Newnes

INCLUDES

FREE
NEWNES ONLINE
MEMBERSHIP

# THE ART OF DESIGNING EMBEDDED SYSTEMS

Second Edition

- Completely updated to include today's real-time code and applications
- Understand how cost/benefit coexists with design and development
- Keep schedules in check as projects and codes grow by taking time to understand the project beforehand

Jack Ganssle

# *The Art of Designing Embedded Systems*

# *The Art of Designing Embedded Systems*

***Second Edition***

Jack Ganssle

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes

*Cover image courtesy of iStockphoto*

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

**Library of Congress Cataloging-in-Publication Data**

Ganssle, Jack G.
The art of designing embedded systems / Jack Ganssle. — 2nd ed.
p. cm.
ISBN 978-0-7506-8644-0
1. Embedded computer systems—Design and construction. I. Title.
TK7895.E42G36 2008
004.16--dc22
2008012453

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library.

08 09 10 11 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

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

# Contents

# *Acknowledgments*

Over almost 20 years writing columns about embedded systems I've corresponded with thousands of engineers all over the world. Some send jokes, others requests for help, some pass along ideas and stories. Thanks to all of you for your thoughts and insights, for challenging my writing and making me refine my thinking.

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

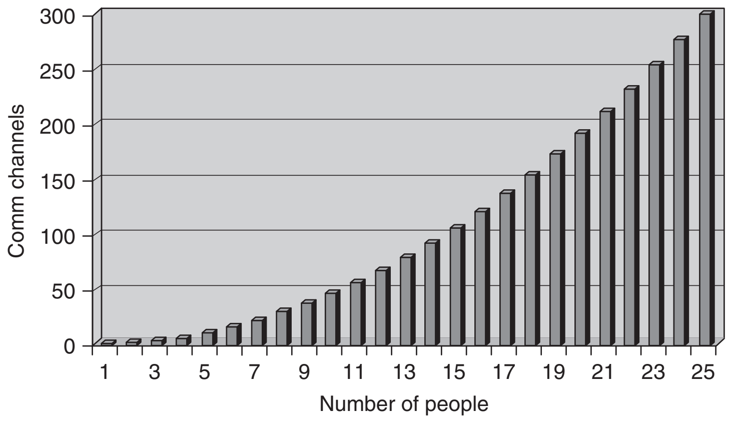


**Figure 2.1: The growth in comm channels with people**

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 |

A maxim of software engineering is that functions should be strongly cohesive but only weakly coupled. Development teams invariably act in the opposite manner. The large number of communications channels makes the entire group highly coupled. Project knowledge is distributed in many brains. Joe needs information about an API call. Mary is stuck finding a bug in the interface with Bob's driver. Each jumps up and disturbs a concentrating team member, destroying that person's productivity.

### 2.1.2 COCOMO

Barry Boehm, the god of software estimation, derived what he calls the Constructive Cost Model, or COCOMO [5], for determining the cost of software projects. Though far from perfect, COCOMO is predictive, quantitative, and probably the most well-established model extant.

Boehm defines three different development modes: organic, semidetached, and embedded—where "embedded" means a software project built under tight constraints in a complex of hardware, software, and sometimes regulations. Though he wasn't thinking of firmware as we know it, this is a pretty good description of a typical embedded system.

Under COCOMO the number of person-months (*PM*) required to deliver a system developed in the "embedded" mode is:

$$PM = \prod_{i=1,15} F_i \times 2.8 \times KLOC^{1.20}$$

where *KLOC* is the number of lines of source code in thousands and $F_i$ are 15 different cost drivers.

Cost drivers include factors such as required reliability, product complexity, real-time constraints, and more. Each cost driver is assigned a weight that varies from a little under 1.0 to a bit above. It's reasonable for a first approximation to assume these cost driver figures all null to about 1.0 for typical projects.

This equation's intriguing exponent dooms big projects. *Schedules grow faster than the code size*. Double the project's size and the schedule will grow by more, sometimes far more, than a factor of two.

Despite his unfortunate eponymous use of the word to describe a development mode, Boehm never studied real-time embedded systems as we know them today so there's some doubt about the validity of his exponent of 1.20. Boehm used American Airlines' Saber reservation system as a prime example of a real-time application. Users wanted an answer "pretty fast" after hitting the enter key. In the real embedded world where missing a deadline by even a microsecond results in *60 Minutes* appearing on the doorstep and multiple indictments, managing time constraints burns through the schedule at a prodigious rate.

Fred Brooks believes the exponent should be closer to 1.5 for real-time systems. Anecdotal evidence from some dysfunctional organizations gives a value closer to 2. Double the code and multiply man-months by 4. Interestingly, that's close to the number of communications channels between engineers, as described above.

Let's pick an intermediate and conservative value, 1.35, which sits squarely between Boehm's and Brooks' estimate and is less than most anecdotal evidence suggests. Figure 2.2 shows how productivity collapses as the size of the program grows.

(Most developers rebel at this point. "I can crank 1000 lines of code over the weekend!" And no doubt that's true. However, these numbers reflect costs over the entire development cycle, from inception to shipping. Maybe you are a superprogrammer and consistently code much faster than, say, 200-LOC/month. Even so, the shape of the curve, the exponential loss of productivity, is undeniable.)

Computer science professors show their classes graphs like the one in Figure 2.2 to terrify their students. The numbers are indeed scary. A million-LOC project sentences us to the 32-LOC/month chain gang. We can whip out a small system over the weekend but big ones take years.

Or do they?

### *2.1.3 Partitioning Magic*

Figure 2.2 shows software hell but it also illuminates an opportunity. What if we could somehow cheat the curve, and work at, perhaps, the productivity level for 20-KLOC programs even on big systems? Figure 2.3 reveals the happy result.

The upper curve in Figure 2.3 is the COCOMO schedule; the lower one assumes we're building systems at the 20-KLOC/program productivity level. The schedule, and hence costs, grow linearly with increasing program size.

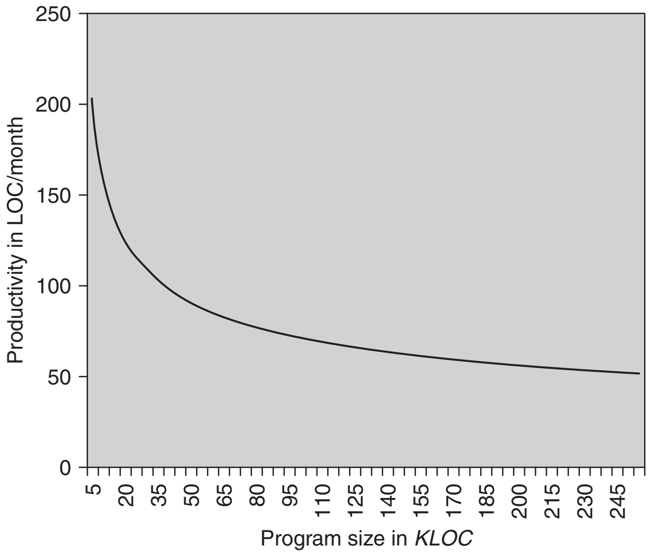


**Figure 2.2: The productivity collapse**

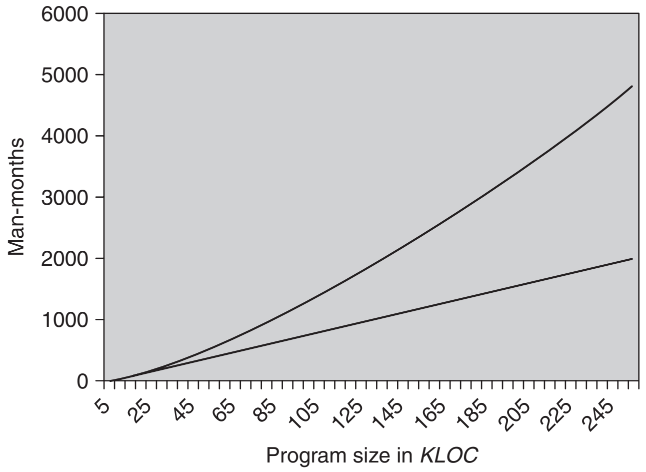


**Figure 2.3: Cheating the schedule's exponential growth**

But how can we operate at these constant levels of productivity when programs exceed 20 KLOC? The answer: by partitioning! By understanding the implications of Brook's Law and DeMarco and Lister's study. The data is stark; if we don't compartmentalize the project, divide it into small chunks that can be created by tiny teams working more or less in isolation. Schedules will balloon exponentially.

Professionals look for ways to maximize their effectiveness. As professional software engineers *we have a responsibility to find new partitioning schemes.* Currently 70–80% of a product's development cost is consumed by the creation of code and that ratio is only likely to increase because firmware size doubles about every 10 months. Software will eventually strangle innovation without clever partitioning.

Academics drone endlessly about top-down decomposition (TDD), a tool long used for partitioning programs. Split your huge program into many independent modules, divide these modules into functions, and only then start cranking code.

TDD is the biggest scam ever perpetrated on the software community.

Though TDD is an important strategy that allows wise developers to divide code into manageable pieces, it's not the *only* tool we have available for partitioning programs. TDD is merely one arrow in the quiver, never used to the exclusion of all else. We in the embedded systems industry have some tricks unavailable to IT programmers.

Firmware is the most expensive thing in the universe. In his book *Augustine's Laws* [6], Lockheed Martin's Chairman Norman Augustine relates a tale of trouble for manufacturers of fighter aircraft. By the late 1970s it was no longer possible to add things to these planes because adding new functionality meant increasing the aircraft's weight, which would impair performance. However, the aircraft vendors needed to add something to boost their profits. They searched for something, anything, that weighed nothing but cost a lot. And they found it—firmware! It was the answer to their dreams.

The F-4 was the hot fighter of the 1960s. In 2007 dollars these airplanes cost about $20 million each and had essentially no firmware. Today's F-22, just coming into production, runs a cool $333 million per copy. Half of that price tag is firmware.

The DoD contractors succeeded beyond their wildest dreams.

One way to keep productivity levels high—at, say, the 20-KLOC/program number—is to break the program up into a number of discrete entities, each no more than 20 KLOC long. Encapsulate each piece of the program into its own processor.

That's right: add CPUs merely for the sake of accelerating the schedule and reducing engineering costs.

Sadly, most 21st century embedded systems look an awful lot like mainframe computers of yore. A single CPU manages a disparate array of sensors, switches, communications links, PWMs, and more. Dozens of tasks handle many sorts of mostly unrelated activities. A hundred thousand lines of code all linked into a single executable enslaves dozens of programmers all making changes throughout a byzantine structure no one completely comprehends. Of course development slows to a crawl.

Transistors are cheap. Developers expensive.

Break the system into small parts, allocate one partition per CPU, and then use a small team to build each subsystem. Minimize interactions and communications between components and between the engineers.

Suppose the monolithic, single-CPU version of the product requires 100 K lines of code. The COCOMO calculation gives a 1403–man-month development schedule.

Segment the same project into four processors, assuming one has 50 KLOC and the others 20 KLOC each. Communications overhead requires a bit more code so we've added 10% to the 100-KLOC base figure.

The schedule collapses to 909 man-months, or 65% of that required by the monolithic version.

Maybe the problem is quite orthogonal and divides neatly into many small chunks, none being particularly large. Five processors running 22 KLOC each will take 1030 man-months, or 73% of the original, not-so-clever design.

Transistors *are* cheap—so why not get crazy and toss in lots of processors? One processor runs 20 KLOC and the other 9 each run 10-KLOC programs. The resulting 724–man-month schedule is just half of the single-CPU approach. The product reaches consumers' hands twice as fast and development costs tumble. You're promoted and get one of those hot foreign company cars plus a slew of appreciating stock options. Being an engineer was never so good.

## *2.1.4 Save Big Bucks by Reducing NRE*

Hardware designers will shriek when you propose adding processors just to accelerate the software schedule. Though they know transistors have little or no cost, the EE's zeitgeist is to always minimize the bill of materials. Yet since the dawn of the microprocessor age, it has been routine to add parts just to simplify the code. No one today would consider building a software UART, though it's quite easy to do and wasn't terribly uncommon decades ago. Implement asynchronous serial I/O in code and the structure of the entire program revolves around the software UART's peculiar timing requirements. Consequently, the code becomes a nightmare. So today we add a hardware UART. Always. The same can be said about timers, pulse-width modulators, and more. The hardware and software interact as a synergistic whole orchestrated by smart designers who optimize both product and engineering costs.

Sum the hardware component prices, add labor and overhead, and you still haven't properly accounted for the product's cost. NRE, non-recurring engineering, is just as important as the price of the PCB. Detroit understands this. It can cost more than $2 billion to design and build the tooling for a new car. Sell one million units and the consumer must pay $2000 above the component costs to amortize those NRE bills.

Similarly, when we in the embedded world save NRE dollars by delivering products faster, we reduce the system's recurring cost. Everyone wins.

Sure, there's a cost to adding processors, especially when doing so means populating more chips on the PCB. But transistors are particularly cheap inside of an ASIC. A full 32-bit CPU can cost as little as 20–30K gates. Interestingly, customers of IP vendor Tensilica average six 32-bitters per ASIC, with at least one using more than 180 processors! So if time to market is *really* important to your company (and when isn't it?), if the code naturally partitions well, and if CPUs are truly cheap, what happens when you break all the rules and add *lots* of processors? Using the COCOMO numbers, a one-million-LOC program divided over 100 CPUs can be completed five times faster than using the traditional monolithic approach, at about 1/5 the cost.

Adding processors increases system performance, not surprisingly simultaneously reducing development time. A rule of thumb states that a system loaded to 90% processor capability doubles development time [7] (over one loaded at 70% or less). At 95% processor loading, expect the project schedule to *triple*. When there's only a handful of bytes left over, adding even the most trivial new feature can take weeks as the developers rewrite massive sections of code to free up a bit of ROM. When CPU cycles are in short supply, an analogous situation occurs.

Consider these factors:

- 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 20 cents in quantity). Controllers have everything you need on one chip—RAM, ROM, I/O, etc.
- An additional processor need not eat much board space (see Figure 2.4).
- Think OTP—One Time Programmable or Flash 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.

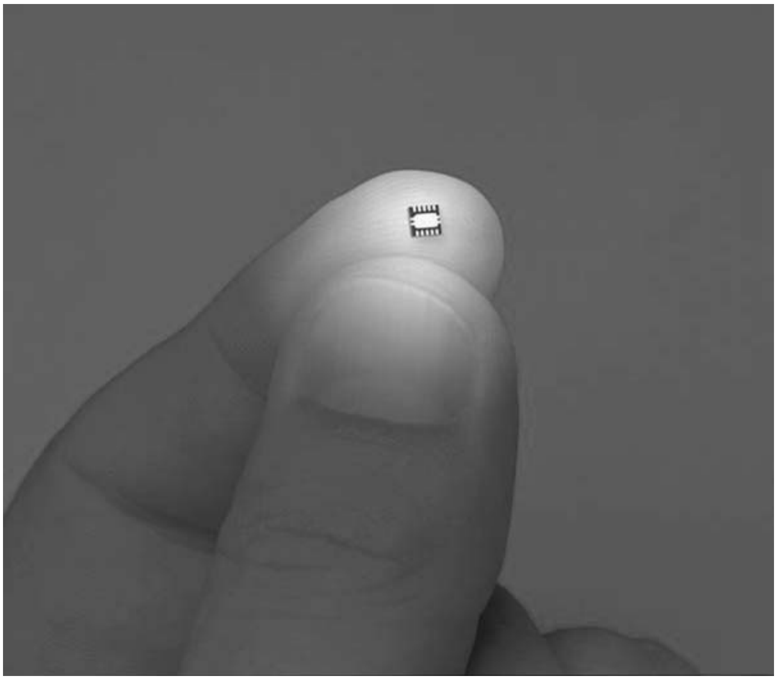


**Figure 2.4: A complete MCU uses, to a first approximation, exactly zero PCB space (Photo courtesy of Silabs)**

- 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 acts as a natural barrier to complex communications and interaction between processors. Don't defeat this by 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/s there's enough speed for most applications. Even better: you can steal the code from Microchip's and National Semiconductor's websites.

