How to Become an Embedded Geek

Version 6 – Feb, 2007 – Added info on Freescale kit
Version 5 – Oct, 2006 -Added info about new Edition of Programming Embedded Systems
Version 4 - Feb, 2006 - Added info about Zilog's Z8 board and project ideas link
Version 3 - Oct, 2005 - Added info on Computer Science Lab
Version 2 – Dec, 2004 - Added many links to resources
Version 1 – Initial release

Jack G. Ganssle
jack@ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 504-6660
fax (647) 439-1454

Disclaimer: I love this field. It's a ton of fun. But jobs can be hard to find. Don't take the following as advice to toss your current career to the four winds and jump into the embedded industry unless you do such from love, for unhappily, long periods of unemployment are far from unknown in this industry.

I'm writing this while at anchor in Bermuda. There's no net access aboard, so once or twice a week I head for an Internet Café ashore and dig through the email avalanche. For some inscrutable reason lately I've been drowning in emails from embedded system wannabees. "Dear Jack: I lately learned Visual C++ and now want to start a career in firmware. But no one wants to hire me as I have no experience. What do I do? How to I learn about firmware?"

Perhaps my experience was atypical. I helped midwife the embedded business, learning while building products using the very first microprocessors. Like mastering the mysteries of the birds and the bees, I ran experiments, checked the results, talked to friends, and iterated till achieving some level of mastery.

Times are different now. Then, we were all amateurs. Today expectations are higher, competition for available positions brutal. Lately the staggering economy spawns few new job opportunities; those that surface are more often taken by experienced engineers than newbies.

So for people making the transition from college to real life I recommend hiding out for a year or two, if you can afford it. Consider getting an MS degree. If your BS is in Computer Science, take EE classes. Since the job market is so depressed it makes sense to optimize your skills to compete better when good times return.

And they will return. It seems we hit a bad economic patch in the beginning of each decade. Each one feels overwhelming, but they all pass. In the early 70s engineering collapsed with the loss of the Apollo program. Inflation and other woes caused a big contraction around 1980. The recession of the early 90s killed the elder Bush's reelection bid. Our current problems, too, will pass, fading into an ugly but almost forgotten memory.

The Long and Winding Road

Too many of my email correspondents are looking for shortcuts. "How do I convince a potential boss to hire me?" "What book can I read to teach me firmware?" Sorry – there's no easy path, no way to pass Go and collect $200. Though the bookstores have plenty of titles like "Learn to Program in 21 days," don't expect to see an equivalent book for embedded systems.

Perhaps there is one easy way: get an EE degree. All other approaches will be harder. The degree gives you instant credibility in the marketplace, though a newly-minted sheepskin commands bottom-of-the rung salaries.

I wonder if this search for shortcuts is a quintessentially American characteristic. We're so anxious to do things *today*! As a parent I'm constantly astonished to find how kids need to go through so many experiences themselves. They can't learn from my hard-won insights. There's something intrinsic to humans about learning by doing.

Maybe you're a master of C++, a whiz at programming in Windows using MFC and the wealth of tools we expect of any desktop computer. That's a fantastic, valuable skill. It does not translate to an "in" into this field. C or C++ are base level skills for any firmware developer, but are merely a subset of the required expertise.

For there's a huge gulf between the resource-rich environment of a desktop machine and a typical embedded system. It's tough to generalize about firmware, because some projects run on 4 bit micros in 100 words of code, while others boot complete Windows or Linux operating systems. But I believe there are some skills shared by all of the best firmware developers.

First is the ability to work with limited resources. ROM and RAM may be very costly in high volume or low power products. Where on a desktop heaps and stacks are seemingly infinite, we firmware folks sometimes trim each to a razor's edge, too often with catastrophic results. The Windows developer knows how to speed transcendental math using lookup tables, but his embedded counterpart looks askance at the sometimes staggering memory requirements. We use C subsets on minimal processors, where sometimes the pseudo-C is just a cut above assembly. Processors with poor stack architectures invariably spawn compilers that play complex games with automatic variables, games that can and will bite back when used by the unwary developer.

