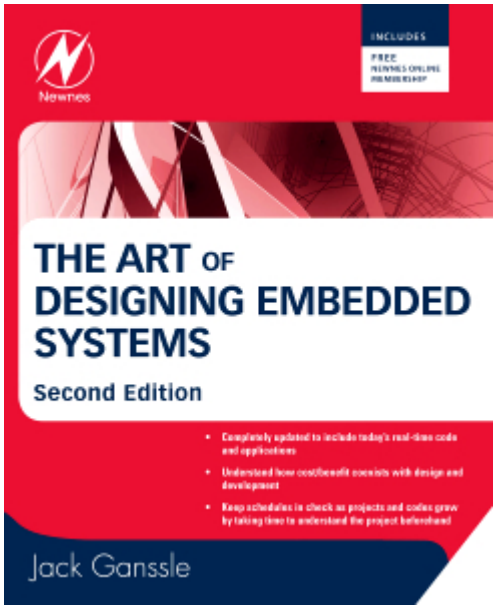
THE ART OF DESIGNING EMBEDDED SYSTEMS, 2ND EDITION

By Jack Ganssle

May 2008 ▪ ISBN 978-0-7506-8644-0▪ Paperback ▪ 308 Pages ▪ $69.95



Features:

- Authored by Jack Ganssle, Tech Editor of Embedded Systems Programming and weekly column on embedded.com
- Keep schedules in check as projects and codes grow by taking time to understand the project beforehand
- Understand how cost/benefit coexists with design and development

Jack Ganssle has been forming the careers of embedded engineers for 20+ years. He has done this with four books, over 500 articles, a weekly column, and continuous lecturing. Technology moves fast and since the first edition of this best-selling classic much has changed. The new edition will reflect the author's new and ever evolving philosophy in the face of new technology and realities.

Now more than ever an overarching philosophy of development is needed before just sitting down to build an application. Practicing embedded engineers will find that Jack provides a high-level strategic plan of attack to the often times chaotic and ad hoc design and development process. He helps frame and solve the issues an engineer confronts with real-time code and applications, hardware and software coexistences, and streamlines detail management.

CHAPTER 1

Introduction

For tens of thousands of years the human race used their muscles and the labor of animals to build a world that differed little from that known by all their ancestors. But in 1776 James Watt installed the first of his improved steam engines in a commercial enterprise, kicking off the industrial revolution.

The 1800s were known as "the great age of the engineer." Engineers were viewed as the celebrities of the age, as the architects of tomorrow, the great hope for civilization. (For a wonderful description of these times read *Isamard Kingdom Brunel*, by L.T.C. Rolt.) Yet during that century, one of every four bridges failed. Tunnels routinely flooded.

How things have changed!

Our successes at transforming the world brought stink and smog, factories weeping poisons, and landfills overflowing with products made obsolete in the course of months. The *Challenger* explosion destroyed many people's faith in complex technology (which shows just how little understanding Americans have of complexity). An odd resurgence of the worship of the primitive is directly at odds with the profession we embrace. Declining test scores and an urge to make a lot of money now have caused drastic declines in US engineering enrollments.

To paraphrase Rodney Dangerfield: "We just can't get no respect."

It's my belief that this attitude stems from a fundamental misunderstanding of what an engineer is. We're not scientists, trying to gain a new understanding of the nature of the universe. Engineers are the world's problem solvers. We convert dreams to reality. We bridge the gap between pure researchers and consumers.

Problem solving is surely a noble profession, something of importance and fundamental to the future viability of a complex society. Suppose our leaders were as single-mindedly dedicated to problem solving as is any engineer: we'd have effective schools, low taxation, and cities of light and growth rather than decay. Perhaps too many of us engineers lack the social nuances to effectively orchestrate political change, but there's no doubt that our training in problem solving is ultimately the only hope for dealing with the ecological, financial, and political crises coming in the next generation.

My background is in the embedded tool business. For two decades I designed, built, sold, and supported development tools, working with thousands of companies, all of which were struggling to get an embedded product out the door, on-time, and on-budget. Few succeeded. In almost all cases, when the widget was finally complete (more or less; maintenance seems to go on forever due to poor quality), months or even years late, the engineers took maybe 5 seconds to catch their breath and then started on yet another project. Rare was the individual who, after a year on a project, sat and thought about what went right and wrong on the project. Even rarer were the people who engaged in any sort of process improvement, of learning new engineering techniques and applying them to their efforts. Sure, everyone learns new tools (say, for ASIC and FPGA design), but few understood that it's just as important to build an effective way to design products as it is to build the product. We're not applying our problem-solving skills to the way we work.

In the tool business I discovered a surprising fact: most embedded developers work more or less in isolation. They may be loners designing all of the products for a company, or members of a company's design team. The loner and the team are removed from others in the industry and so develop their own generally dysfunctional habits that go forever uncorrected. Few developers or teams ever participate in industry-wide events or communicate with the rest of the industry. We, who invented the communications age, seem to be incapable of using it!

One effect of this isolation is a hardening of the development arteries: we are unable to benefit from others' experiences, so work ever harder without getting smarter. Another is a feeling of frustration, of thinking "what is wrong with us; why are our projects so much more a problem than anyone else's?" In fact, most embedded developers are in the same boat.