Non-Recurring Engineering (NRE) costs are the bane of most technology managers' lives. NRE is that cost associated with developing a product. Its converse is the Cost of Goods Sold (COGS), a.k.a. Recurring Costs.

NRE costs are amortized over the life of a product in fact or in reality. Mature companies carefully compute the amount of engineering a the product—a car maker, for instance, might spend a billion bucks engineering a new model with a lifespan of a million units sold; in this case the cost of the car goes up by $1000 to pay for the NRE. Smaller technology companies often act like cowboys and figure that NRE is just the cost of doing business; if we are profitable then the product's price somehow (!) reflects all engineering expenses.

Increasing NRE costs drives up the product's price (most likely making it less competitive and thus reducing profits), or directly reduces profits.

Making an NRE versus COGS decision requires a delicate balancing act that deeply mirrors the nature of your company's product pricing. A $1 electronic greeting card cannot stand any extra components; minimize COGS uber alles. In an automobile the quantities are so large engineers agonize over saving a foot of wire. The converse is a one-off or short production run device. The slightest development hiccup costs tens of thousands—easily—which will have to be amortized over a very small number of units.

Sometimes it's easy to figure the tradeoff between NRE and COGS. You should also consider the extra complication of opportunity costs—"If I do this, then what is the cost of not doing that?" As a young engineer I realized that we could save about $5000 a year by changing from EPROMs to masked ROMs. I prepared a careful analysis and presented it to my boss, who instantly turned it down as making the change would shut down my other engineering activities for some time. In this case we had a tremendous backlog of projects, any of which could yield more revenue than the measly $5K saved. In effect, my boss's message was "you are more valuable than what we pay you." (That's what drives entrepreneurs into business—the hope they can get the extra money into their own pockets!)

## *2.1.5 The Superprogrammer Effect*

Developers come in all sorts of flavors, from somewhat competent plodders to miracle-workers who effortlessly create beautiful code in minutes. Management's challenge is to recognize the superprogrammers and use them efficiently. Few bosses do; the best programmers get lumped with relative dullards to the detriment of the entire team.

Table 2.4 summarizes a study done by Capers Jones [8]. The best developers, the superprogrammers, excel on small jobs. Toss them onto a huge project and they'll slog along at about the same performance as the very worst team members.

Big projects wear down the superstars. Their glazed eyes reflect the meeting and paperwork burden; their creativity is thwarted by endless discussions and memos.

The moral is clear and critically important: wise managers put their very best people on the small sections partitioned off of the huge project. Divide your system over many CPUs and let the superprogrammers attack the smallest chunks.

**Table 2.4: The superprogrammer effect**

| Size in KLOC | Best programmer (months/KLOC) | Worst programmer (months/KLOC) |
|---|---|---|
| 1 | 1 | 6 |
| 8 | 2.5 | 7 |
| 64 | 6.5 | 11 |
| 512 | 17.5 | 21 |
| 2048 | 30 | 32 |

Though most developers view themselves as superprogrammers, competency follows a bell curve. Ignore self-assessments. In *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments*, Justin Kruger and David Dunning [9] showed that though the top half of performers were pretty accurate in evaluating their own performance, the bottom half are wildly optimistic when rating themselves.

### 2.1.5.1 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 customers we'll be unable to satisfy them. 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.

The example in Figure 2.5 shows the feature matrix for a printer. Notice that the first few items are not really features; they're basic, low level functions required just to get the thing to start up, as indicated by the "Importance" factor of "Required."

Beyond these, though, are things used to differentiate the product from competitive offerings. Downloadable fonts might be important, but do not affect the unit's ability to just put ink on paper. Image rotation, listed as the least important feature, 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 in trouble because the due date looms with virtually nothing complete. Perhaps hundreds

| *Feature* | *Importance* | *Priority* | *Complexity* |
|---|---|---|---|
| | | | |
| Shell | Required | | 500 |
| RTOS | Required | | (purchased) |
| Keyboard handler | Required | | 300 |
| LED driver | Required | | 500 |
| Comm. with host | Required | | 4,000 |
| Paper handling | Required | | 2,000 |
| Print engine | Required | | 10,000 |
| Downloadable fonts | Important | 1 | 1,000 |
| Main 100 local fonts | Important | 2 | 6,000 |
| Unusual local fonts | Less important | 3 | 10,000 |
| Image rotation | Less important | 4 | 3,000 |

**Figure 2.5: A printer's feature matrix**

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 status.

(The complexity rating in Figure 2.5 is 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. 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.

Eighty percent 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 (Figure 2.6).

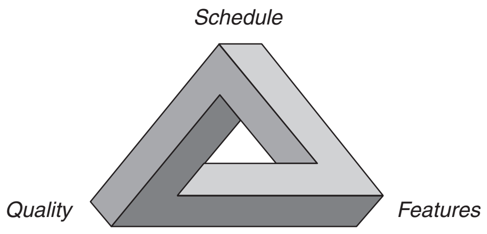


**Figure 2.6: The twisted tradeoff**

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's 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*. That leaves only one leg of the Twisted Tradeoff left to fiddle. Cut a few of the less important features to 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. An older 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.

### *2.1.6 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.

As soon as practicable build your system's skeleton and switch it on. Build the startup code. Get chip selects working. Create stub tasks or calling routines. Glue in purchased packages and prove to yourself that they work as advertised and as required. Deal with the vendor, if trouble surfaces, *now* rather than in a last minute debug panic when they've unexpectedly gone on holiday for a week.

In a matter of days or a week or two you'll have a skeleton assembled, a skeleton that actually operates in some very limited manner. Perhaps it runs a null loop. Using your development tools test this small-scale chunk of the application.

Start adding the lowest level code, testing as you go. Soon your system will have all of the device drivers in place (tested), ISRs (tested), the startup code (tested), and the major support items like comm packages and the RTOS (again tested). Integration of your own applications code can then proceed in a reasonably orderly manner, plopping modules into a known good code framework, facilitating testing at each step.

The point is to immediately build a framework that operates, and then drop features in one at a time, testing each as it becomes available. You're testing the entire system, such as it is, and expanding those tests as more of it comes together. Test and integration are no longer individual milestones; they are part of the very fabric of development.

Success requires a determination to constantly test. Every day, or at least every week, build the entire system (using all of the parts then available) and insure that things work correctly. *Test constantly*. Fix bugs immediately.

The daily or weekly testing is the project's heartbeat. It insures that the system really can be built and linked. It gives a constant view of the system's code quality, and encourages early feature feedback (a mixed blessing, admittedly, but our goal is to satisfy the customer, even at the cost of accepting slips due to reengineering poor feature implementation).

At the risk of sounding like a new-age romantic, someone working in aroma therapy rather than pushing bits around, we've got to learn to deal with human nature in the design process. Most managers would trade their firstborn for an army of Vulcan programmers, but until the Vulcan economy collapses ("emotionless programmer, will work for peanuts and logical discourse") we'll have to find ways to efficiently use humans, with all of their limitations.

We people need a continuous feeling of accomplishment to feel effective and to be effective. Engineering is all about *making things work*; it's important to recognize this and create a development strategy that satisfies this need. Lots of little progress points, where we see our system doing something, are tons more satisfying than coding for a year before hitting the ON switch.

A hundred thousand lines of carefully written and documented code are nothing more than worthless bits until they are tested. We hear "It's done" all the time in this field, where "done" might mean "vaguely understood" or "coded." To me "done" has one meaning only: "tested."

Incremental development and testing, especially of the high risk areas like hardware and communications, reduce risks tremendously. Even when we're not honest with each other ("Sure, I can crank this puppy out in a week, no sweat"), deep down we usually recognize risk well enough to feel scared. Mastering the complexities up front removes the fear and helps us work confidently and efficiently.

#### 2.1.6.1 Conquer the impossible

Firmware people are too often treated as scum, 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 firmware people can't possibly make up the slipped schedule.

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.

- 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 a 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 80% 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. This step also helps prove the hardware design early, a benefit to everyone.
- Use a simulator. There are a lot of commercial products available today from companies like Keil and Virtutech that actually work. You develop a substantial portion of the project with no extant hardware.
- Consider using one of the agile test harnesses, such as Catsrunner or CPPunit.

### 2.1.7 What About SMP?

For many years processors and memory evolved more or less in lockstep. Early CPUs like the Z80 required a number of machine cycles to execute even a NOP instruction. At the few MHz clock rates then common, processor speeds nicely matched EPROM and SRAM cycle times.

But for a time memory speeds increased faster than CPU clock rates. The 8088/6 had a prefetcher to better balance fast memory to a slow processor. A very small (4–6 bytes) FIFO isolated the core from a bus interface unit (BIU). The BIU was free to prefetch the most likely needed next instruction if the core was busy doing something that didn't need bus activity. The BIU thus helped maintain a reasonable match between CPU and memory speeds.

Even by the late 1980s processors were pretty well matched to memory. The 386, which (with the exception of floating point instructions) has a programmer's model very much like Intel's latest high-end offerings, came out at 16 MHz. The three-cycle NOP instruction thus consumed 188 nsec, which partnered well with most zero wait state memory devices.

But clock rates continued to increase while memory speeds started to stagnate. The 386 went to 40 MHz, and the 486 to over 100. Some of the philosophies of the reduced instruction set (RISC) movement, particularly single-clock instruction execution, were adopted by CISC vendors, further exacerbating the mismatch.

Vendors turned to Moore's Law as it became easier to add lots of transistors to processors to tame the memory bottleneck. Pipelines sucked more instructions on-chip, and extra logic executed parts of many instructions in parallel.

A single-clock 100-MHz processor consumes a word from memory every 10 nsec, but even today that's pretty speedy for RAM and impossible for Flash. So on-chip cache appeared, again exploiting cheap integrated transistors. That, plus floating point and a few other nifty features, meant the 486's transistor budget was over four times as large as the 386's.

Pentium-class processors took speeds to unparalleled extremes, before long hitting 2 and 3 GHz. 0.33-nsec memory devices are impractical for a variety of reasons, not the least of which is the intractable problem of propagating those signals between chip packages. Few users would be content with a 3-GHz processor stalled by issuing 50 wait states for each memory read or write, so cache sizes increased more.

But even on-chip, zero wait state memory is expensive. Caches multiplied, with a small, fast L1 backed up by a slower L2, and in some cases even an L3. Yet more transistors implemented immensely complicated speculative branching algorithms, cache snooping, and more, all in the interest of managing the cache and reducing inherently slow bus traffic.

And that's the situation today. Memory is much slower than processors, and has been an essential bottleneck for 15 years. Recently CPU speeds have stalled as well, limited now by power dissipation problems. As transistors switch, small inefficiencies convert a tiny bit of Vcc to heat. And even an idle transistor leaks microscopic amounts of current. Small losses multiplied by hundreds of millions of devices mean very hot parts.

Ironically, vast numbers of the transistors on a modern processor do nothing most of the time. No more than a single line of the cache is active at any time; most of the logic to handle hundreds of different instructions stands idle till infrequently needed; and page translation units that manage gigabytes handle a single word at a time.

But those idle transistors do convert the power supply to waste heat. The "transistors are free" mantra is now stymied by power concerns. So limited memory speeds helped spawn hugely complex CPUs, but the resultant heat has curbed clock rates, formerly the biggest factor that gave us faster computers every year.

In the supercomputing world similar dynamics were at work. GaAs logic and other exotic components drove clock rates high, and liquid cooling kept machines from burning up. But long ago researchers recognized the futility of making much additional progress by spinning the clock rate wheel ever higher, and started building vastly parallel machines. Most today employ thousands of identical processing nodes, often based on processors used in standard desktop computers. Amazing performance comes from massively parallelizing both the problems and the hardware.

To continue performance gains desktop CPU vendors co-opted the supercomputer model and today offer a number of astonishing multicore devices, which are just two or more standard processors assembled on a single die. A typical configuration has two CPUs, each with its own L1 cache. Both share a single L2, which connects to the outside world via a single bus. Embedded versions of these parts are available as well, and share much with their desktop cousins.

Symmetric multiprocessing has been defined in a number of different ways. I chose to call a design using multiple identical processors which share a memory bus an SMP system. Thus, multicore offerings from Intel, AMD, and some others are SMP devices.

SMP will yield performance improvements only (at best) insofar as a problem can be parallelized. Santa's work cannot be parallelized (unless he gives each elf a sleigh), but delivering mail order products keeps a fleet of UPS trucks busy and efficient.

Amdahl's Law gives a sense of the benefit accrued from using multiple processors. In one form it gives the maximum speedup as:

$$\frac{1}{f + (1 - f)/n}$$

where $f$ is the part of the computation that can't be parallelized, and $n$ is the number of processors. With an infinite number of cores, assuming no other mitigating circumstances, Figure 2.7 shows (on the vertical axis) the possible speedup versus (on the horizontal axis) the percentage of the problem that cannot be parallelized.

The law is hardly engraved in stone as there are classes of problems called "embarrassingly parallel" where huge numbers of calculations can take place simultaneously. Supercomputers have long found their niche in this domain, which includes problems like predicting the weather, nuclear simulations, and the like.

The crucial question becomes: how much can your embedded application benefit from parallelization? Many problems have at least some amount of work that can take place simultaneously. But most problems have substantial interactions between components that must take place in a sequence. It's hard at best to decide at the outset, when one is selecting the processor, how much benefit we'll get from going multicore.

Marketing literature from multicore vendors suggests that a two-core system can increase system performance from 30% to 50% (for desktop applications; how that scales to embedded systems is another question entirely, one that completely depends on the

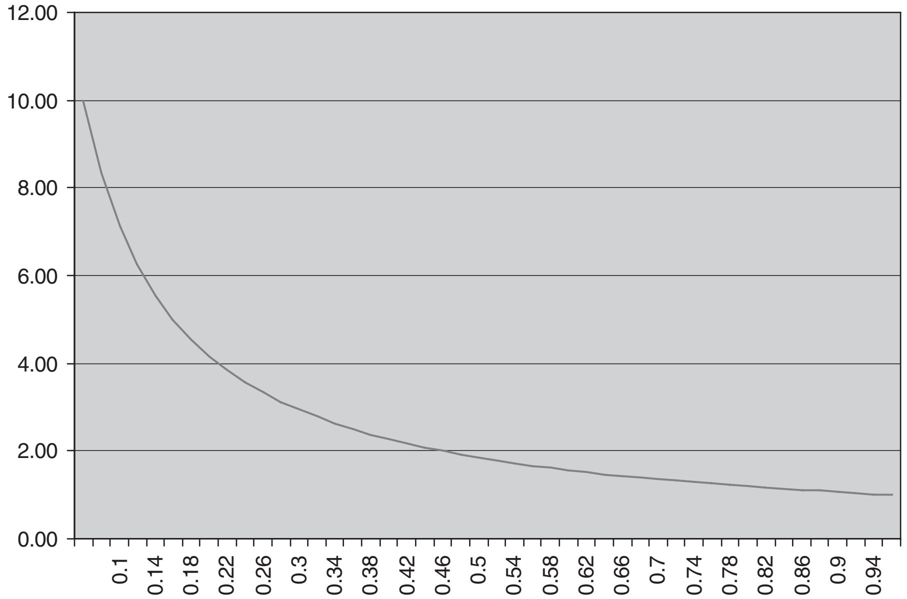


**Figure 2.7: Possible speedup versus percentage of the problem that cannot be parallelized, assuming an infinite number of cores**

application). Assuming the best case (50%) and working Amdahl's Law backward, one sees that the vendors assume a third of the PC programs can be parallelized. That's actually a best, best, case as a PC runs many different bits of software at the same time, and could simply split execution paths by application. But, pursuing this line of reasoning, assuming the dramatic 50% speed improvement comes from running one program, the law shows that with an infinite number of processors the best one could hope for would be a 3× performance boost (excepting the special case of intrinsically parallel programs).

Then there's the bus bottleneck.