It's not uncommon for time to be in short supply as well. There are limits to how fast a small processor can move data around; Moore's Law does not bring the embedded developer a faster CPU every few months. When the system is performance-bound, embedded engineers re-design code, tune routines, and even at times change the hardware design.

So the accomplished firmware developer is a master of cramming more into less: more features into less memory, more performance into fewer CPU cycles. Assembly language always lurks, if only to understand what the compiler generates.

Embedded systems interact in complex and strange ways with the system's peripherals. We're not downloading drivers from some vendor's web site, or relying on a vast infrastructure of OS support. Design a simple data acquisition system and odds are you'll have to initiate A/D conversions, suck in data, and scale and normalize it. Working with a serial channel expect to write your own circular queue handlers.

We firmware folks are responsible for even the most basic of all functions. On most processors we must set up chip select pseudo-peripherals to determine the location and extent of all memory devices, as well as the number of wait states required.

Peculiar devices challenge even the most experience of developers. Pulse width modulated outputs aren't uncommon yet defy many people's understanding. Log amp compression circuits scale inputs in confusing ways. Even the straightforward switch behaves very strangely, it's output bouncing for many milliseconds after it's depressed.

Beyond simple I/O, though, the realities of our systems means we must be masters of interrupts and DMA transfers. Data arrives from a plethora of asynchronous sources and must get routed out as needed. Embedded systems people are expected to be competent at writing ISRs, and must understand how to create reentrant code.

Many firmware applications multitask, generally employing some sort of a real time OS. None of these offer the depth of support common to desktop systems; though some of the commercial OSes give a very complete framework for embedded work, they all look remarkably austere compared to Unix or Windows. Even today many embedded apps don't and can't use an RTOS, but the well-rounded developer must be versed at multitasking.

Debugging is especially difficult in the embedded world. If you've been spoiled by Microsoft's debugger, expect culture shock when trying to peer into the workings of your firmware. In the best of cases there's darn little visibility into the workings of our code. Sometimes we're required to amend the hardware and software design just to make some sort of debugging possible – even be it so humble as wiggling pins and monitoring their states on an oscilloscope. If you can't debug, you can't make your stuff work, so plan to understand ICEs, BDMs, scopes and logic analyzers.

Embedded apps are – or should be – much more reliable than their desktop counterparts. A Word or Excel crash doesn't compare in litigation possibilities to an avionics problem that kills hundreds. Less dramatic failures are just as serious. Having to stop every 20 miles to reboot your car controller is simply unacceptable. Few PC applications run for more than a few hours at a time, so memory leaks can often go undetected by the average user who turns the machine off each night. By contrast, an embedded product might have to run for years without cycling power.

To build reliable code we must understand and practice more extensive design than is common in other software projects. Failure Mode Analysis is required for some products. Extensive exception handling is a must. Code coverage tests are mandated for high-rel projects.

Become an expert C/C++ programmer. Gain competency at assembly language. Master all of the above. That gives you the basic skills needed for firmware development.

Changing Careers

An embedded occupation can be lots of fun, personally satisfying, a creative outlet, and reasonably financially rewarding. A lot of folks see these desirable traits after embarking on other vocations and search for ways to make a mid-life change. For most of these people, various family responsibilities make going back to college for an EE or CE degree impossible. In this case you have to design your own curriculum and advance your own career strategy.

First, read as much as you can. Here's a few suggestions.

Bebop to the Boolean Boogie, Clive Maxfield - A fun and interesting digital design book aimed at folks wanting to understand the hardware.

C Programming for Embedded Systems, Kirk Zurell – A good introduction to working with small systems like the 6805, 6508 and PIC.

Embedded Systems Building Blocks, Jean LaBrosse – a great into to writing peripheral handlers. It also includes his firmware standard, a wonderful model for writing code in a consistent manner.

