a data set and put it in your trusty spreadsheet. Change the math in seconds; graph the results to see what happens. Too many developers write a ton of embedded code only to spend months tuning algorithms in the unforgiving environment of an 8051 with limited memory.

Though a spreadsheet masks the calculations' speed, you can indeed get some sort of final complexity estimate by examining the equations. If the algorithm looks terribly slow, work within the forgiving environment of the spreadsheet to develop a faster approach. We all know, though too often ignore, that the best performance enhancements come from tuning the algorithm, not the code.

Though the PC is a great platform for modeling, do consider using current company products as prototype platforms. Often new products are derivatives of older ones. You may have a lot of extant hardware and software - that works! - in a system on the shelf. Be creative and use every resource available to get the prototype up and running.

Toss out the standards manual. Use every trick in the book to get it done fast. Do code in small functions to get something testable quickly, and to minimize the possibility of making big mistakes.

People

All of us have worked with that creative genius who can build anything, who pounds out a thousand lines of code a day, but who can never seem to complete a project. Worse - the fast coder who spends eons debugging the megabyte of firmware he wrote on a Jolt driven all-nighter. Then there are the folks who produce working code devoid of documentation, who develop rashes or turn into Mr. Hyde when told to add comments.

We struggle with these folks, plead with them, send them to seminars, lead by example, all too often without success. Some of them are prima donnas who should probably get the ax. Others are really quite good, but simply lack the ability to deal with detail… which is essential since, in a released product, every lousy bit must be right.

These are the ideal prototype developers. Bugs aren't a big issue in a model, and documentation is less than important. The prototype lets them exercise their creative zeal, while it's limited scope means problems are not important. Toss twinkies and caffeine into their lair and stand back. You'll get your system fast and they'll be happy employees. Use the more disciplined team members to get the bugless real product to market.

The NASA system I described smoked due to a bad power supply which took out over 100 ICs (socketed, fortunately). The engineer, his resignation rejected, through heroic efforts and countless packs of cigarettes managed to salvage it, delivering it almost on time. Hours later he was deep into another high-pressure project with another impossible deadline and too few resources. He had kids to feed, so unquestioningly accepted the new assignment.

Watching this man's angst I started to see how pressure can shrivel a person's soul. Part of management is effectively using people's strengths while mitigating their weaknesses. Part of it is also giving the workers a break once in a while. No one crank out 70 hour weeks forever without cracking.

The Ganssle Group

Perfecting the Art of Building Embedded Systems

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Email info@ganssle.com
© 2000,2001, 2002 The Ganssle Group

Prototyping - Part 2

Prototyping is a critical skill - in hardware and software.
This is part 2 of a two part series on the skill.

Published in Embedded Systems Programming, October 1996

Last month I talked about developing quick prototypes of software systems. Embedded systems are a unique nexus in computers where the hardware and the software are one. We can never stray far from the electronics that form the physical part of our products.

In many ways hardware is harder to prototype because you actually have to build something. It seems there are always a thousand reasons why something cannot be built in a reasonable time-frame: parts are unavailable (or, we forgot to order them), the PC boards were rejected by incoming inspection, the technicians can't seem to get the board modifications right, we forgot how to burn PALs, etc.

Planning

Engineers have managers, who "run" projects, insuring that resources are available when needed, negotiate deadlines and priorities with higher-ups, and guide/mentor the developers towards producing a decent product on-time. Planning is one of any manager's main goals. Too often, though, managers do planning that more properly belongs to the engineers. You know more about what your project needs than your boss ever will; it's silly, and unfair, to expect him to deal with all of the details.

There are lots of great justifications for a project to run late. In engineering it's usually impossible to predict all of the technical problems you'll encounter! However, lousy planning is simply an unacceptable, though all too common, reason.

I think engineers spend too much time doing, and not enough time thinking about doing. Try spending two hours every Monday morning planning the next week and the next month. What projects will you be working on? What's their status? What is the most important thing you need to do to get the projects done? Focus on the desired goal, and figure out what you need to do to get there. Do you need to order parts? Tools? Does some of your test equipment need repair or calibration?

