TURING 图灵原版电子与电气工程系列

ELSEVIER 爱思唯尔

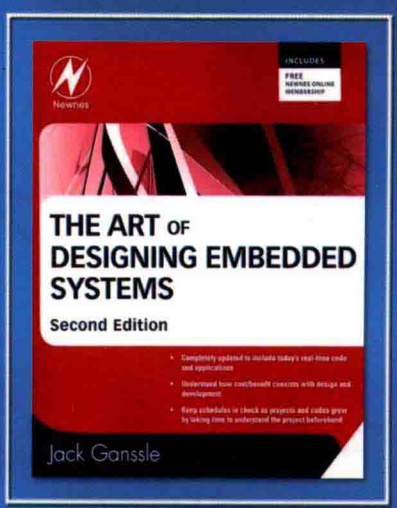
The Art of Designing Embedded Systems (Second Edition)

嵌入式系统设计的艺术

（英文版·第2版）

[美] Jack Ganssle 著

世界级权威力作
嵌入式开发圣经



人民邮电出版社
POSTS & TELECOM PRESS

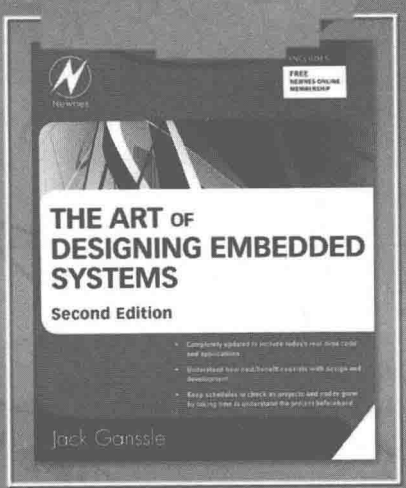
TURING 图灵原版电子与电气工程系列

The Art of Designing Embedded Systems (Second Edition)

嵌入式系统设计的艺术

（英文版·第2版）

[美] Jack Ganssle 著



人民邮电出版社
北京

图书在版编目（CIP）数据

嵌入式系统设计的艺术：第2版：英文 /（美）甘瑟尔（Ganssle, J.）著. —北京：人民邮电出版社，2009.3
（图灵原版电子与电气工程系列）
书名原文:The Art of Designing Embedded Systems, Second Edition
ISBN 978-7-115-19521-0

Ⅰ. 嵌… Ⅱ. 甘… Ⅲ. 微型计算机－系统设计－英文
Ⅳ. TP360.21

内容提要

本书针对嵌入式系统开发中的一些本质问题提出了大量深刻见解，内容涵盖嵌入式系统的开发过程、代码编写、实时性问题等方面。附录部分还给出了固件标准、设计样例及设计指南等方面的丰富内容。

本书是从事嵌入式系统设计和开发的工程技术人员的必备参考书，也可供高等学校相关专业本科生和研究生参考。

图灵原版电子与电气工程系列

嵌入式系统设计的艺术（英文版·第2版）

◆ 著　[美] Jack Ganssle
责任编辑　舒　立

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：19
字数：304千字　2009年3月第1版
印数：1－2 000册　2009年3月北京第1次印刷

著作权合同登记号　图字：01-2008-5053号

ISBN 978-7-115-19521-0/TN

定价：49.00元

读者服务热线：(010)88593802　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

Elsevier (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2009
2009年初版

To my family, Marybeth, Nat, Graham, and Kristy.

Contents

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.

CHAPTER 2

The Project

2.1 Partitioning

In 1946 programmers created software for the ENIAC machine by rewiring plug-boards. Two years later the University of Manchester's Small-Scale Experimental Machine, nicknamed Baby, implemented von Neumann's stored program concept, for the first time supporting a machine language. Assembly language soon became available and flourished. But in 1957 Fortran, the first high level language, debuted and forever changed the nature of programming.

In 1964, Dartmouth BASIC introduced millions of non-techies to the wonders of computing while forever poisoning their programming skills. Three years later, almost as a counterpoint, OOP (object-oriented programming) appeared in the guise of Simula 67. C, still the standard for embedded development, and C++ appeared in 1969 and 1985, respectively.

By the 1990s, a revolt against big, up-front design led to a flood of new "agile" programming methodologies including eXtreme Programming, SCRUM, Test-Driven Development, Feature-Driven Development, the Rational Unified Process, and dozens more.

In the 50 years since programming first appeared, software engineering has morphed to something that would be utterly alien to the software developer of 1946. That half-century has taught us a few pivotal lessons about building programs. Pundits might argue that the most important might be the elimination of "gotos," the use of objects, or building from patterns.

They'd be wrong. The fundamental insight of software engineering is to keep things small. Break big problems into little ones.

For instance, we understand beyond a shadow of a doubt the need to minimize function sizes. No one is smart enough to understand, debug, and maintain a 1000-line routine, at least not in an efficient manner. Consequently, we've learned to limit our functions to around 50 lines of code. Reams of data prove that restricting functions to a page of code or less reduces bug rates and increases productivity.