Embedded Systems Design, Arnold Berger - A nice intro to the embedded world, with a focus on tools. Also has good hints on selecting processors.

An Embedded Software Primer, David Simon – This is the best introductory book available. Extremely highly recommended.

Guidelines for the Use of the C Language in Vehicle Based Software, by MISRA - This is a list of dos and don't dos for writing reliable C code. Not a book per se, but a hundred page list of rules. All will make you think.

High Speed Digital Design, Howard Johnson and Martin Graham - The best book available about high-speed issues. The focus is entirely on hardware issues in fast systems.

MicroC/OS-II, The Real Time Kernel, Jean LaBrosse - Jean LaBrosse. The best book on real time operating systems. A must-read.

Programming Embedded Systems in C and C++, Michael Barr and Anthony Massa – An extremely good introduction to the subject, with projects for self-study. In October, 2006 the second edition came out, which is a complete rewrite. There's much more information than before, and it uses the GNU toolchain to keep costs down for readers who wish to practice in sync with reading the book. This is possibly the best introductory text on the market.

Serial Port Complete, Jan Axelson - A very complete reference to serial communications. Handling serial data is a basic skill for every developer.

Reviews of these books, plus many others, are at http://www.ganssle.com/bkreviews.htm.

Broke? Download Motorola's introduction to microprocessors (http://e-www.motorola.com/files/microcontrollers/doc/ref_manual/M68HC05TB.pdf). It's a fabulous 300+ page book that gives basic insight into many aspects of working with microprocessors.

Read embedded.com and chipcenter.com regularly. Scan every issue of Dr Dobb's Journal, Embedded Systems Programming, and EE Times magazine.

Read code, too, to see how experienced developers actually make things work. There's plenty scattered around the web, on sites like 8051.com, chipcenter.com, microchip.com, and the like. I especially recommend reading the source to ecos (redhat.com), an open-source RTOS. Even better, read the source to Jean LaBrosse's UC/OS, available in the previously-mentioned book and on ucos-ii.com. Both of these operating systems are the very model of how we must write code – beautiful, well documented, clear, concise, and extensible.

Computer Science Lab

If you're anxious to learn the elements of 8051 programming, basic C, and basic C++, check out John Koplin's Computer Science Lab (http://www.computersciencelab.com), a $19 CD for Windows that contains three separate courses that take the newbie from no real knowledge of programming to a working knowledge of assembly, C and C++.

The Lab comes with a 8051 simulation environment and IDE for assembly language.A hyperlinked lesson plan implemented in Microsoft Help files instructs the student in both the use of the simulator and in the nature of the 8051. A page or two of very basic electronics sets the stage, followed by a very high level tour of machine and assembly, addressing modes and hex. The detail is just enough to give a non-technical manager a sense of our world.

From there the course dive into the processor's architecture and a basic assembly language. Each lesson is short and revolves around an example program. Each introduces a few more instructions and programming concepts. Students use the simulator to run the supplied examples, and to extend these small programs.

Figure 1 shows the simulator's IDE. A user can step into, step over, set and remove breakpoints, right click to change a register or data item. It's quite conventional and reasonably complete.