Each of the twins in a dual-core SMP chip has its own zero wait state cache, which feeds instructions and data at sizzling rates to the CPU. But once off L1 they share an L2, which though fast, stalls every access with a couple of wait states. Outside of the L2, a single bus serves two insanely high-speed processors that have ravenous appetites for memory cycles, cycles slowed by so many wait states as to make the processor clock rate for off-chip activity almost irrelevant.

And here's the irony: a multi-GHz CPU that can address hoards of GB of memory, that has tens of millions of transistors dedicated to speeding up operations, runs mind-numbingly fast only as long as it executes out of L1, which is typically a microscopic 32–64 KB. PIC-sized. Run a bigger program, or one that uses lots of data, and the wait state logic jumps on the brakes.

A couple of Z80s might do almost as well.

In the embedded world we have more control of our execution environment and the program itself than in a PC. Some of the RTOS vendors have come up with clever ways to exploit multicore more efficiently, such as pinning tasks to particular cores. I have seen a couple of dramatic successes with this approach. If a task fits entirely within L1, or even spills over to L2, expect tremendous performance boosts. But it still sort of hurts one's head—and pocketbook—to constrain such a high-end CPU to such small hunks of memory.

Any program that runs on and off cache may suffer from determinism problems. What does "real time" mean when a cache miss prolongs execution time by perhaps an order of magnitude or more? Again, your mileage may vary as this is an extremely application-dependent issue, but proving a real-time system runs deterministically is hard at best.

Cache, pipelines, speculative execution, and now two CPUs competing for the same slow bus all greatly complicate the issue. By definition, a hard real-time system that misses a deadline is as broken as one that has completely defective code.

Multicore does address a very important problem, that of power consumption. Some vendors stress that their products are more about MIPs/watt than raw horsepower. Cut the CPU clock a bit, double the number of processors, and the total power needs drop dramatically. With high-end CPUs sucking 100 watts or more (at just over a volt; do the math and consider how close that is to the amps needed to start a car), power is a huge concern, particularly in embedded systems. Most of the SMP approaches that I've seen, though, still demand tens of watts, far too much for many classes of embedded systems.

One wonders if a multicore approach using multiple 386 s stripped of most of their fancy addressing capability and other bus management features, supported by lots of "cache," or at least fast on-board RAM, wouldn't offer a better MIPS/watt/price match, at least in the embedded space where gigantic applications are relatively rare.

Finally, the holy grail of SMP for 30 years has been an auto-parallelizing compiler, something that can take a sequential problem and divide it among many cores. Progress has been made, and much work continues. But it's still a largely unsolved problem that is being addressed in the embedded world at the OS level. QNX, Green Hills, and others have some very cool tools that partition tasks both statically and dynamically among cores. But expect new sorts of complex problems that make programming a multicore system challenging at best.

While this rant may be seen by some to be completely dismissive of multicore that's not the case at all; my aim is to shine a little light into the marketing FUD that permeates multicore, as it does with the introduction of any new technology. Multicore processors are here to stay, and do offer some important benefits. You may find some impressive performance gains by employing SMP, depending upon your specific application.

As always, do a careful analysis of your particular needs before making a possibly expensive foray into a new technology.

### 2.1.8 Conclusion

In 1946 only one programmable electronic computer existed in the entire world. A few years later dozens existed; by the 60 s, hundreds. These computers were still so fearsomely expensive that developers worked hard to minimize the resources their programs consumed.

Though the microprocessor caused the price of compute cycles to plummet, individual processors still cost many dollars. By the 1990s, companies such as Microchip, Atmel, and others were already selling complete microcontrollers for subdollar prices. For the first time most embedded applications could cost-effectively exploit partitioning into multiple CPUs. However, few developers actually *do* this; the non-linear schedule/LOC curve is nearly unknown in embedded circles.

Smaller systems contain fewer bugs, of course, but also tend to have a much lower defect *rate*. A recent study [10] by Chu, Yang, Chelf, and Hallem evaluated Linux version 2.4. In this released and presumably debugged code, an automatic static code checker identified many hundreds of mistakes. Error rates for big functions were two to six times higher than for smaller routines.

Partition to accelerate the schedule … and ship a higher quality product.

NIST (the National Institute of Standards and Technology) found [11] that poor testing accounts for some $22 billion in software failures each year. Testing is hard; as programs grow the number of execution paths explodes. Robert Glass estimates [12] that for each 25% increase in program size, the program complexity—represented by paths created by function calls, decision statements, and the like—doubles.

Standard software-testing techniques simply don't work. Most studies find that conventional debugging and QA evaluations find only half the program bugs.

In no other industry can a company ship a poorly tested product, often with known defects, without being sued. Eventually the lawyers will pick up the scent of fresh meat in the firmware world.

Partition to accelerate the schedule, ship a higher quality product … and one that's been properly tested. Reduce the risk of litigation.

Extreme partitioning *is* the silver bullet to solving the software productivity crisis.

Obviously, not all projects partition as cleanly as those described here. But only a very few systems fail to benefit from clever partitioning.

## 2.2 Scheduling

1986 estimates pegged F-22 development costs at $12.6 billion. By 2001, with the work nearing completion, the figure climbed to $28.7 billion. The 1986 schedule appraisal of 9.4 years soared to 19.2 years.

A rule of thumb suggests that smart engineers double their estimates when creating schedules. Had the DoD hired a grizzled old embedded systems developer back in 1986 to apply this 2x factor the results would have been within 12% on cost and 2% on schedule.

Interestingly, most developers spend about 55% of their time on a project, closely mirroring our intuitive tendency to double estimates. The other 45% goes to supporting other projects, attending company meetings, and dealing with the very important extra-project demands of office life.

Traditional Big-Bang delivery separates a project into a number of sequentially performed steps: requirements analysis, architectural design, detailed design, coding, debugging, test, and, with enormous luck, delivery to the anxious customer. But notice there's no explicit scheduling task. Most of us realize that it's dishonest to even attempt an estimate till the detailed design is complete, as that's the first point at which the magnitude of the project is really clear. Realistically, it may take months to get to this point.

In the real world, though, the boss wants an accurate schedule by Friday.

So we diddle triangles in Microsoft Project, trying to come up with something that seems vaguely believable, though no one involved in the project actually credits any of these estimates with truth. Our best hope is that the schedule doesn't collapse till late into the project, deferring the day of reckoning for as long as possible.

In the rare (unheard of?) case where the team does indeed get months to create the complete design before scheduling, they're forced to solve a tough equation: schedule = effort/productivity. Simple algebra, indeed, yet usually quite insolvable. How many know their productivity, measured in lines of code per hour or any other metric?

Alternatively, the boss benevolently saves us the trouble of creating a schedule by personally, defining the end date. Again there's a mad scramble to move triangles around in Project to, well, not to create an accurate schedule, but to make one that's somewhat believable. Until it inevitably falls apart, again hopefully not till some time in the distant future.

Management is quite insane when using either of these two methods. Yet they do need a reasonably accurate schedule to coordinate other business activities. When should ads start running for the new product? At what point should hiring start for the new production line? When and how will the company need to tap capital markets or draw down the line of credit? Is this product even worth the engineering effort required?

Some in the Agile software community simply demand that the head honcho toughen up and accept the fact that great software bakes at its own rate. It'll be done when it's done. But the boss has a legitimate need for an accurate schedule early. And we legitimately cannot provide one without investing months in analysis and design. There's a fundamental disconnect between management's needs and our ability to provide.

There is a middle way.

Do some architectural design, bring a group of experts together, have them estimate individually, and then use a defined process to make the estimates converge to a common meeting point. The technique is called Wideband Delphi, and can be used to estimate nearly anything from software schedules to the probability of a spacecraft failure. Originally developed by the Rand Corporation in the 1940s, Barry Boehm later extended the method in the 1970s.

### *2.2.1 Wideband Delphi*

The Wideband Delphi (WD) method recognizes that the judgment of experts can be surprisingly accurate. But individuals often suffer from unpredictable biases, and groups may exhibit "follow the leader" behavior. WD shortcuts both problems.

WD typically uses three to five experts—experienced developers, people who understand the application domain and who will be building the system once the project starts. One of these people acts as a moderator to run the meetings and handle resulting paperwork.

The process starts by accumulating the specifications documents. One informal survey at the Embedded Systems Conference a couple of years ago suggested that 46% of us get no specs at all, so at the very least develop a list of features that the marketing droids are promising to customers.

More of us should use features as the basis for generating requirement documents. Features are, after all, the observable behavior of the system. They are the only thing the customer—the most important person on the project, he who ultimately pays our salaries—sees.

Consider converting features to use cases, which are a great way to document requirements. A use case is a description of a behavior of a system. The description is written from the point of view of a user who has just told the system to do something in particular, and is written in a common language (English) that both the user and the programmer understand. A use case captures the *visible* sequence of events that a system goes through in response to a *single* user stimulus. A visible event is one the user can see. Use cases do not describe hidden behavior at all.

While there is a lot to like about using UML, it is a complex language that our customers will never get. UML is about as useful as Esperanto when discussing a system with non-techies. Use cases grease the conversation.

Here's an example for the action of a single button on an instrument:

- Description: Describes behavior of the "cal" button
- Actors: User
- Preconditions: System on, all self-tests OK, standard sample inserted
- Main Scenario: When the user presses the button the system enters the calibrate mode. All three displays blank. It reads the three color signals and applies constants ("calibration coefficients") to make the displayed XYZ values all 100.00. When the cal is complete, the "calibrated" light comes on and stays on.
- Alternative Scenario: If the input values are below (20, 20, 20) the wrong sample was inserted or the system is not functioning. Retry three times then display "—", and exit, leaving the "calibrate" light off.
- Postconditions: The three calibration constants are stored and then all other readings outside of "cal" mode are multiplied by these.

There are hidden features as well, ones the customer will never see but that we experts realize the systems needs. These may include debouncing code, an RTOS, protocol stacks, and the like. Add these derived features to the list of system requirements.

Next select the metric you'll use for estimation. One option is lines of code. Academics fault the LOC metric and generally advocate some form of function points as a replacement. But most of us practicing engineers have no gut feel for the scale of a routine with 100 function points. Change that to 100 LOC and we have a pretty darn good idea of the size of the code.

Oddly, the literature is full of conversions between function points and LOC. On average, across all languages, one FP burns around 100 lines of code. For C++ the number is in the 1950s. So the two metrics are, for engineering purposes at least, equivalent.

Estimates based on either LOC or FP suffer from one fatal flaw. What's the conversion factor to months? Hours are a better metric.

Never estimate in weeks or months. The terms are confusing. Does a week mean 40 work hours? A calendar week? How does the 55% utilization rate factor in? Use hours of time spent actively working on the project.

## *2.2.2 The Guesstimating Game*

The moderator gives all requirements including both observed and derived features to each team member. A healthy company will give the team time to create at least high level designs to divide complex requirements into numerous small tasks, each of which gets estimated via the WD process.

Team members scurry off to their offices and come up with their best estimate for each item. During this process they'll undoubtedly find missing tasks or functionality, which gets added to the list and sized up.

It's critical that each person log any assumptions made. Does Tom figure most of `Display_result ()` is reused from an earlier project? That has a tremendous impact on the estimate.

Team members must ignore schedule pressure. If the boss wants it done January 15 maybe it's time to bow to his wishes and drop any pretense at scheduling. They also assume that they themselves will be doing the tasks.

All participants then gather in a group meeting, working on a single feature or task at a time. The moderator draws a horizontal axis on the white board representing the range of estimates for this item, and places Xs to indicate the various appraisals, keeping the source of the numbers anonymous (Figure 2.8).

Generally there's quite a range of predictions at this point; the distribution can be quite disheartening for someone not used to the WD approach. But here's where the method shows its strength. The experts discuss the results *and the assumptions* each has made. Unlike other rather furtive approaches, WD shines a 500,000 candlepower spotlight into the estimation process. Typically you'll hear discussions like:

> "I figured 8 hours since we already have the board support package." "Yeah, but it doesn't work. Proved that on the last project."
>
> "It's just sucking in A/D data—no big deal." "Except we need a complicated least squares fit in real time since the data is so noisy … that'll cost ya lots of hours."

Uncertainties, the biggest schedule killers, are exposed.

The moderator plots the results of a second round of estimates, done in the meeting and still anonymously. The results nearly always start to converge.

The process continues till four rounds have occurred, or till the estimates have converged sufficiently, or no one is willing to yield anymore (Figure 2.9).

Compute the average of the estimates for each feature ($x_i$), and the average of all of the estimates:

$$\bar{x} = \frac{\sum x_i}{n}$$

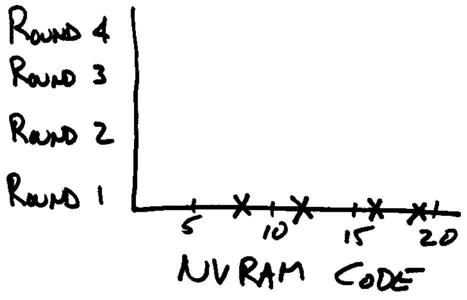


**Figure 2.8: First round of estimation on the whiteboard**

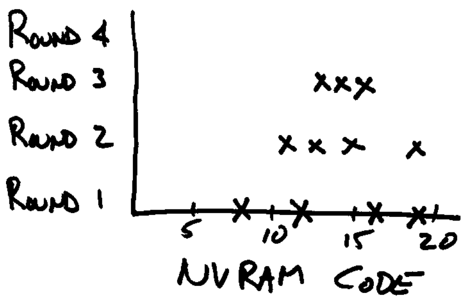


**Figure 2.9: Third round of estimation on the whiteboard**

where *n* is the number of features estimated, and the standard deviation of each:

$$\sigma_i = \sqrt{\frac{1}{n}\sum(x_i - \overline{x})^2}$$

Sum the average feature estimates to create the final project schedule:

$$S_t = \sum x_i$$

Combine all of the standard deviations:

$$\sigma_t = \sqrt{\sum(\sigma_i)^2}$$

Here's the hard part. We don't give the boss what he wants.

No one believes that there are any probability distributions that look like Figure 2.10.

That just doesn't happen in the real world.

There's a 0% chance we'll deliver in 10 msec, and a 100% chance it will be done before the year 3000 AD. Between those two extremes the probability resembles a curve, and this is the data we'll give the boss. That's the hard part; giving an answer that doesn't fit the boss's expectations.

In science, data is often distributed about the Normal (or Gaussian) bell curve. But that shape doesn't really reflect schedule distributions, as there are a few ways things can go really well to deliver unexpectedly early, but a million things that can go wrong. In practice it has been found that a Rayleigh distribution more closely reflects project behavior (Figure 2.11).