But why is partitioning so important?

A person's short-term memory is rather like cache—a tiny cache—actually, one that can hold only 5–9 things before new data flushes the old. Big functions blow the programmer's mental cache. The programmer can no longer totally understand the code; errors proliferate.

2.1.1 The Productivity Crash

But there's a more insidious problem. Developers working on large systems and subsystems are much less productive than those building tiny applications.

Consider the data in Table 2.1, gathered from a survey [1] of IBM software projects. Programmer productivity plummets by an order of magnitude as projects grow in scope! That is, of course, exactly the opposite of what the boss is demanding, usually quite loudly.

The growth in communications channels between team members sinks productivity on large projects. A small application, one built entirely by a single developer, requires zero comm channels—it's all in the solo guru's head. Two engineers need only one channel.

Table 2.1: IBM productivity in lines of code per programmer per month

Project size man/months	Productivity lines of code/month
1	439
10	220
100	110
1000	55

The number of communications channels between n engineers is:

$$\frac{n(n-1)}{2}$$

This means that communications among team members grow at a rate similar to the square of the number of developers. Add more people and pretty soon their days are completely consumed with email, reports, meetings, and memos (Figure 2.1).

Fred Brooks in his seminal (and hugely entertaining) work [2] "The Mythical Man-Month" described how the IBM 360/OS project grew from a projected staffing level of 150 people to over 1000 developers, all furiously generating memos, reports, and the occasional bit of code. In 1975, he formulated Brooks' Law, which states: adding people to a late project makes it later. Death-march programming projects continue to confirm this maxim, yet management still tosses workers onto troubled systems in the mistaken belief that an N man-month project can be completed in 4 weeks by N programmers.

Is it any wonder some 80% of embedded systems are delivered late?

Table 2.2 illustrates Joel Aron's [2] findings at IBM. Programmer productivity plummets on big systems, mostly because of interactions required between team members.

The holy grail of computer science is to understand and ultimately optimize software productivity. Tom DeMarco and Timothy Lister [3] spent a decade on this noble quest, running a yearly "coding war" among some 600 organizations. Two independent teams

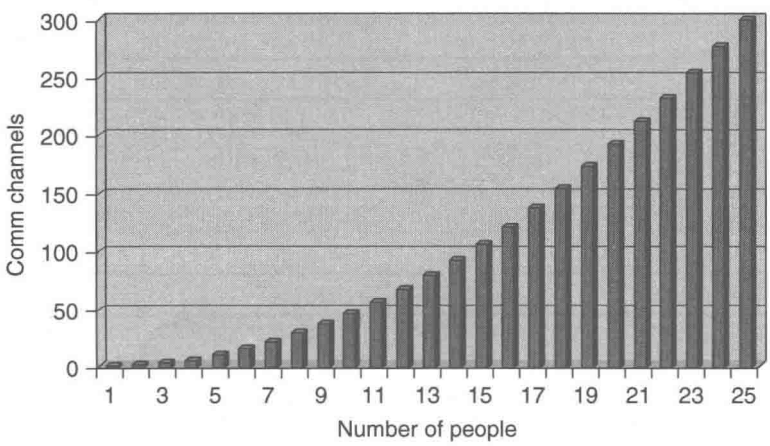


Figure 2.1: The growth in comm channels with people

at each company wrote programs to solve a problem posited by the researchers. The resulting scatter plot looked like a random cloud; there were no obvious correlations between productivity (or even bug rates) and any of the usual suspects: experience, programming language used, salary, etc. Oddly, at any individual outfit the two teams scored about the same, suggesting some institutional factor that contributed to highly—and poorly—performing developers.

A lot of statistical head-scratching went unrewarded till the researchers reorganized the data as shown in Table 2.3.

The results? The top 25% were 260% more productive than the bottom quartile!

The lesson here is that interruptions kill software productivity, mirroring Joel Aron's results. Other work has shown it takes the typical developer 15 minutes to get into a state of "flow," where furiously typing fingers create a wide-bandwidth link between the programmer's brain and the computer. Disturb that concentration via an interruption and the link fails. It takes 15 minutes to rebuild that link but, on average, developers are interrupted every 11 minutes [4].

Interrupts are the scourge of big projects.

Table 2.2: Productivity plummets as interactions increase

Interactions	Productivity
Very few interactions	10,000 LOC/man-year
Some interactions	5000 LOC/man-year
Many interactions	1500 LOC/man-year

Table 2.3: Coding war results

	1st Quartile	4th Quartile
Dedicated workspace	78 sq ft	46 sq ft
Is it quiet?	57% yes	29% yes
Is it private?	62% yes	19% yes
Can you turn off phone?	52% yes	10% yes
Can you divert your calls?	76% yes	19% yes
Frequent interruptions?	38% yes	76% yes