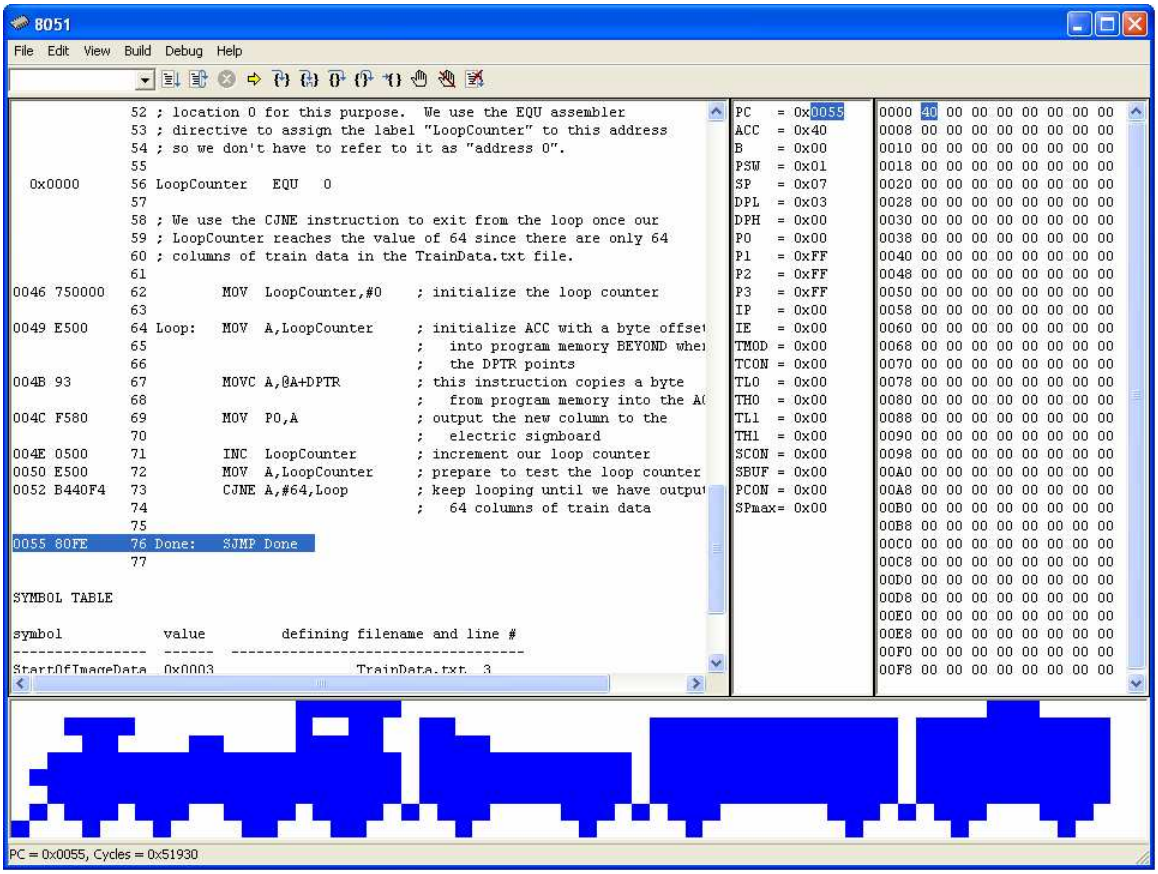

Figure 1: 8051 Simulator

The first program is a simple infinite loop. Then a counter, then another that computes factorials. In many cases the programs mirror those implemented with the RPN calculator, showing clearly the difference between building code in different languages.

But he quickly gets to fun, more engaging programs, such as one that uses PIO bits to "draw" a moving train on the output window (bottom window of Figure 1). Fun is important to keep interest levels high, especially in attention-starved adolescents.

The course goes deep and doesn't neglect uniquely embedded issues like working with the 8051's internal peripherals. Simulated timers, UARTs and the like interact with the programs just as they would on a real chip. It even covers interrupts; the simulator provides periodic interrupts which the code can capture and process.

While the RPN calculator section ends with a nice summary, the 8051 segment does not. That's a shame. We learn best when you tell them what you're going to tell them, tell them, and then summarize what you just told them. Instead this part ends somewhat abruptly after describing the final interrupt program.

From there we leave the truly embedded world behind and move to another IDE, one running C/C++ under Windows. Though this IDE looks something like that offered by

Microsoft's Visual Studio it's really a shell around GNU C++ compiler and GDB. The interface prettifies the raw GNU tools, but doesn't completely isolate you from the very powerful but maddeningly complex command set. Buttons control most common functions like single stepping, but for many other operations (like to view a variable) you must issue a GDB command in the scrolling text window.