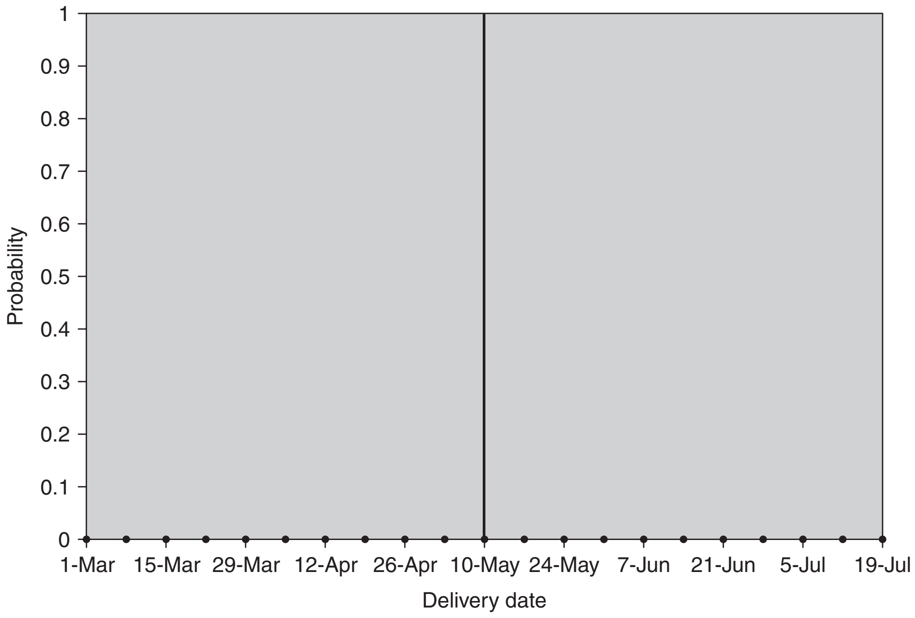


**Figure 2.10: Probability distribution**

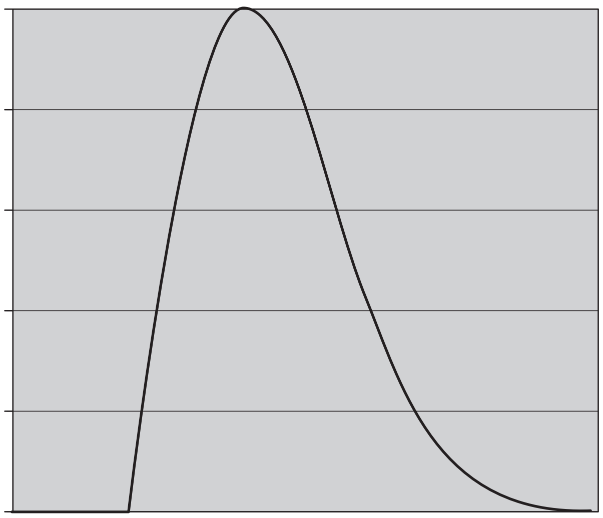


**Figure 2.11: The Rayleigh distribution**

There's a 50% chance we'll be done at the peak of the curve, which is the average of the estimates, $S_i$. Give the boss that number, and make it clear the chances of delivering then are only 50%.

Then add twice the total standard deviation:

$$S_{upper} = S_t + 2\sigma_t$$

Give this number to the boss as well. We have an 89% chance of being done in that amount of time. Not perfect, but darn near an "A" on any test.

(Astute readers may think that 89% is wrong; that twice sigma is 95%. That's true for a Normal curve, but the different shape of the Rayleigh changes the numbers a bit).

We give the boss a range of results because this is "estimation." A single fixed point would be called "exactimation," which never happens in scheduling.

### *2.2.3 A Few More Thoughts*

An important outcome of WD is a heavily refined specification. The discussion and questioning of assumptions sharpen vague use cases. When coding does start, it's much clearer what's being built, which, of course, only accelerates the schedule. No single estimate for any feature will be particularly accurate, but WD relies on the law of large numbers. It works … and is fun!

WD requires that we look in detail into what we're building. The alternative is clairvoyance. You might as well hire a gypsy with a crystal ball.

It won't work with vague specifications, unless the engineers can hammer out those details during the WD process. It is impossible to estimate the unknown.

WD will succeed in the face of changing requirements, though, if an effective change control procedure is in place. Change is inevitable, even good. But it has a cost which should be quickly estimated, presented to the customer/boss, and folded into the project plan.

WD's output is a range of hours actually spent building the system. You can't divide these by 40 to get weeks, as we're never fully occupied on one project. Either collect actual utilization figures at your company and scale the hours by that, or divide the WD results by 0.55 as noted earlier.

## References

1. Jones, C. *Applied Software Measurement: Assuring Productivity and Quality*. New York, NY: McGraw Hill, 1991.
2. Brooks, F. *The Mythical Man-Month: Essays on Software Engineering 20th Anniversary Edition*. New York, NY: Addison-Wesley Professional, 1995.
3. DeMarco, T. and Lister, T. *Peopleware: Productive Projects and Teams*. New York, NY: Dorset House Publishing Company, 1999.
4. Jones, C. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall, 1994.
5. Boehm, B. *Software Engineering Economics*. Upper Saddle River, NJ: Prentice Hall, 1981.
6. Augustine, N. *Augustine's Laws*. Reston, VA: AIAA (American Institute of Aeronautics & Astronautics), 1997.
7. Davis, A. *201 Principles of Software Development*. New York, NY: McGraw-Hill, 1995.
8. Jones, C. Private study conducted for IBM, 1977.
9. Kruger, J. and Dunning, D. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6), 1999 December.
10. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating system errors. *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.
11. RTI. The Economic Impact of Inadequate Software Testing, Planning Report 02–3, May 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf.
12. Glass, R. *Facts and Fallacies of Software Engineering*. New York, NY: Addison-Wesley, 2002.

CHAPTER 3

# The Code

## 3.1 Firmware Standards

English is a complex and wonderfully expressive language, full of oddball peculiarities, homonyms, and perverse punctuations. Anyone who reads USENET, email, or even (alas) billboards, sees how few Americans know the difference between their, there, and they're. Than and then get confused almost as often as affect and effect. And is it its or is it it's?

Lynne Truss humorously pointed out that a Panda "eats shoots & leaves." Add a comma, the tiniest of all typographical constructs, after "eats" and the phrase now expresses an entirely different idea.

Do you lie down or lay down? What does "lay" mean, anyway? Just the verb form has 52 different definitions in *The American Heritage Dictionary of the English Language*. As a noun it can be a ballad; as an adjective it denotes a secular person.

Then there's "lie," which means reclining, being mendacious, the hiding place of an animal, and more. Sometimes the context makes the meaning clear, which creates this epistemological puzzle that words themselves carry only a fraction of one's intent. Since a sentence may have many words whose meaning recursively depends on context it's a wonder we manage to communicate at all.

C, too, is a wonderfully expressive language, one whose grammar lets us construct torturous though perfectly legal constructions, so:

```
****************i=0;
```

is semantically and linguistically correct, yet solving Poincare's Conjecture might be a shade easier than following that chain of indirection. Just a few deft lines of nested `ifs` can yield a veritable explosion of possible permutations that the compiler accepts with the wisdom of Buddha. But humans and lesser gods cannot decipher all of the inscrutable possibilities.

The International Obfuscated C Code Contest (www.ioccc.org) runs a competition to, ah, "honor" incomprehensible yet working C programs. One winner in 2004 supplied the code in Figure 3.1, which (somehow) graphs polynomials.

Any computer language that lets us construct such convoluted and un-maintainable bits of agony must be eliminated or tamed. C isn't going away, so it's up to us to use it in a grown-up fashion that eases debug, test, and maintenance.

Post-processing tools like Lint help find bugs, but they don't insure we're building clean code that people can understand. Complexity analyzers do flag modules that look difficult, but, again, only work on code we've already written. I ran the program in Figure 3.1 through a complexity analyzer and the tool sputtered out some warnings and then crashed.

We *must* write all of our code to comply with a firmware standard. If the code doesn't both work and clearly communicate its intent, it's junk. A standard helps to decryptify the

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define _            ;double
#define void              x,x
#define case(break,default)     break[O]:default[O]:
#define switch(bool)            ;for(;x<bool;
#define do(if,else)        inline(else)>int##if?
#define true               (--void++)
#define false              (++void--)

char*O=" <60>!?\\\n"_ doubIe[010]_ int0,int1 _ Iong=0 _ inIine(int eIse){int
OlO=!O _ l=!O;for(;OlO<010;++OlO)l+=(OlO[doubIe]*pow(eIse,OlO));return l;}int
main(int booI,char*eIse[]){int I=1,x=-*O;if(eIse){for(;I<010+1;I++)I[doubIe-1]
=booI>I?atof(I(eIse]):!O switch(*O)x++)abs(inIine(x))>Iong&&(Iong=abs(inIine(x
)));int1=Iong;main(-*O>>1,0);}else{if(booI<*O>>1){int0=int1;int1=int0-2*Iong/0
[O]switch(5[O]))putchar(x-*O?(int0>=inIine(x)&&do(1,x)do(0,true)do(0,false)
case(2,1)do(1,true)do(0,false)6[O]case(-3,6)do(0,false)6[O]-3[O]:do(1,false)
case(5,4)x?booI?0:6[O]:7[O])+*O:8[O]),x++;main(++booI,0);}}}
```

**Figure 3.1: Your worst nightmare: Maintaining this (reprinted with permission of the IOCCC)**

C; everyone on the team writes code the same way, so our eyes don't trip over stylistic issues when we're trying to extract meaning.

The standard insures all firmware developed at your company meets minimum levels of readability and maintainability. Source code has two equally important functions: it must *work*, and it must clearly *communicate how it works* to a future programmer, or to the future version of yourself. Just as a standard English grammar and spelling make prose readable, standardized coding conventions illuminate the software's meaning.

A peril of instituting a firmware standard is the wildly diverse opinions people have about inconsequential things. Indentation is a classic example: developers will fight for months over quite minor issues. The only important thing is to *make a decision*. "We are going to indent in this manner. Period." Codify it in the standard, and then hold all of the developers to those rules. Appendix A has one firmware standard; plenty of others exist as well.

### 3.1.1 Names

'Tis but thy name that is my enemy;

Thou art thyself, though not a Montague.

What's Montague? It is nor hand, nor foot,

Nor arm, nor face, nor any other part

Belonging to a man. O, be some other name!

What's in a name? That which we call a rose

By any other name would smell as sweet;

So Romeo would, were he not Romeo call'd,

Retain that dear perfection which he owes

Without that title. Romeo, doff thy name,

And for that name which is no part of thee

Take all myself.

Maybe to Juliet names were fungible, but names and words matter. Biblical scholars refute attacks on scripture by exhaustive analysis of the meaning of a single word of Greek or Aramaic, whose nuance may have changed in the intervening millennia, corrupting a particular translation.

In zoology the binomial nomenclature, originally invented by Carl Linnaeus, rigorously specifies how species are named. Genus names are *always* capitalized while the species name *never* is. That's the standardized way zoologists communicate. Break the standard and you're no longer speaking the language of science.

Names are so important there's an entire science, called *onomatology*, devoted to their use and classification.

In the computer business names require a level of precision that's unprecedented in the annals of human history. `Motor_start()` and `motor_start()` are as different as the word for "hair" in Urdu and Esperanto. Mix up "l" and "1" or "0" and "O" and you might as well be babbling in Babylonian. Yet, depending on the compiler implementation, `this_is_a_really_long_variable_name` and `this_is_a_really_long_variable_name_complete_nonsense` are identical.

Yet we still use "`i`", "`ii`", and "`iii`" (my personal favorite) for index variables. You have to admire anyone devoted to his family, but that's no excuse for the too common practice of using a spouse's or kid's name in a variable declaration.

Words matter, as do names. Don't call me "Dave." I won't respond. Don't call a variable `foobar`. It conveys nothing to a future maintainer. *Great* code requires a disciplined approach to naming.

There are some 7000 languages used today on this planet, suggesting a veritable Babel of poor communication. But only about 9 are spoken by more than 100 million people; 69 are known by 10 million or more. The top ranks are disputed, but speakers of Mandarin, Spanish, English, and perhaps Hindi far outnumber those for any other language. C itself is composed entirely of English words like "if," "while," and "for," though in many companies programmers comment in their native language. This mix can't lead to clarity, which is the overarching goal of naming and documenting.

Spelling matters. Misspelled words are a sign of sloppy work. Names must be spelled correctly.

Long names are a great way to convey meaning, but C99 requires that only the first 31 to 63 identifiers be significant for external and internal names, respectively. Restrict all names to 31 characters or less.

Don't redefine a name using C's scoping rules. Though legal, having two names with different meanings is confusing. Similarly, don't use names that differ only in case.

On the subject of case, it's pretty traditional to define macros and constants in uppercase while using a mix of cases for functions and variable names. That seems reasonable to me. But what about camel case? Or should I write that CamelCase? Or is it camelCase? Everyone has a different opinion. But camel case is merely an awkward way to simulate a space between words, which gets even more cryptic when using acronyms: `UARTRead`. Some advocate only capitalizing the first letter of an acronym, but that word-izes the acronym, torturing the language even more.

We really, really want to use a space, but the language recognizes the space character as an end-of-token identifier. The closest typographical character to space is underscore, so why not use that? `This_is_a_word` is, in my opinion, easier to grok while furiously scanning hundreds of pages of code than `thisIsAWord`. Underscore's one downside is that it eats away at the 31-character name size limit, but that rarely causes a problem.

#### *3.1.1.1 Types*

Developers have argued passionately both for and against Hungarian notation since it was first invented in the 1970s by space tourist Charles Simonyi. At first blush the idea is appealing: prefix variables with a couple of letters indicating the type, increasing the name's information density. Smitten by the idea years ago, I drank the Hungarian kool-aid.

In practice Hungarian makes the code ugly. Clean names get mangled. `szString` meaning "`String`" is zero-terminated. `uiData` flags an unsigned int. Then I found that when changing the code (after all, everything changes all the time) sometimes an int had to morph to a long, which meant editing every invocation of the name. One team I know avoids this problem by typedefing a name like `iName` to long, which means not only is the code ugly, but the Hungarian nomenclature lies to the unwary.

C types are problematic. Is an int 16 bits? 32? Don't define variables using C's int and long keywords; follow the MISRA standard and use the following typedefs to remove all ambiguity and to make porting much simpler:

| | |
|---|---|
| `int8_t` | 8-bit signed integer |
| `int16_t` | 16-bit signed integer |
| `int32_t` | 32-bit signed integer |
| `uint8_t` | 8-bit unsigned integer |
| `uint16_t` | 16-bit unsigned integer |
| `uint32_t` | 32-bit unsigned integer |

### *3.1.1.2 Forming names*

Linnaeus' hierarchy of names, which today consists of the kingdom, phylum, class, order, family, genus, and species, is reflected in biological names like *Homo sapiens*. The genus comes first, followed by the more specific, the species. It's a natural way to identify large sets. Start from the general and work toward the specific.

The same goes for variable and function names. They should start with the big and work toward the small. `Main_Street_Baltimore_MD_USA` is a lousy name as we're not sure till the very end which huge domain—the country—we're talking about. Better: `USA_MD_Baltimore_Main_Street`.

Yet names like `Read_Timer0()`, `Read_UART()`, or `Read_DMA()` are typical. Then there's a corresponding `Timer0_ISR()`, with maybe `Timer0_Initialize()` or `Initialize_Timer0()`. See a pattern? I sure don't.

Better:

```
Timer_0_Initialize()

Timer_0_ISR()

Timer_0_read()
```

With this practice we've grouped everything to do with Timer 0 together in a logical, Linnaean taxonomy. A sort will clump related names together.

In a sense this doesn't reflect English sentence structure. "Timer" is the object; "read" the verb, and objects come after the verb. But a name is not a sentence, and we do the best we can do in an imperfect world. German speakers, though, will find the trailing verb familiar.

Since functions usually do something, it's wise to have an action word, a verb, as part of the name. Conversely, variables are just containers and do nothing. Variable names should be nouns, perhaps modified by adjectives.

Avoid weak and non-specific verbs like "handle," "process," and "update." What does "`ADC_Handle()`" mean? "`ADC_Curve_Fit()`" conveys much more information.

Short throwaway variable names are fine occasionally. A single line `for` loop that uses the not terribly informative index variable "`i`" is reasonable if the variable is both used and disposed of in one line. If it carries a value, which implies context and semantics, across more than a single line of code, pick a better name.

#### *3.1.1.3 TLAs and cheating*

In Hodge, M. H. and Pennington, F. M. "Some Studies of Word Abbreviation Behavior," *Journal of Experimental Psychology*, 98(2):350–361, 1973, researchers had subjects abbreviate words. Other subjects tried to reconstruct the original words. The average success rate was an appalling 67%.

What does "Disp" mean? Is it the noun meaning the display hardware, or is it the verb "to display"? How about "Calc"? That could be percent calcium, calculate, or calculus.

With two exceptions never abbreviate a name. Likewise, with the same caveats, never use an acronym. Your jargon may be unknown to some other maintainer, or may have some other meaning. Clarity is our goal!

One exception is the use of industry-standard acronyms and abbreviations like LED, LCD, CRT, UART, etc. that pose no confusion. Another is that it's fine to use any abbreviation or acronym documented in a dictionary stored in a common header file. For example:

```
/* Abbreviation Table

* Dsply    == Display (the verb)

*  Disp    == Display (our LCD display)

* Tot      == Total
* Calc     == Calculation
* Val      == Value
* MPS      == Meters per second
* Pos      == Position
*/
```

I remember with some wonder when my college physics professor taught us to cheat on exams. If you know the answer's units it's often possible to solve a problem correctly just by properly arranging and canceling those units. Given a result that must be in miles per hour, if the only inputs are 10 miles and 2 hours, without even knowing the question it's a good bet the answer is 5 mph.

Conversely, ignoring units is a sure road to disaster. Is `Descent_Rate` meters per second? CM/sec? Furlongs per fortnight? Sure, the programmer who initially computed the result probably knows, but it's foolish to assume everyone is on the same page. Postfix all physical parameters with the units. `Descent_Rate_MPS` (note in the dictionary above I defined MPS). `Timer_Ticks`. `ADC_Read_Volts()`.

#### *3.1.1.4 Comments*

According to Henry Petroski, the first known book about engineering is the 2000-year-old work *De Architectura* by Marcus Vitruvius Pollio. It's a fairly complete description of how these skilled artisans created their bridges and tunnels in ancient Rome.

One historian said of Vitruvius and his book: "He writes in atrocious Latin, but he knows his business." Another wrote: "He has all the marks of one unused to composition, to whom writing is a painful task."

How little things have changed! Even two millennia ago engineers wrote badly yet were recognized as experts in their field. Perhaps even then these Romans were geeks. Were engineers from Athens Greek geeks?

Some developers care little about their poor writing skills, figuring they interact with machines, not people. And of course we developers just talk to other writing-challenged engineers anyway, right?

Wrong.

This is the communications age. The spoken and written word has never been more important. Consider how email has reinvigorated letter-writing … yet years ago philologists moaned about the death of letters.

Old timers will remember how engineers could once function perfectly with no typing skills. That seems quaint today, when most of us live with a keyboard all but strapped to our hands. Just as old-fashioned is the idea of a secretary transcribing notes and fixing spelling and grammar. Today it's up to *us* to express ourselves clearly, with only the assistance of a spellchecker and an annoyingly picky grammar engine.

Even if you're stuck in a hermietically sealed cubicle never interacting with people and just cranking code all day, you still have a responsibility to communicate clearly and grammatically with others. Software is, after all, a mix of computerese (the C or C++ itself) and comments (in America, at least, an English-language description meant for humans, not the computer). If we write perfect C with illegible comments, we're doing a lousy job.

Consistently well-done comments are rare. Sometimes I can see the enthusiasm of the team at the project's outset. The start-up code is fantastic. `Main()`'s flow is clear and well documented. As the project wears on functions get added and coded with less and less care. Comments like

```
/* ???? */
```

or my favorite:

```
/* Is this right? */
```

start to show up. Commenting frequency declines; clarity gives way to short cryptic notes; capitalization descends into chaotic randomness. The initial project excitement, as shown in the careful crafting of early descriptive comments, yields to schedule panic as the developers all but abandon anything that's not executable.

Onerous and capricious schedules are a fact of life in this business. It's natural to chuck everything not immediately needed to make the product work. Few bosses grade on quality of the source code. Quality, when considered at all, is usually a back-end complaint about all the bugs that keep surfacing in the released product, or the ongoing discovery of defects that pushes the schedule back further and further.

We firmware folks know that quality starts at the front-end, in proper design and implementation, using reasonable processes. Quality also requires fine workmanship. Our profession parallels that of the trade crafts of centuries ago. The perfect joint in a chair may be almost invisible, but will last forever. A shoddy alternative could be just as hard to see but is simply not acceptable. Professional pride mandates doing the right thing just because we know it's the best way to build the product.

Though we embedded people work at the border between hardware and software, where sometimes it's hard to say where one ends and the other starts, even hardware designers work in the spotlight. Their creations are subject to ongoing audits during manufacturing, test, and repair. Technicians work with the schematics daily. Faults glare from the page for everyone to see. Sloppy work can't be hidden.

(Now, though, ASICs, programmable logic, and high level synthesis can bury lots of evil in the confines of an inscrutable IC package. The hardware folks are inheriting all of the perils of software.)

In my experience software created without pride is awful. Shortcuts abound. The limited docs never mirror current reality. Error conditions and exceptions are poorly thought out. Programs written in C usually have no intrinsic array bounds checking; worse, the dynamic nature of pointers makes automatic run-time checks that much more problematic.

Someone versed in the functionality of the product—but not the software—should be able to follow the program flow by reading the comments without reference to the code itself. Code implements an algorithm; the comments communicate the code's operation to you and others. Maybe even to a future version of yourself during maintenance years from now.

Write every bit of the documentation (in the United States at least) in English. Noun, verb. Use active voice. Be concise; don't write the Great American Novel. Be explicit and complete; assume your reader has not the slightest insight into the solution of the problem. In most cases I prefer to incorporate an algorithm description in a function's header, even for well-known approaches like Newton's Method. A description that uses your variable names makes a lot more sense than "see any calculus book for a description." And let's face it: once carefully thought out in the comments it's almost trivial to implement the code.

Capitalize per standard English procedures. IT HASN'T MADE SENSE TO WRITE ENTIRELY IN UPPERCASE SINCE THE TELETYPE DISAPPEARED 25 YEARS

AGO. the common c practice of never using capital letters is also obsolete. Worst aRe the DevElopeRs wHo usE rAndOm caSe changeS. Sounds silly, perhaps, but I see a lot of this. And spel al of the wrds gud.

Avoid long paragraphs. Use simple sentences. "Start_motor actuates the induction relay after a three second pause" beats "this function, when called, will start it all off and flip on the external controller but not until a time defined in HEADER.H goes by."

Begin every module and function with a header in a standard format. The format may vary a lot between organizations, but should be consistent within a team. Every module (source file) must start off with a general description of what's in the file, the company name, a copyright message if appropriate, and dates. Start every function with a header that describes what the routine does and how, goes-intas and goes-outas (i.e., parameters), the author's name, date, version, a record of changes with dates, and the name of the programmer who made the change.

C lends itself to the use of asterisks to delimit comments, which is fine. This:

```
/**************
* comment *
**************/
```

is a lousy practice. If your comments end with an asterisk as shown, every edit requires fixing the position of the trailing asterisk. Leave it off, as follows:

```
/**************
 comment
**************/
```

Most modern C compilers accept C++'s double slash comment delimiters, which is more convenient than the /* */ C requires. Start each comment line with the double slash so the difference between comments and code is crystal clear.

Some folks rely on a fancy editor to clean up comment formatting or add trailing asterisks. Don't. Editors are like religion. Everyone has a own preference, each of which is configured differently. Someday compilers will accept source files created with a word processor which will let us define editing styles for different parts of the program. Till then dumb ASCII text formatted with spaces (not tabs) is all we can count on to be portable and reliable.

Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines which work together to perform a macro function.

Explain the meaning and function of every variable declaration. Long variable names are merely an *aid* to understanding; accompany the descriptive name with a deep, meaningful, prose description.

One of the perils of good comments—which is frequently used as an excuse for sloppy work—is that over time the comments no longer reflect the truth of the code. Comment drift is intolerable. Pride in workmanship means we change the docs as we change the code. The two things happen in parallel. Never defer fixing comments till later, as it just won't happen. Better: edit the descriptions first, and then fix the code.

One side effect of our industry's inglorious 50-year history of comment drift is that people no longer trust comments. Such lack of confidence leads to even sloppier work. It's hard to thwart this descent into commenting chaos. Wise developers edit the header to reflect the update for each patch, but even better add a note that says "comments updated, too" to build trust in the docs.

Finally, consider changing the way you write a function. I have learned to write all of the comments first, including the header and those buried in the code. Then it's simple, even trivial, to fill in the C or C++. Any idiot can write software following a decent design; inventing the design, reflected in well-written comments, is the really creative part of our jobs.

## 3.2 Code Inspections

There *is* a silver bullet that can drastically improve the rate you develop code while also reducing bugs. Though this bit of magic can reduce debugging time by an easy factor of 10 or more, despite the fact that it's a technique well known since 1976, and even though neither tools nor expensive new resources are needed, few embedded folks use it.

Formal code inspections are probably the most important tool you can use to get your code out faster with fewer bugs. The inspection plays on the well-known fact that "two heads are better than one." The goal is to identify and remove bugs *before* testing the code.

Those that are aware of the method often reject it because of the assumed “hassle factor.” Usually few developers are aware of the benefits that have been so carefully quantified over time. Let’s look at some of the data.

- The very best of inspection practices yield stunning results. For example, IBM manages to remove 82% of all defects before testing even starts!
- One study showed that, as a rule of thumb, each defect identified during inspection saves around 9 hours of time downstream.
- AT&T found inspections led to 14% increase in productivity and tenfold increase in quality.
- HP found 80% of the errors detected during inspections were unlikely to be caught by testing.
- HP, Shell Research, Bell Northern, and AT&T all found inspections 20–30 times more efficient than testing in detecting errors.
- IBM found inspections gave a 23% increase in productivity and a 38% reduction in bugs detected after unit test.

So, though the inspection may cost up to 20% more time up front, debugging can shrink by an order of magnitude or more. The reduced number of bugs in the final product means you’ll spend less time in the mind-numbing weariness of maintenance as well.

There is no known better way to find bugs than through code inspections! Skipping inspections is a sure sign of the amateur firmware jockey.

### 3.2.1 The Inspection Team

The best inspections come about from properly organized teams. *Keep management off the team.* Experience indicates that when a manager is involved usually only the most superficial bugs are caught, since no one wishes to show the author to be the cause of major program defects.

Four formal roles exist: the Moderator, Reader, Recorder, and Author.

The Moderator, always technically competent, leads the inspection process. He or she paces the meeting, coaches other team members, deals with scheduling a meeting place and disseminating materials before the meeting, and follows up on rework (if any).

The Reader takes the team through the code by paraphrasing its operation. Never let the Author take this role, since he or may read what he or she meant instead of what was implemented.

A Recorder notes each error on a standard form. This frees the other team members to focus on thinking deeply about the code.

The Author's role is to understand the errors and to illuminate unclear areas. As code inspections are never confrontational, the Author should never be in a position of defending the code.

An additional role is that of Trainee. No one seems to have a clear idea how to create embedded developers. One technique is to include new folks (only one or two per team) into the code inspection. The Trainee then gets a deep look inside of the company's code and an understanding of how the code operates.

It's tempting to reduce the team size by sharing roles. Bear in mind that Bull HN found four-person inspection teams are twice as efficient and twice as effective as three-person teams. A code inspection with three people (perhaps using the Author as the Recorder) surely beats none at all, but do try to fill each role separately.

#### *3.2.1.1 The process*

Code inspections are a *process* consisting of several steps; all are required for optimal results. The steps are:

*Planning*: When the code compiles cleanly (no errors or warning messages), and after it passes through Lint (if used) the Author submits listings to the Moderator, who forms an inspection team. The moderator distributes listings to each team member, as well as other related documents such as design requirements and documentation. The bulk of the Planning process is done by the Moderator, who can use email to coordinate with team members. An effective Moderator respects the time constraints of colleagues and avoids interrupting them.

*Overview*: This *optional* step is a meeting when the inspection team members are not familiar with the development project. The Author provides enough background to team members to facilitate their understanding of the code.

*Preparation*: Inspectors individually examine the code and related materials. They use a checklist to insure they check all potential problem areas. Each inspector marks up his or her copy of the code listing with suspected problem areas.

*Inspection Meeting*: The entire team meets to review the code. The Moderator runs the meeting tightly. The only subject for discussion is the code under review; any other subject is simply not appropriate and is not allowed.

The person designated as Reader presents the code by paraphrasing the meaning of small sections of code in a context higher than that of the code itself. In other words, the Reader is translating short code snippets from computer-lingo to English to insure the code's implementation has the correct meaning.

The Reader continuously decides how many lines of code to paraphrase, picking a number that allows reasonable extraction of meaning. Typically he's paraphrasing two to three lines at a time. He paraphrases every decision point, every branch, case, etc. One study concluded that only 50% of the code gets executed during typical tests, so be sure the inspection looks at *everything*.

Use a checklist to be sure you're looking at all important items. See the "Code Inspection Checklist" for details. Avoid ad hoc nitpicking; follow the firmware standard to guide all stylistic issues. Reject code that does not conform to the letter of the standard.

Log and classify defects as Major or Minor. A Major bug is one that could result in a problem visible to the customer. Minor bugs are those that include spelling errors, non-compliance with the firmware standards, and poor workmanship that does not lead to a major error.

Why the classification? Because when the pressure is on, when the deadline looms near, management will demand that you drop inspections as they don't seem like "real work." A list of classified bugs gives you the ammunition needed to make it clear that dropping inspections will yield more errors and slower delivery.

Fill out two forms. The "Code Inspection Checklist" is a summary of the number of errors of each type that's found. Use this data to understand the inspection process's effectiveness. The "Inspection Error List" contains the details of each defect requiring rework.

The code itself is the only thing under review; the Author may not be criticized. One way to defuse the tension in starting up new inspection processes (before the team members are truly comfortable with it) is to have the Author supply a pizza for the meeting. Then he seems like the good guy.

At this meeting make no attempt to rework the code or to come up with alternative approaches. Just find errors and log them; let the Author deal with implementing solutions. The Moderator must keep the meeting fast-paced and efficient.

Note that comment lines require as much review as code lines. Misspellings, lousy grammar, and poor communication of ideas are as deadly in comments as outright bugs in code. Firmware must both *work* and *communicate its meaning*. The comments are a critical part of this and deserve as much attention as the code itself.

It's worthwhile to compare the size of the code to the estimate originally produced (if any!) when the project was scheduled. If it varies significantly from the estimate, figure out why, so you can learn from your estimation process.

Limit inspection meetings to a maximum of 2 hours. At the conclusion of the review of each function decide whether the code should be accepted as is or sent back for rework.

*Rework*: The Author makes all suggested corrections, gets a clean compile (and Lint if used) and sends it back to the Moderator.

*Follow-up*: The Moderator checks the reworked code. Once the Moderator is satisfied, the inspection is formally complete and the code may be tested (Figure 3.2).

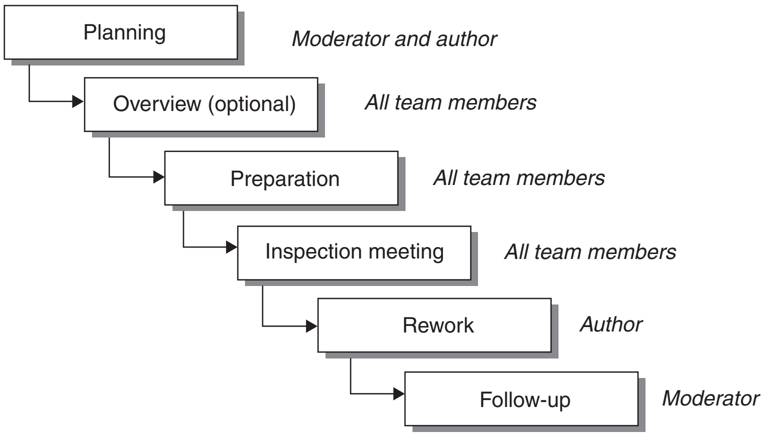


**Figure 3.2: The code inspection process**

#### *3.2.1.2 Other points*

One hidden benefit of code inspections is their intrinsic advertising value. We talk about software reuse, while all too often failing spectacularly at it. Reuse is certainly tough, requiring lots of discipline. One reason reuse fails, though, is simply because people don't know a particular chunk of code exists. If you don't know there's a function on the shelf, ready to rock 'n roll, then there's no chance you'll reuse it. When four people inspect code, four people have some level of buy-in to that software, and all four will generally realize the function exists.

The literature is full of the pros and cons of inspecting code before you get a clean compile. My feeling is that the compiler is nothing more than a tool, one that very cheaply and quickly picks up the stupid silly errors we all make. Compile first and use a Lint tool to find other problems. Let the tools—not expensive people—pick up the simple mistakes.

I also believe that the only good compile is a clean compile. No error messages. No warning messages. Warnings are deadly when some other programmer, maybe years from now, tries to change a line. When presented with a screenful of warnings he'll have no idea if these are normal or a symptom of a newly induced problem.

Do the inspection post-compiler but pre-test. Developers often ask if they can do "a bit" of testing before the inspection—surely only to reduce the embarrassment of finding dumb mistakes in front of their peers. Sorry, but testing first negates most of the benefits. First, inspection is the cheapest way to find bugs; the entire point of it is to avoid testing. Second, all too often a pre-tested module never gets inspected. "Well, that sucker works OK; why waste time inspecting it?"

Tune your inspection checklist. As you learn about the types of defects you're finding, add those to the checklist so the inspection process benefits from actual experience.

Inspections work best when done quickly—but not too fast. Figure 3.3 graphs percentage of bugs found in the inspection versus number of lines inspected per hour as found in a number of studies. It's clear that at 500 lines per hour no bugs are found. At 50 lines per hour you're working inefficiently. There's a sweet spot around 150 lines per hour which detects most of the bugs you're going to find, yet keeps the meeting moving swiftly.

Code inspections cannot succeed without a defined firmware standard. The two go hand in hand. Without a standard inspectors argue over stylistic issues. With one, the moderator stops such debates with a simple "Does this conform to the standard?" A yes

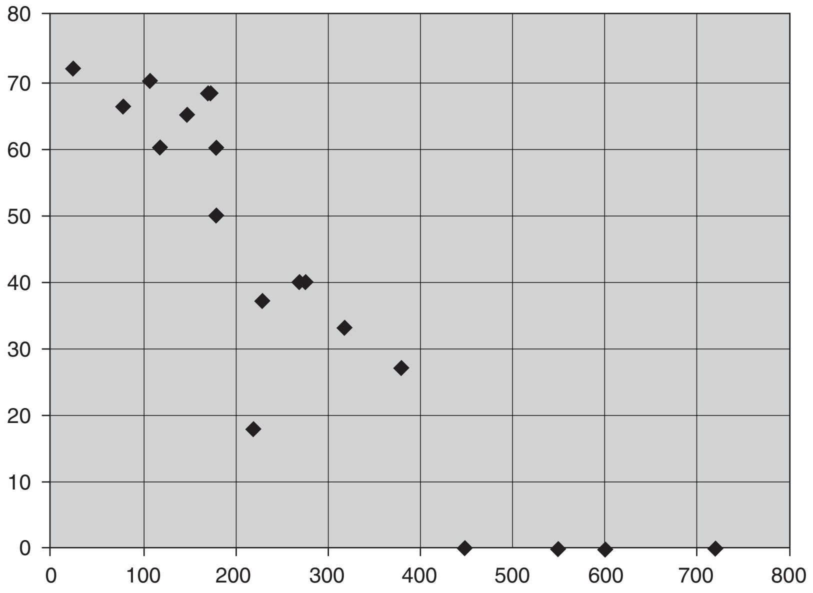


**Figure 3.3: Percentage of bugs found versus number of lines inspected per hour**

answer means discussion closed. A no means fix the code to conform to the standard. And without inspections, the use of the firmware standard will surely fail.

What does it cost to inspect code? We do inspections because they have a significant net negative cost. Yet sometimes management is not so sanguine; it helps to show the total cost of an inspection assuming there are *no* savings from downstream debugging.

The inspection includes four people: the Moderator, Reader, Recorder, and Author. Assume (for the sake of discussion) that these folks average an $80,000 salary, and overhead at your company is 100%, Then:

```
One person costs:    $160,000     = $80,000 * 2 (overhead)
One person costs:    $77/h = $160,000/2080 work hours/year
Four people cost:    $308/h       = $77/h * 4
Inspection cost/line: $2.05  = $308 per hour / 150 lines inspected per hour
```

Since we know code costs $20–50 per line to produce, this $2.5 cost is obviously in the noise. And this is only if inspections have no value.

**Code Inspection Checklist**

| | |
|---|---|
| Project: | ____________________________ |
| Author: | ____________________________ |
| Function Name: | ____________________________ |
| Date: | ____________________________ |

| *Number of errors* | | *Error Type* |
|---|---|---|
| Major | Minor | |
| | | Code does not meet firmware standards |
| | | Function size and complexity unreasonable |
| | | Unclear expression of ideas in the code |
| | | Poor encapsulation |
| | | Function prototypes not correctly used |
| | | Data types do not match |
| | | Uninitialized variables at start of function |
| | | Uninitialized variables going into loops |
| | | Poor logic — won't function as needed |
| | | Poor commenting |
| | | Error condition not caught (e.g., return codes from malloc())? |
| | | Switch statement without a default case (if only a subset of the possible conditions used)? |
| | | Incorrect syntax — such as proper use of ==, =, &&, &, etc. |
| | | Non reentrant code in dangerous places |
| | | Slow code in an area where speed is important |
| | | Other |
| | | Other |

*A major bug is one that if not removed could result in a problem that the customer will see. Minor bugs are those that include spelling errors, non-compliance with the firmware standards, and poor workmanship that does not lead to a major error.*

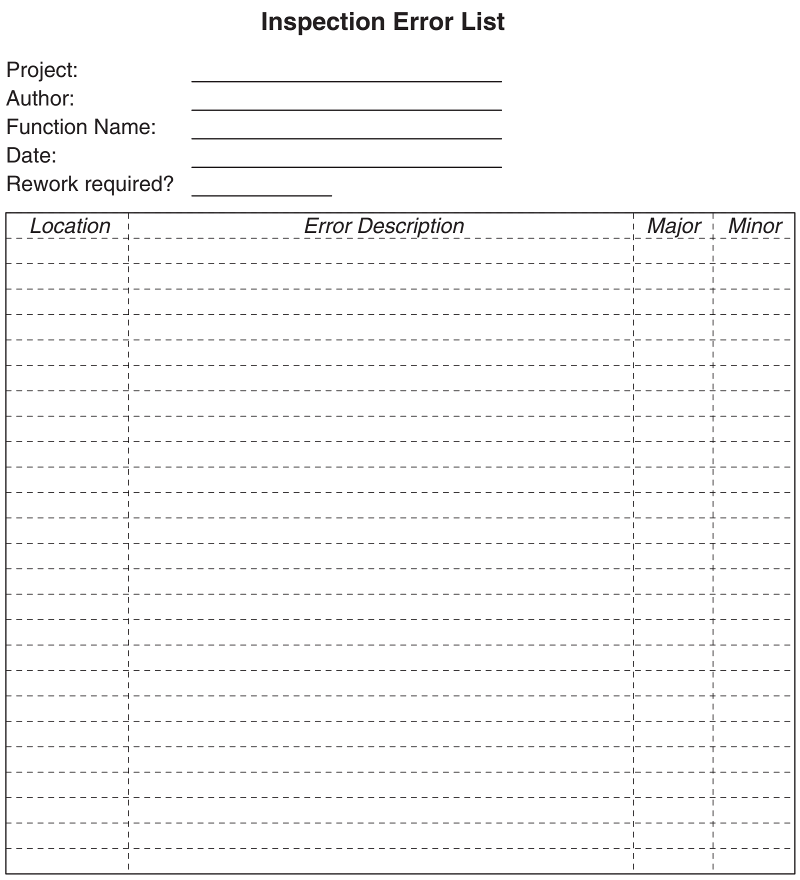
**Figure 3.4: Code inspection checklist**

### *3.2.1.3 Resources*

*Software Inspection* (Tom Gilb and Dorothy Graham, 1993, TJ Press (London)) is the bible about doing inspections. It's very complete, and is a sure cure for insomnia.

An alternative is Karl Weiger's excellent *Peer Reviews in Software*, 2001, Addison Wesley (Boston, MA) which covers traditional inspections as well as less formal ones using fewer people.

*Software Inspection – An Industry Best Practice* (David Wheeler, Bill Brykczynski, and Reginald Meeson, 1996 by IEEE Computer Society Press (CA)) is another useful resource. It's a collection of papers about successful and failed inspections. Don't read it, though. It's the book I keep at hand, so when I just can't stand the thought of another inspection, I read one paper at random. Like a tent revival meeting, it helps one continue doing the right thing even when conditions are hard (Figures 3.4 and 3.5).



**Inspection Error List**

Project: _____________________________

Author: _____________________________

Function Name: _____________________________

Date: _____________________________

Rework required? _____________

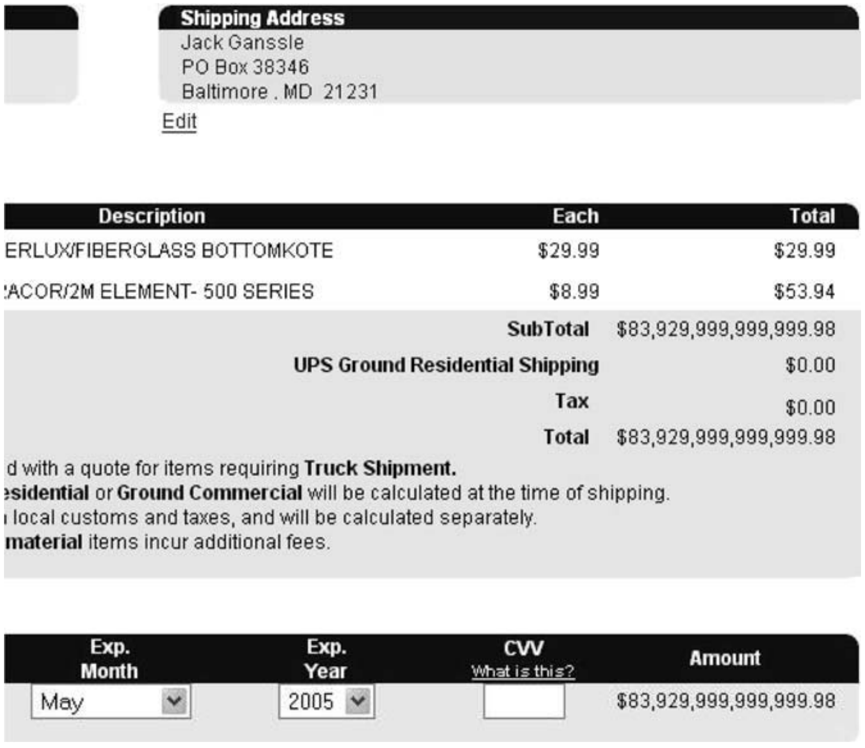
| *Location* | *Error Description* | *Major* | *Minor* |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

**Figure 3.5: Inspection error list**

## 3.3 Design by Contract™

It's hard to remember life before the web. Standing in lines to buy stuff and mail-ordering from out-of-date print catalogs were frustrations everyone accepted. But today many of those problems are long gone, replaced by sophisticated software that accepts orders on-line. For instance, Figure 3.6 shows the result of an order I tried to place with Defender last year. I was tempted to complete the transaction to see my credit card burst into flames.

At least they didn't charge tax, though it would be nearly enough to pay off the national debt.



**Shipping Address**
Jack Ganssle
PO Box 38346
Baltimore , MD 21231
Edit

| Description | Each | Total |
|---|---|---|
| ERLUX/FIBERGLASS BOTTOMKOTE | $29.99 | $29.99 |
| :ACOR/2M ELEMENT- 500 SERIES | $8.99 | $53.94 |
| | **SubTotal** | $83,929,999,999,999.98 |
| | **UPS Ground Residential Shipping** | $0.00 |
| | **Tax** | $0.00 |
| | **Total** | $83,929,999,999,999.98 |

d with a quote for items requiring **Truck Shipment.**
**esidential** or **Ground Commercial** will be calculated at the time of shipping.
ı local customs and taxes, and will be calculated separately.
**material** items incur additional fees.

| Exp. Month | Exp. Year | CVV What is this? | Amount |
|---|---|---|---|
| May | 2005 | | $83,929,999,999,999.98 |

**Figure 3.6: How did we manage in the pre-computer age?**

Thirty-five years ago, as student's in a college Fortran class, we were admonished by the instructors to check our results. Test the goesintas and goesoutas in every subroutine. Clearly, Defender's clever programmers never got this lesson. And, in most of the embedded code I read today few of us get it either. We assume the results will be perfect.

But they're often not. Ariane 5's maiden launch in 1996 failed spectacularly, at the cost of half a billion dollars, due to a minor software error: an overflow when converting a 64-bit float to a 16-bit integer. The code inexplicably did not do any sort of range checking on a few critical variables.

Every function and object is saddled with baggage, assumptions we make about arguments and the state of the system. Those assumptions range from the ontological nature of computers (the power is on) to legal variable ranges to subtle dependency issues (e.g., interrupts off). Violate one of these assumptions and Bad Things will happen.

Why should anything be right? Isn't it possible that through some subtle error propagated through layers of inscrutable computation a negative number gets passed to the square root approximation? If one reuses a function written long ago by some dearly departed developer it's all-too-easy to mistakenly issue the call with an incorrect argument or to make some equally erroneous assumption about the range of values that can be returned.

Consider:

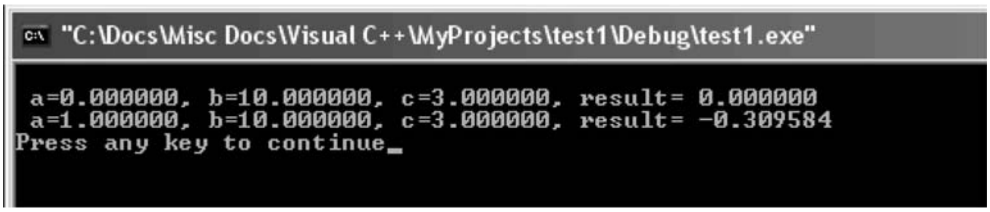
```
float solve_post(float a, float b, float c){
  float result;
  result=(-b + sqrt(b*b -4*a*c))/2*a;
  return result;}
```

… which computes the positive solution of a quadratic equation. I compiled it and a driver under Visual Studio, with the results shown in Figure 3.7.

The code runs happily even as it computes an impossible result, as is C's wont. The division by zero that results when "a" is zero creates neither an exception nor a signal to the user that something is seriously, maybe even dangerously, amiss.

This simple routine can fail in other ways, too. Consider the square root: we *know* that a negative argument isn't allowed. But that's not really true; that's an implicit assumption, which may not be correct. In the complex domain different assumptions apply.

Bertrand Meyer thinks languages should include explicit constructs that make it easy to validate the hidden and obvious assumptions that form a background fabric to our software. His must-read *Object-Oriented Software Construction* (at 1200 pages a massive