Find the critical paths and do what's required to clear the road ahead. Few engineers do this effectively; learn how, and you'll be in much higher demand.

When developing a rush project (all projects are rush projects….) the first design step is a block diagram of the each board. From this you'll create the schematic; then do a PCB layout; create a bill of materials, and finally, order parts for the prototype.

Not. The worst thing you can do is have a very expensive quick-turn PCB arrive, with all of the components still on back order. The technicians will snicker about your "hurry up and wait" approach, and management will be less than thrilled to spend heavily for fast-turn boards that idle away the weeks on a shelf.

Buy the parts first, before your design is complete. Surely you'll know what all of the esoteric parts are - the CPU, odd analog components, sensors, and the like. These are likely to be the hardest and slowest to get, so put them on order immediately.

The nickel and dime components, like gates and PALs, resistors and capacitors, are hard to pin down till the schematic is complete. These should mostly be in your engineering spares closet. Again, part of planning is making sure your lab has the basic stuff needed for doing the job, from soldering irons to engineering spares. Make sure you have a good selection of the sort of components you company regularly uses, and avoid the temptation to use new parts unless there's a good reason.

PCB Issues

Time Magazine, Business Week, Fortune, and our upper management should all form a choir to sing the great hit of the 90's "The Time To Market Blues":

My market share is fallin'
And the bank's come a callin'.

If we don't ship it promptly
We're gonna be his-to-ry.

Chorus: We're gonna be his-to-ry

So design that schematic;
Crank some code while you're at it.

But ship by tomorrow morn!

Chorus: But ship by tomorrow morn!

Ahem. Though we hear them singing, sometimes we continue to do things in the same old tired ways. One of these is the traditional wire-wrap breadboard.

In the bad old days we created wire wrapped prototypes because they were faster to make than a PCB, and a lot cheaper. This is no longer the case. Except for the very smallest boards, the cost of labor is so high that it's hard to get a wire wrapped prototype made for less than $500 to several thousand dollars. Turnaround time is easily a week.

Cheap autorouting software means any engineer can design a PCB in a matter of a couple of days - and, you'll have to do this eventually anyway, so it's not wasted time. Dozens of outfits will convert your design to a couple of PCBs in under a week for a very reasonable price. How much? We generally pay $1000-$1500 for a 50 square inch 4 to 6 layer board, with one week turnaround.

It's magic. Modem your board design to the vendor, and days later Fedex delivers your custom design, ready for assembly and test.

PCBs are much quieter, electrically, than their wire wrapped brethren. With fast rise times and high clock rates noise is a significant problem even in small embedded designs. I've seen far too many cases of "well, it doesn't work reliably but that's probably due to the wire wrap. It'll probably get better when we go to PC." These are clearly cases where the prototype does not accomplish it's prime objective: identify and fix all risk factors.

The best source for information about speed and noise issues on PC boards is "High Speed Digital Design (a Handbook of Black Magic)" by Howard Johnson and Martin Graham (1993 PTR Prentice Hall, NJ).

Design your prototype PCB with room for mistakes. Designing a pure surface mount board? These usually use tiny vias (the holes between layers) to increase the density. Think about what happens during the prototyping phase: you'll make design changes, inevitably implemented by a maze of wires. It's impossible to run insulated wire through the tiny holes! Be sure to position a number of unusually large vias (say, .031") around the board that can act as wiring channels between the component and circuit sides of the board.

Add pads for extra chips; there's a good chance you'll have to squeeze another PAL in somewhere. My latest design was so bad I had to glue on five extra chips. Guess who felt like an idiot for a few days…

Always build at least two copies of each prototype PCB. One may lag the other in engineering modifications, but you'll have options if (when) the first board smokes. Anyone who has been at this for a while has blown up a board or two.