John's approach to teaching C and C++ is much like that for 8051 assembly language. A serious of graduated lessons with examples, all of which run on the IDE, takes the student from no C/C++ knowledge at all to that of an apprentice programmer. Mastery will take more practice, more trial programs, and a bit more depth than provided. But you will be writing real, useful programs by the end of the course.

This section has two parts – basic C/C++ working in a DOS command window, and then the elements of writing Windows code. I tried – I really tried – some time ago to write a Windows program. Visual Studio even created a complete, working "Hello World" program. But where was the printf? A search found it sitting in a puzzling resource file. I guess I'll stick to embedded.

This course cleared up the mysteries. It's not easy as even the simplest bit of Windows code uses an awful lot of cryptic API calls. But this course will get you building simple apps in Microsoftland… ironically while working with GNU tools.

John writes conversationally and engagingly. He doesn't resist the chance to editorialize, bashing Microsoft, Sun, Apple, Java slightly to the course's detriment. He detests iostreams, operator overloading, and exceptions. Don't expect to learn about these concepts. You won't deal with RTOSes, reentrancy, and other advanced concepts, but will master a *lot* using examples that are much more appealing to engineers than the contrived ones always found in C/C++ books.

Build Stuff

Book knowledge is crucial but complement it with practical experience. Do projects. Build things. Make them work. Expect problems, but find solutions. Don't abandon a project because it's too hard or you're confused. Most real development efforts are plagued by what initially appears as insurmountable problems, that the boss demands we overcome.

Remember your first crystal radio? I was 10, and wound wire around a toilet paper tube to form the inductor. Those were indeed the olden days so the only "active" element was a galena crystal grazed by a wire whisker. The thrill of picking up an AM station in the earpiece is something I still remember. Today you can get a kit for this sort of radio for only $11 from http://www.elenco.com/. Fortunately galena is out and a diode is in, so the device performs a bit better and is much easier to use and construct. They also offer simple but cool $20-$35 robot kits, and a wealth of other projects. Definitely check out this site.

Forest Mims created a $60 course sold by Radio Shack which offers a hands-on introduction to electronic circuits (http://www.radioshack.com/product.asp?cookie%5Ftest=1&catalog%5Fname=CTLG&product%5Fid=28-280). Readers tell me the material is not as well-written as most of Mims' work, but the material is very complete and works one up to the level of a freshman EE lab class.

Parallax, home of the ever-popular Basic Stamp, offers a variety of kits aimed squarely at the education market (www.parallax.com). Their Understanding Signals kit, for instance, includes a simple oscilloscope, a bag of parts, and an excellent manual guiding one through the mysteries of signals. You will need a Basic Stamp computer module to run all of the experiments. The entire manual is free at http://www.parallax.com/dl/docs/prod/sic/Signals.pdf.

The company's $169 Basic Stamp Discovery kit includes everything needed to learn about building simple embedded systems, with parts and instructions for some 40 projects. Yes, the language is Basic. But that's a lot more accessible to teens than assembly or C, especially since Basic is interpreted. There's no compile/link/download cycle to interfere with the fun of building and testing. Think Doom rather than chess.

Do check out their PING))) ultrasonic sensor – a $25 sonar module that ranges at distances up to 11 feet and requires only a single I/O pin. And there's the $30 Compass Module, a dual-axis unit that can be the start of a replacement for the standard-issue Boy Scout compass. I can think of a million projects based on these sensors and hope to find time to play with them in the future.

Freescale has an interesting and inexpensive ($100) dual processor kid available. See: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=LFEBS12UB&fsrch=1 .

The venerable 8051 is still used everywhere by everyone. If you're willing to teach the essence of 8051-work, there are a lot of cheap development platforms listed at http://www.8052.com/links.phtml.