This book comes from seeing how we all share the same problems while not finding solutions. Never forget that engineering is about solving problems … including the ones that plague the way we engineer!

Engineering is the process of making choices; make sure yours reflect simplicity, common sense, and a structure with growth, elegance, and flexibility, with debugging opportunities built in.

How many of us designing microprocessor-based products can explain our jobs at a cocktail party? To the average consumer the word "computer" conjures up images of mainframes or PCs. He blithely disregards or is perhaps unaware of the tremendous number of little processors that are such an important part of everyone's daily lives. He wakes up to the sound of a computer-generated alarm, eats a breakfast prepared with a digital microwave, and drives to work in a car with a virtual dashboard. Perhaps a bit fearful of new technology, he'll tell anyone who cares to listen that a pencil is just fine for writing, thank you; computers are just too complicated.

So many products that we take for granted simply couldn't exist without an embedded computer! Thousands owe their lives to sophisticated biomedical instruments like CAT scanners, implanted heart monitors, and sonograms. Ships as well as pleasure vessels navigate by GPS that torturously iterate non-linear position equations. State-of-the-art DSP chips in traffic radar detectors attempt to thwart the police, playing a high tech cat and mouse game with the computer in the authority's radar gun. Compact disc players give perfect sound reproduction using high integration devices that provide error correction and accurate track seeking.

It seems somehow appropriate that, like molecules and bacteria, we disregard computers in our day-to-day lives. The microprocessor has become part of the underlying fabric of late 20th century civilization. Our lives are being subtly changed by the incessant information processing that surrounds us.

Microprocessors offer far more than minor conveniences like TV remote control. One ultimately crucial application is reduced consumption of limited natural resources. Smart furnaces use solar input and varying user demands to efficiently maintain comfortable temperatures. Think of it—a fleck of silicon saving mountains of coal! Inexpensive programmable sprinklers make off-peak water use convenient, reducing consumption by turning the faucet off even when forgetful humans are occupied elsewhere. Most industrial processes rely on some sort of computer control to optimize energy use and to meet EPA discharge restrictions. Electric motors are estimated to use some 50% of all electricity produced—cheap motor controllers that net even tiny efficiency improvements can yield huge power savings. Short of whole new technologies that don't yet exist,

smart, computationally intense use of resources may offer us the biggest near-term improvements in the environment.

What is this technology that so changed the nature of the electronics industry? Programming the VCR or starting the microwave you invoke the assistance of an embedded microprocessor—a computer built right into the product.

Embedded microprocessor applications all share one common trait: the end product is not a computer. The user may not realize that a computer is included; certainly no 3-year-old knows or cares that a processor drives Speak and Spell. The teenager watching MTV is unaware that embedded computers control the cable box and the television. Mrs. Jones, gossiping long distance, probably made the call with the help of an embedded controller in her phone. Even the "power" computer user may not know that the PC is really a collection of processors; the keyboard, mouse, and printer each include at least one embedded microprocessor.

For the purpose of this book, an embedded system is any application where a dedicated computer is built right into the system. While this definition can apply even to major weapon systems based on embedded blade servers, here I address the perhaps less glamorous but certainly much more common applications using 8-, 16-, and 32-bit processors.

Although the microprocessor was not explicitly invented to fulfill a demand for cheap general purpose computing, in hindsight it is apparent that an insatiable demand for some amount of computational power sparked its development. In 1970 the minicomputer was being harnessed in thousands of applications that needed a digital controller, but its high cost restricted it to large industrial processes and laboratories. The microprocessor almost immediately reduced computer costs by a factor of a thousand. Some designers saw an opportunity to replace complex logic with a cheap 8051 or Z80. Others realized that their products could perform more complex functions and offer more features with the addition of these silicon marvels.

This, then, is the embedded systems industry. In two decades we've seen the microprocessor proliferate into virtually every piece of electronic equipment. The demand for new applications is accelerating.

The goal of the book is to offer approaches to dealing with common embedded programming problems. While all college computer science courses teach traditional

programming, few deal with the peculiar problems of embedded systems. As always, schools simply cannot keep up with the pace of technology. Again and again we see new programmers totally baffled by the interdisciplinary nature of this business. For there is often no clear distinction between the hardware and software; the software in many cases is an extension of the hardware; hardware components are replaced by software-controlled algorithms. Many embedded systems are real time—the software must respond to an external event in some number of microseconds and no more. We'll address many design issues that are traditionally considered to be the exclusive domain of hardware gurus. The software and hardware are so intertwined that the performance of both is crucial to a useful system; sometimes programming decisions profoundly influence hardware selection.

Historically, embedded systems were programmed by hardware designers, since only they understood the detailed bits and bytes of their latest creation. With the paradigm of the microprocessor as a controller, it was natural for the digital engineer to design as well as code a simple sequencer. Unfortunately, most hardware people were not trained in design methodologies, data structures, and structured programming. The result: many early microprocessor-based products were built on thousands of lines of devilishly complicated spaghetti code. The systems were un-maintainable, sometimes driving companies out of business.

The increasing complexity of embedded systems implies that we'll see a corresponding increase in specialization of function in the design team. Perhaps a new class of firmware engineers will fill the place between hardware designers and traditional programmers. Regardless, programmers developing embedded code will always have to have detailed knowledge of both software and hardware aspects of the system.