I generally buy three blank prototype PCBs, assemble two, and use the third to see where tracks run.Though sometimes you'll have to go back to the artwork to find inner tracks, it sure is handy to have the spare blank board on the bench during debug.

Design For Debug

Though it might be interesting to get into a philosophical debate about the nature of man, suffice to say we're all less than perfect, as manifested in our less than perfect designs. Whether we're writing code or designing a board, perfection is an elusive (impossible?) goal.

Knowing this, why do we design systems that are so hard to work on? Face it: you're going to spend a lot of time troubleshooting your creations. Design the system from the outset for ease of access and to simplify debugging.

Not many designs include a ground point, yet there's no question that we'll spend a lot of late nights wielding a scope probe. With high density SMT it's almost impossible to get a decent ground by soldering a resistor lead to the corner of a chip. Include a convenient ground point. On a big board scatter several around, as you'll want to keep the ground lead short.

Include other test points as appropriate. Make sure critical signals you always use for triggering test equipment are easily accessible. We've started adding a special connector, that is not loaded on production versions, for the logic analyzer. It's designed so the analyzer we're standardized on here plugs directly into the socket, giving us immediate and complete access to the most important nodes.

If you plan to use an emulator, is the CPU socket covered by another board? Ditto for the ROM sockets when a ROM emulator is the debugger of choice.

Many of Motorola's CPUs have a "BDM" port, a simple debugging interface that's built right into the chip. Connect this to a small 10 pin connector and you've got a cheap and effective software debugging tool. No matter what sort of debugger you plan to use, include this connector. If you don't load it on production PCBs the costs will be vanishingly small, yet the debugger will always be there, ready for action if needed.

The 186 family (from AMD and Intel) has a feature that allows you to tri-state surface mounted CPUs, easing the pain of connecting an emulator. The emulator will drive RESET to tri-state the CPU… as will your logic. Design your logic so both sources can effectively drive RESET, otherwise your debugging options will be severely limited. (For more info on this check out http://www.softaid.com/reset.html).

If your budget is tool-poor by all means add a simple debug port - an 8 bit I/O port with LEDs or a scope connection - that the code exercises as it runs. You'll get at least some indication of what is going on. Without insight into the system's operation debugging is all but impossible.

Making Changes

After spending a couple of months writing code it's a bit of a shock to come back to the hardware world. Fixing bugs is a real pain! Instead of a quick edit/compile you've got to break out a soldering iron, wire, parts, and then manipulate a pin that might be barely visible.

PALs, FPGAs, and PLDs all, to some extent, ease this process. Many changes are not much more difficult than editing and recompiling a file. It is important to have the right tools available: your frustration level will skyrocket if the PAL burner is not right at the bench.

FPGAs that are programmed at boot time via a ROM download usually have a debugging mechanism - a serial connection from the device to your PC, so you can develop the logic in a manner analogous to using a ROM emulator. Be sure to put the special connector on your design, and buy the little adapter and cable. Burning ROMs on each iteration is a terrible waste of time.

PLDs often come like EPROMs, in ceramic packages with quartz erasure windows. These are great… if you were clever enough to either socket the parts, or to have left room around the part for a socket.

On through-hole designs I generally have the technicians load sockets for every part on the prototype. I want to replace suspected failed devices quickly, without spending a lot of time agonizing over "is it really dead?"

Sockets also greatly ease making circuit modification. With an 8 layer board it's awfully hard to know where to cut a track that snakes between layers and under components. Instead, remove the pin from the socket and wire directly to it.

You can't lift pins on programmable parts, as the device programmer needs all of them inserted when reburning the equations. Instead, stack sockets. Insert a spare socket between the part and the socket soldered on the board. Bend the pins up on this one. All too often the metal on the upper socket will, despite the bent-out pin, still short to the socket on the bottom. Squish the metal in the bottom socket down into the plastic to eliminate this hard-to-find problem.

Conclusion

With a bit of careful planning, an admission of our lack of perfection, and a clever approach to debugging, prototyping hardware will still be hard! Practice and invention will make each project easier than the last. Be creative.