TI sells a variety of development kits for their wonderful MSP430 series of processors. These 16 bitters offer a lot of performance for little money. I've got their $49 Flash Emulation Tool (http://focus.ti.com/docs/toolsw/folders/print/msp-fet430x110.html) and find it's a great platform for experimenting. The kit comes with a compiler and debugger, though stripped down to handle only small programs.

The ubiquitous PIC processor has a lot of support. One site that offers inexpensive development boards, plus a good book, is http://www.downtowninternet.com/elproducts/products.htm.

Imagine Tool's Microprocessor Starter Kit (http://imaginetools.com/products/MicroStarterKit.shtml) is a Rabbit-based board with

parts and instructions. Build a ranger, GPS clock, thermostat control and many other things. The Rabbit comes with Dynamic C, and interactive version of the language that doesn't require explicit link and download steps. When my son had to build a school project that illuminated light bulbs in various patterns based on switch settings, I suggested using a computer instead of a lot of wires and diodes. We stuffed an ImagineTools board into the project. He knew no C, so I wrote the device drivers and showed him the elements of the language. In a few hours he had a working system, one that apparently worked around the point of the lesson and nicely ticked off the teacher.

http://www.avrfreaks.net offers several hundred projects, all free, all on-line, ranging from the deadly dull innards of filesystems to robotics, MP3 players, telescope controllers and much more. This very deep site devoted entirely to the AVR line of processors also lists tools, free and otherwise. Read the "Newbie's Guide to AVR Development" and pass the wisdom on to an interested kid.

Check out the inexpensive PIC and AVR boards at www.dontronics.com, and follow his newbie advice at http://www.dontronics.com/auto.html.

Want ideas for a project to build? Check out http://instruct1.cit.cornell.edu/courses/ee476/ideas/EE476.project.ideas.html.

Zilog has a Z8 development board for an unbelievable $40 (http://www.zilog.com/products/partdetails.asp?id=Z8F08200100KIT), available from Digikey.

For parts and supplies it's hard to beat Digikey (http://www.digikey.com), which has everything electronic, from resistors to complete development platforms. I find their web site very hard to use; get a (free) printed catalog. Small Parts (http://smallparts.com/) is a great source for petite gears, drives, bearings and more. The mother of all sites for anything mechanical is McMaster-Carr (http://www.mcmaster.com/), whose service is astonishing (I get my orders the next day without paying for expedited shipments). They have everything. Need a bit of stainless angle iron? No problem. Fasteners? You'll find all sorts on-line. Tools, plastic, rubber, gears, handles – it's all there. The printed catalog, which isn't needed as the site is so good, runs over 4000 pages, so you get the idea of the extent of their products.

You'll be on the way to mastery when the programs become large, not from lousy implementations, but due to the demanding nature of the project. I figure that a 1000 line project will teach a lot, but by the time the code reaches 5-10,000 lines of code it starts to resemble a simple but real-world app.

Conclusion

Without an appropriate degree, expect to work for a time as an intern or apprentice. Your salary will drop till you can acquire and demonstrate your competence. I suspect few people can avoid this painful reality.

Don't be afraid to ask lots of questions… and be determined to move ahead.

Don't be afraid to advertise your career dreams; let your current boss know you want to get into the embedded side of the business (if there is one). These days few companies are hiring, so it's easier to make a lateral move from within the organization.

This sounds like a huge amount of work, and it is. If it's too much, maybe you're not cut out for the embedded industry. I suspect that most great developers succeed because they love doing the work. Indeed, various salary surveys show that, for engineers, money is one of the least important motivating factors. Doing cool projects inevitably ranks first.

Is it fun all of the time? Of course not. We pay for the thrills by wading through mind-numbing technical articles and putting up with unenlightened bosses. But if you love technical challenges, fighting really tough problems that span the range from hardware to software to even the basic science of some devices, embedded is the field for you. Be tough, be determined, think long-term… and have fun.