```
"C:\Docs\Misc Docs\Visual C++\MyProjects\test1\Debug\test1.exe"

 a=0.000000, b=10.000000, c=3.000000, result= 0.000000
 a=1.000000, b=10.000000, c=3.000000, result= -0.309584
Press any key to continue_
```

Figure 3.7: Result of solving a quadratic equation

tome which is sometimes pedantic but is brilliantly written) describes Eiffel, which supports Design By Contract (DBC, and a trademark of Eiffel Software). This work is the best extant book about DBC.

In important human affairs, from marriage to divorce to business relationships, we use contracts to clearly specify each party's obligations. The response "I do" to "Do you promise to love and to hold …" is a contract, and agreement between the parties about mutual expectations. Break a provision of that or any other contract and lawyers will take action to mediate the dispute … or to at least enrich themselves. Meyer believes software components need the same sort of formal and binding specification of relationships.

DBC asks us to explicitly specify these contractual relationships. The compiler will generate additional code to check them at run-time. That immediately rules the technique out for small, resource-constrained systems. But today, since a lot of embedded systems run with quite a bit of processor and memory headroom, it's not unreasonable to pay a small penalty to get significant benefits. If one is squeezed for resources, DBC is one of the last things to remove. What's the point of getting the wrong answer quickly?

And those benefits *are* substantial:

- The code is more reliable since arguments and assumptions are clearly specified in writing.
- They're checked every time the function is called so errors can't creep in.
- It's easier to reuse code since the contracts clearly specify behaviors; one doesn't have to reverse engineer the code.
- The contracts are a form of documentation that always stays in-sync with the code itself. Any sort of documentation drift between the contracts and the code will immediately cause an error.
- It's easier to write tests since the contracts specify boundary conditions.
- Function designs are clearer, since the contracts state, very clearly, the obligations of the relationship between caller and callee.
- Debugging is easier since any contractual violation immediately tosses an exception, rather than propagating an error through many nesting levels.

- Maintenance is easier since the developers, who may not be intimately familiar with all of the routines, get a clear view of the limitations of the code.
- Peace of mind. If anyone misuses a routine, he or she will know about it immediately.

Software engineering is topsy-turvy. In all other engineering disciplines it's easy to add something to increase operating margins. Beef up a strut to take unanticipated loads or add a weak link to control and safely manage failure. Use a larger resistor to handle more watts or a fuse to keep a surge from destroying the system. But in software a single bit error can cause the system to completely fail. DBC is like adding fuses to the code. If the system is going to crash, it's better to do so early and seed debugging breadcrumbs than fail in a manner that no one can subsequently understand.

## 3.3.1 Contracts

Let's look at a common process that occurs in many households. How does one get a teenager up for school? There are three main steps, though some may be repeated *N* times, where *N* may be a very large number:

- Wake up the kid
- Wake up the kid again
- Scream at the kid

If we were implementing these in software using DBC, the code might look something like this:

```
Wake_kid
      Precondition: Kid_is_in_bed // Kid hasn't snuck out
                                 // during the night

      Invariant: coffee machine is on
Wake_kid_again
      Precondition: kid_is_not_awake AND Wake_kid implemented once

      Invariant: coffee machine is on
Scream_at_kid
      Precondition: kid_is_not_awake AND (wake_kid called N times)
      Postcondition: kid_appears_awake
      Invariant: coffee machine is on
```

Each of the three routines has dependencies called "preconditions." A precondition must be satisfied before the function can execute. In `Wake_kid,` if the precondition `Kid_is_in_bed` is not true, then an error is thrown, the cops get called, and dad's ulcer acts up.

`Scream_at_kid` has a "postcondition," something that must be true when the routine exits. As long as the kid isn't conscious, the actor continues to scream.

All of the routines have "invariants," conditions that must be true both before the function starts and when it exits. Invariants are supra-functional aspects of system behavior. In this case if the coffee machine isn't on, life is barely worth living so an exception is trapped. An invariant could be included as part of the pre- and postcondition expressions, but repetition is fraught with errors. And, by specifically indicating that the condition is invariant, one makes a strong statement about that condition. Invariants are tested before the precondition check and after the postcondition.

Precondition faults signal a problem in the caller; postcondition problems mean the function failed.

A function (or class) is correct only if it satisfies the invariants and pre- and postconditions. These are formal contracts that the client and supplier make. In defensive programming one learns to check input arguments, for instance. Contracts are the meta-pattern of argument checking, as they insure the inputs, outputs, and stable, unchanging aspects are all correct.

Here's another example. When I do arithmetic by hand I usually end with a postcondition. To subtract 1.973 from 2.054 I'd do this:

```
 2.054
-1.973
-------
 0.081
+1.973
-------
 2.054
```

Long experience has taught me that my manual calculations are sometimes wrong, and a quick check costs little. That extra step of adding the result to the subtrahend is nothing more than a postcondition. In C:

```
float subtract(minuend, subtrahend){
  float temp;
  temp=minuend-subtrahend;
  postcondition: minuend == temp+subtrahend;
  return temp;
}
```

(assuming there were some construct like "postcondition:" in C). Clearly, this test guarantees the code is correct. Yup, it doubles the function's workload. But this is a trivial example; we'll see more realistic ones later.

How is this different from a C-style assertion? An assertion expresses the developer's belief about correctness at one point in time, at the beginning, end, or somewhere in the middle of the routine. DBC's constructs, instead, state things that are **Truth**. They're basic truisms about the structure of the system that stay carved in stone for longer than the scope of a single function. Though this may seem a minor philosophical point, I hope you'll see the distinction as this discussion unfolds.

Yes, it's possible to implement these ideas using assertions. But the concept transcends the limited single-node debugging zeitgeist of an assertion.

It's worth knowing a little about Eiffel even though the language has essentially zero market share in the embedded space, as Eiffel's built-in DBC support has influenced the language people use when discussing Design By Contract. The keyword "require" means a precondition. "Ensure" indicates a postcondition, and, unsurprisingly, "invariant" means just what you think.

## 3.3.2 Good Contracts

Contracts are not for detecting erroneous inputs. For instance, an error handler wouldn't use a contract to flag a typo from a keyboard … but you may use one if the keyboard burst into flames. Good contracts simply supervise relationships between the function and its caller. "I, the function, promise to fulfill these conditions." "And I, the caller, promise to fulfill these."

Software can fail due to outright bugs—programming mistakes—and exceptional conditions, events that are unwanted but not unexpected. Since there's usually no recovery from the former, it's wise to trap those with contracts. The latter covers events like missed data packets, bad input, and noisy analog, and should all be handled by code that takes some remedial action.

Good contracts supervise software-to-software relationships, not software to users.

Good contracts check things that seem obvious. Though any idiot knows that you can't pull from an empty queue, that idiot may be you months down the road when, in a panic to ship, that last minute change indeed does assume a "`get()`" is always OK.

Good contracts have no effects. In C, if the contract has a "=" there's something wrong as the state of the system gets changed each time the code tests the contract.

Good contracts have invariants for anything that's structurally important to the function's behavior that may not change during the execution of the function. For instance, if a function relies on a global variable remaining unchanged, write an invariant for that.

Put contracts on things that seem obvious, as months or years later a change in the code or an unexpected hardware modification may create error conditions. If the scale's hardware simply *can't* weigh anything over 100 grams … define a contract to insure that assumption never changes.

### 3.3.3 DBC in C

People familiar with DBC will be aghast that I'm torturing the concept into a procedural, rather than object-oriented, description. Yet today most embedded developers use C. It would be tragic to leave 70% of us unable to use these ideas.

DBC is hardly new but as yet has almost no penetration into the embedded space due to a lack of knowledge about the concept and the fact that mainstream languages provide no intrinsic support.

The time to start debugging is when you're writing the code, not when a bug rears up and slaps you in the forehead like a cold fish jumping out of the water. Since code typically has a 5–10% error rate after passing the compiler's syntax check, it's clear that a lot of what we produce will be wrong. DBC is perhaps the most powerful technique we have to toss a penalty flag when someone tries to use a function incorrectly.

Unfortunately DBC is not a part of C or C++. But there are some options. None is perfect, none achieves the clean and seamless DBC possibilities embodied in the Eiffel language. But a 90% solution counts as an A grade in most things. A first approximation to perfection is often good enough.

Charlie Mills wrote a free preprocessor for C programs that extracts contracts embedded in comments, turning them into normal C code. Let's look at some examples using the syntax that Charlie adopted. Since C doesn't accept any sort of DBC keywords, his preprocessor extracts mnemonics embedded in comments as follows:

```
/**
  pre: (precondition)
  post:(postcondition)
  inv: (invariant)
*/
```

That is, a set of contracts exists in its own comment block which starts with double asterisks. Consider that function that computes the positive root of a quadratic equation. Here's the same function instrumented with three contracts:

```
/* Find the positive solution of a quadratic */
/**
  pre: a != 0
  pre: (b*b-4*a*c) >= 0
  post: result != NaN
*/
float solve_pos(float a, float b, float c){
      float result;
      result= (-b + sqrt(b*b - 4*a*c))/2*a;
      return result;}
```

The triptych of contracts traps the three ways that this, essentially one line of code, can fail. Isn't that astonishing? A single line possibly *teeming* with mistakes. Yet the code is correct. Problems stem from *using* it incorrectly, the very issue DBC prevents. So rather than "failure" think of the contracts as provisions that define and enforce the relationship between the function and the caller. In this case, the caller may not supply zero for `a`, to avoid a divide by zero. If `(b*b-4*a*c)` is negative `sqrt()` will fail. And a very small `a` could overflow the capacity of a floating point number.

Run this through Charlie’s preprocessor and the result, edited for clarity, is:

```
float solve_pos(float a, float b, float c){
  float ocl__ret_val, result;
  /* Preconditions: */
  if (!(a != 0)) dbc_error("quad.c:3: " "precondition failed");
  if (!((b*b-4*a*c) >=0))dbc_error("quad.c:4: " "precondition failed");
  result= (-b + sqrt(b*b - 4*a*c))/2*a;
  ocl__ret_val = result; goto ocl__return_here;
ocl__return_here:

  /* Postconditions: */
  if (!(result != NaN))dbc_error("quad.c:5: " "postcondition failed");
  return ocl__ret_val;
}
```

Break a contract and `dbc_error()`—which you supply—runs. `printf` if you can, log the error, and halt if you must. But any attempt to use `solve_pos()` illegally will immediately flag a problem, long before an incorrect error would propagate through hundreds of other calls.

Computing the discriminant twice sounds computationally expensive! The gcc compiler, though, using the `-Os` (optimize for size) switch on an x86 does the math once and stores the result for use later. gcc also optimizes out the unneeded `goto`s. Contracts don’t necessarily add much of a memory or performance penalty.

In DBC contracts never cause side effects. Thus the preprocessor does not support the “=” operator.

The preprocessor has numerous options beyond what I’ve mentioned. See the references for the full syntax. I think some of the more complex features (which create loops) are not needed and are too computationally expensive.

Downsides? the preprocessor’s output is ugly. The indentation strategy (edited out above) just couldn’t be worse. I’ve found one bug: some constants with decimal points stop the preprocessor. The strings (e.g., “`quad.c:5 postcondition failed`”) are long and burn valuable memory. And wouldn’t it be nice for the program to optimize out unnecessary jumps? None of these is a showstopper, though, and a Ruby expert (the preprocessor is coded in Ruby) could quickly fix those problems.

### 3.3.4 Other Options

Falling back to Eiffel's DBC syntax, one can write `require()` (precondition) and `ensure()` (postcondition) macros. I have no idea how to create an invariant without using some sort of preprocessor, but you could embed the invariant's tests in the two new macros. Use:

```
void assert(int);
#define precondition(a) ((void)(a), assert(a))
#define postcondition(a) ((void)(a), assert(a))
```

Why the double arguments? Contracts should have no side effects. The assignment operator "=" and function calls aren't allowed. Couple the above definition with the use of Lint (you do use Lint, don't you? It's an essential part of any development environment) to flag a contract that uses assignments or function calls.

Using Gimpel's Lint (www.gimpel.com) add this command to the code:

```
//Lint -emacro(730,require)
```

so Lint won't complain about passing Booleans to functions.

Now, Lint will be happy with a contract like:

```
precondition(n >=0);
```

But will give warning 666 ("expression with side effects") for contracts like:

```
precondition(m=n);
```

or

```
precondition(g(m));
```

Contracts are also a form of documentation since they clearly describe the limits of a function's behavior. There's some merit in substituting constructs familiar only to C programmers with more English-like words, so I like to define `AND` and `OR` corresponding to `&&` and `||`, using the former in writing contracts.

The gnu Nana project (see Mitchell and McKim) offers another alternative. It's a collection of DBC-like constructs that do ultimately resolve to error handlers little different than `assert()` macros, but that are well thought out with an eye to the limited resources typical of our beloved embedded world. One option enables contracts using gdb resources, greatly reducing the overhead required by `assert()`'s traditional `__line__` and `__file__` parameters. Support of a limited amount of predicate calculus lets one define quite involved contracts.

Nana's use of `I()` for invariants and other too short, cryptic nomenclature is a mistake. Worse is that it supports assignment and function calls in contracts. But it's worth reading the (alas, all too limited) documents and code at the website.

There's at least one commercial DBC product that supports C and C++. The 249 C2 from case-challenged aechmea ties tightly to Microsoft's Visual Studio and preprocesses C files to convert contracts embedded in comments to code. The keywords are `@pre`, `@post`, and `@invariant`.

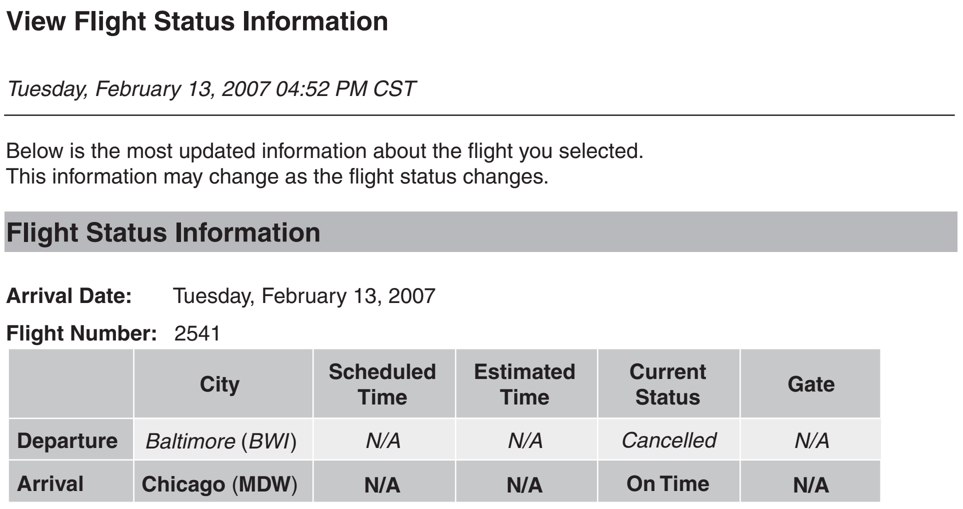
A free Linux version is available, but the license limits its use to non-commercial work.

aechmea's product works well; I'd be enthusiastic about it, except for its limiting Linux license and Windows operation solely with Visual Studio.

Eiffel does have a complete DBC mechanism. Yet Eiffel has only a microscopic presence in the embedded space. SPARK, a subset of Ada, is an intriguing alternative that includes pre-compile-time contract checking. That's not a misprint; it has an extremely expressive contract language that transcends the simple invariants, pre- and postconditions discussed here. SPARK's analysis tools examine the code *before* compilation, flagging any state the code can enter which will break a contract. That's a fundamentally different approach to the run-time checking of contracts, and to debugging in general. If you carefully define the contracts and fix any flaws detected by the tools, the SPARK tools can guarantee your program freedom from large classes of common defects, including all undefined and implementation-dependent behaviors, data-flow errors, and so-called "run-time errors" like buffer overflow, arithmetic overflow, and division by zero.

### *3.3.5 Conclusion*

One cold February day snowstorms generated a flurry of flight cancellations. Figure 3.8 shows that, though cancelled, my flight to Chicago was still expected to arrive on time.



View Flight Status Information

*Tuesday, February 13, 2007 04:52 PM CST*

Below is the most updated information about the flight you selected.
This information may change as the flight status changes.

Flight Status Information

**Arrival Date:** Tuesday, February 13, 2007

**Flight Number:** 2541

| | City | Scheduled Time | Estimated Time | Current Status | Gate |
|---|---|---|---|---|---|
| Departure | *Baltimore (BWI)* | *N/A* | *N/A* | *Cancelled* | *N/A* |
| Arrival | Chicago (MDW) | N/A | N/A | On Time | N/A |

**Figure 3.8: Southwest's flight status page**

Clearly, we programmers continue to labor under the illusion that, once a little testing is done, the code is perfect.

Is Design by Contract just a fancy name for the tried and tested `assert()` macro? I, and many others, argue *no* even though some C implementations merely expand DBC's keywords into `assert()` or a construct very much like it. DBC represents a different philosophy about writing programs. Programmers sprinkle assertions about wherever the spirit moves them (which usually means not at all). In DBC we define a function's behavior, to a certain extent, by the contracts. Functions routinely start and end with contracts. Contracts help document the code, yet can never suffer from documentation drift as the slightest misalignment with the code will instantly flag an error.

Contracts are truly about the way we *design* the code. Assertions check for errors in *implementation*.

eXtreme Programming, Test-Driven Development, and other agile methods stress the importance of writing test code early. DBC complements that idea, adding built-in tests to insure we don't violate design assumptions.

Write a function's comment header block first. Then write the contracts. Fill in the rest of the comments, and only then produce the code.

### 3.3.6 Resources

1. Charlie Mills's article on his DBC by C: http://www.onlamp.com/pub/a/onlamp/2004/10/28/design_by_contract_in_c.html
2. *Design By Contract, By Example*, by Richard Mitchell and Jim McKim (2002, Addison-Wesley, no city listed). A frustrating and badly organized book that offers good advice on designing good contracts.
3. *Practical Statecharts in C/C++* by Miro Samek (2002, CMP Books, Lawrence, KS). Though not about DBC, Miro does have some useful discussion of the subject.
4. Help with installing racc: http://www.dets-home.de/it-writings/Install_rd2_winnt.html
5. Also: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/9552614596b8c1d4/2899cc3bfd6b60e0?lnk=gst&q=racc+install&rnum=10&hl=en#2899cc3bfd6b60e0
6. And: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/57f387e0f96dc40e/11f8c95b71c90ccd?lnk=gst&q=racc+install&rnum=17&hl=en#11f8c95b71c90ccd
7. *Object-Oriented Software Construction* (1997, Prentice Hall PTR, Upper Saddle River, NJ).

## 3.4 Other Ways to Insure Quality Code

### 3.4.1 MISRA

Enter MISRA, the Motor Industry Software Reliability Association (http://www.misra.org.uk/). This consortium of automotive and related companies was formed to find better ways to build firmware for cars. For the vendors are terrified of software. Though it adds tremendous value to their products, the cost of defects is staggeringly high. One little bug can initiate a crippling recall. My wonderful hybrid Prius whose average 52 MPG results from very smart firmware was recalled last year due to a software problem. (Yes, the 52 is real, averaged over 37K miles, though unlike most of us my wife and I don't drive as if all the demons of hell are in hot pursuit.)

MISRA members range from AB Automotive to Visteon, and include Ford, Lotus, and others. Among other activities, MISRA produced a set of rules for programming in C. The standard is gaining traction in the automotive industry and others. It's available as a PDF from http://www.misra-c2.com/ for £10, or as a hardcopy in that annoying and difficult-for-Americans-to-file A4 format from the same site for £25.

For those who have the 1998 version things have changed. As of 2004 it was substantially updated and improved.

MISRA-C (2004) has 121 mandatory and 20 advisory rules. I guarantee you won't agree with all of them, but most are pretty reasonable and worth following. All derive from the following five principles:

1. C is incompletely specified. How does `process(j++, j);` behave? And exactly what is the size of an `int`? How astounding that such a basic component of any program is undefined!
2. Developers make mistakes, and the language does little to point out many of the even obvious screwups. It's so easy to mix up "=" and "==".
3. Programmers don't always have a deep knowledge of the language and so make incorrect assumptions.
4. Compilers have bugs, or purposely deviate from the ANSI standard. Most 8051 compilers, for instance, have run-time packages that take and return single precision results for trig functions instead of the prescribed doubles.
5. C offers little intrinsic support for detecting run-time errors.

The MISRA-C standard does not address stylistic issues, like indentations and brace placement. Only the bravest dare propose that his or her brace placement rules were Intelligently Designed. As the saying goes, put two programmers in a room and expect three very strong opinions.

Some of the rules are just common sense. For instance:

- Rule 1.2: No reliance shall be placed on undefined or unspecified behavior.
- Rule 9.1: All automatic variables shall have been assigned a value before being used.
- Rule 14.1: There shall be no unreachable code.

Other guidelines aren't particularly controversial. For instance:

- Rule 6.3: Typedefs that indicate size and signedness should be used in place of the basic types.

`Int, long, float, double`, their names are legion, the meanings vague. It's much better to use `int16_t`, for instance, instead of `int`; and `int32_t` instead of, well, `int` or `long`. Oddly, this rule is advisory only. In my opinion it's an absolute requirement.

- Rule 16.10: If a function returns error information, then that error information shall be tested.

Duh, right? Yet how often do we see an "`if`" testing `malloc()`'s return value?

- Rule: 7.1: Octal constants (other than zero) and octal escape sequences shall not be used.

The Univac 1108's 36-bit word was a perfect match to octal, but hex is a much better representation for 8-, 16-, and 32-bit computers. Octal should have died long ago.

You may have problems with some of the rules, though. Consider:

- Rule 2.2: Source code shall only use /* … */ style comments.

The rationale is that MISRA-C is based on C90, which doesn't permit the nice double slash comments we've swiped from C++.

- Rule 16.2: Functions shall not call themselves, either directly or indirectly.

Recursion makes my head hurt. It sure is useful for some limited applications, though. The standard does provide a mechanism for breaking rules in exceptional circumstances.

Some will start plenty of debate:

- Rule 4.2: Trigraphs shall not be used.

Though I'm totally with the MISRA folks on this one, more than a few folks won't give up their trigraphs till pried from their cold, dead hands.

- Rule 5.2: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Another reasonable rule that far too much code violates. The last thing we need are English-language-like homonyms obfuscating variable and function names.

- Rule 14.9: This rule is badly worded. So instead here's the original form of the rule from the 1998 version of the standard: The statements forming the body of an `if, else if, else, while, do … while` or `for` statement shall always be enclosed in braces.

The idea, which makes sense to me, is that we often make changes and additions to the statements following these constructs. If we always build a block structure such changes are easier and less likely to create problems.

Then there are the great ideas that might be impossible to adopt:

- Rule 3.6: All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

While a nice idea most of the popular compilers for embedded applications don't claim to offer run-time libraries that are MISRA-C compliant. What about protocol stacks and other third party tools?

- Rule 20.4: Dynamic heap memory allocation shall not be used.

This rule also specifically outlaws the use of library functions that dynamically allocate memory. Admittedly, `malloc()` and its buddies can wreak havoc on deterministic systems that never get cleaned up via a reboot. But it's a whole lot of work to recode libraries that employ these common functions … if you even know how the library routines work. What fun we might have stripping all of the `malloc()`s and `free()`s from embedded Linux! That would be a programmers' jobs creation program. Let's see if we can get it stashed in among the other 15,000 items of pork Congress awards to their constituents each year.

The 119-page MISRC-C document explains the rationale behind each of the rules. Even if you disagree with some, you'll find the arguments are neither capricious nor poorly thought out. It's worth reading just to make you think about the way you use C.

141 commandments sounds like a lot to remember, particularly in today's climate where so many seem unable to manage the 10 Moses brought down. A great deal, though, are so

commonsensical that they're already part of your daily practice. With some thought and practice it becomes second nature to comply with the entire batch.

A number of tools will automatically check source files for MISRA-C compliance. Examples include Programming Research's QA-C (http://www.programmingresearch.com/solutions/qac3.htm), Gimpel's Lint (http://www.gimpel.com/html/products.htm), and Parasoft's C++Test (http://www.parasoft.com/jsp/products/home.jsp?product=CppTest).

The MISRA team put years of work and big bucks into exploring and codifying these rules. It's just one part of the armor the auto industry wears to protect itself from company-busting bugs. For a lousy 10 quid all of us can get all the benefits with none of the work. Talk about a no-brainer!

### 3.4.2 Lint

Back when dinosaurs roamed the Earth most of our computer work was on punched card mainframes. Some wag at my school programmed the Fortran compiler to count error messages; if your program generated more than 50 compile-time errors it printed a big picture of Alfred E. Neuman with the caption "This man never worries. But from the look of your program, you should."

This bit of helpful advice embarrassed many till the university's administrators discovered that folks were submitting random card decks just to get the picture. Wasting computer time was a sin not easily forgiven, so the systems people were instructed to remove the compiler's funny but rude output. They, of course, simply made the picture hard to get, an early Easter egg that was a challenge to our cleverness.

How times have changed! Not only do we no longer feed punched cards to our PCs, but if only we got just 50 errors or warnings from a compilation of new code! The tool almost shouts that the code may be flawed. That assignment looks suspicious. Do you really want to use a pointer that way?

With deaf ears we turn away, link, and start debugging. Sure enough, some of these potential problems create symptoms that we dutifully chase down via debugging, the slowest possible way. Some of the flaws don't surface till the customer starts using the product.

Even more horrifying are the folks who disable warnings or always run the compiler with the minimum level of error checking. Sure, that reduces output, but it's rather like tossing those unread nastygrams from the IRS into the trash. Sooner or later you'll have to pay, and paying later always costs more.

Build a PC product and count on product lifecycles measured in microseconds. Embedded systems, though, seem to last forever. That factory controller might run for years or even decades before being replaced. Surely, someone, sometime, will have to enhance or fix the firmware. In 3 or 10 years, when resurrecting the code for an emergency patch, how will that future programmer respond to 300 warnings screaming by? He or she won't know if the system is supposed to compile so unhappily, or if it's something he or she did wrong when setting up the development system from old media whose documentation was lost.

Maintenance is a fact of life. If we're truly professional software engineers, we *must* design systems that can be maintained. Clean compiles and links are a crucial part of building applications that can be opened and modified.

Did you know that naval ships have their wiring exposed, hanging in trays from the overhead? Fact is, the electrical system needs routine and non-routine maintenance. If the designers buried the cables in inaccessible locations, the ship would work right out of the shipyard, but would be un-maintainable; junk, a total design failure.

*Working* is not the sole measure of design success, especially in firmware. Maintainability is just as important, and requires as much attention.

Beyond maintenance, when we don't observe warnings we risk developing the *habit* of ignoring them. Good habits form the veneer of civilization. Dining alone? You still probably use utensils rather than lapping it up canine-like. These habits means we don't even have to think about doing the right thing during dinner with that important date. The same goes for most human endeavors.

The old saying "the way to write beautiful code is to write beautiful code for twenty years" reflects the importance of developing and nurturing good habits. Once we get in the so-easy-to-acquire habit of ignoring warning messages, we lose a lot of the diagnostic power of the compiler.

Of course spurious warnings are annoying. Deal with it. If we spend 10 minutes going through the list and find just one that's suggestive of a real problem, we'll save hours of debugging.

We can and should develop habits that eliminate all or most spurious warnings. A vast number come from pushing the C standard too hard. Stick with plain vanilla ANSI C with no tricks and no implied castings that forces the compiler to make no assumptions. The code might look boring, but it's more portable and generally easier to maintain.

Did you know that the average chunk of code contains between 5% and 20% of errors before we start debugging? (see Charlie Mills's article). That's 500–2000 bugs in a little 10,000-line program. My informal data, acquired from talking to many, many developers but lacking a scientific base, suggests we typically spend about half of the project time debugging. So anything we can do to reduce bugs before starting debug pays off in huge ways.

We need a tool that creates *more* warnings, not fewer. A tool that looks over the code and finds the obvious and obscure constructs that might be a problem; that says "hey, better check this a little closer … it looks odd."

Such a tool does exist and has been around practically since the dawn of C. Lint (named for the bits of fluff it picks from programs) is like the compiler's syntax-checker on steroids. Lint works with a huge base of rules and points out structures that just seem weird. In my opinion, Lint is an essential part of any developer's toolbox, and is the first weapon against bugs. It will find problems much faster than debugging.

How is Lint different from your compiler's syntax checker? First, it has much stronger standards for language correctness than the compiler. For instance, most Lints track type definitions—as with typedef—and resolve possible type misuse as the ultimate types are resolved and used.

Lint, unlike a compiler's syntax checker, is more aware of a program's structure, so is better able to find possible infinite loops, and unused return values. Will your compiler flag these as problems?

```
b[i] = i++;
status & 2 == 0;
```

Lint will.

But much more powerfully, Lints can look at how multiple C files interact. Separate compilation is a wonderful tool for keeping information hidden, to reduce file size, and to keep local things local. But it means that the compiler's error checking is necessarily limited to just a single file. We do use function prototypes, for instance, to help the compiler spot erroneous use of external routines, but Lint goes much further. It can flag inconsistent definitions or usage across files, including libraries.

Especially in a large development project with many programmers, Lint is a quick way to find cross-file problems.

The downside to Lint, though, is that it can be very noisy. If you're used to ignoring a handful of warning messages from the compiler then Lint will drive you out of your mind. It's not unusual to get 30,000 messages from Linting a 1000-line module.

The trick is to train the product. Every Lint offers many different configuration options aimed to tune it to your particular needs. Success with Lint—as with any tool—requires a certain amount of your time. Up front, you'll lose productivity. There's a painful hump you'll have to overcome before gaining its benefits.

Commercial and free Lints abound; while all are similar they do differ in the details of what gets checked and how one goes about teaching the product to behave in a reasonable fashion.

Probably the most popular of all PC-hosted commercial Lints is the $239 version by Gimpel Software (www.gimpel.com). This product has a huge user base and is very stable. It's a small price for such fantastic diagnostic information … particularly in the embedded world where compilers may cost many thousands of dollars.

SpLint is a freebie package whose C source code is also available (http://lclint.cs.virginia .edu/). Linux, other UNIX, and PC distributions are all available.

### *3.4.3 Static Analysis*

Give me a program with two buttons: "find bug" and "fix bug." Though such alchemy is not possible today, we've long had syntax checkers, Lint, and other contraptions that do find some problems.

A new and evolving class of static analyzers by Polyspace, Green Hills, Coverity, and Klocwork do deep analysis of source code to find those hard-to-isolate problems.

Though I've yet to take these offerings for a test drive the claims made by the vendors are compelling.

Static analyzers look for execution-time problems without running your code. Complex path analysis and big rule sets find null pointer dereferences, memory leaks, buffer overflows and much more, just by analyzing the source tree. I've been told that, on average, the tools uncover about 1 bug per thousand lines of code analyzed. That might not seem like much … but since some defects might take days or weeks to find, the benefit is clear. That's 1000 bugs per megaline; think of the time required to find and fix a thousand bugs! And when a single bug can result in a huge product recall, this sort of insight has vast benefits.

At this writing the tools are in their infancy, but will surely become an important factor in producing quality code.

## 3.5 Encapsulation

If God didn't want us to use global variables, he wouldn't have invented them. Rather than disappoint God, use as many globals as possible.

This must have been the philosophy of the developer I know who wrote a program over 100K lines long that sported a nifty 5000 global variables. Five *thousand*. The effect: he was the only person in the universe who could maintain the code. Yet I heard constant complaints from him about being "stuck on the project's maintenance."

Then he quit.

Global variables are a scourge that plunges many systems into disaster. Globals are seductive; they leer at us as potential resources, crooning "just put one in here, how bad can it be?" Like a teenager mismanaging a credit card, that first careless use too often leads to another and another, until, as Thomas McGuane wrote, "The night wrote a check the morning couldn't cash."

But "globals" is a term that refers to much more than just variables. *Any* shared resource is a potential for disaster. That's why we all write device drivers to handle peripherals, layering a level of insulation between their real-world grittiness and the rest of our code. This is called encapsulation, and is a basic part of all OO design, but one can—and must—encapsulate in any language.

You do *religiously* use device drivers, don't you? I read a lot of C; it's astonishing how developers sprinkle thoughtless input/output instructions throughout the code like Johnny Appleseed tossing seeds into the wind.

### 3.5.1 The Problem

Globals break the important principle of information hiding. Anyone can completely comprehend a small system with 10 variables and a couple of hundred lines of code. Scale that by an order of magnitude or three and one soon gets swamped in managing implementation details. Is `user_input` a char or an int? It's defined in some header, somewhere. When thousands of variables are always in scope, it doesn't take much of a brain glitch or typo to enter `set_data` instead of `data_set`, which may refer to an entirely unrelated variable.

Next, globals can be unexpectedly stepped on by anyone: other developers, tasks, functions, and ISRs. Debugging is confounded by the sheer difficulty of tracking down the errant routine. Everyone is reading and writing that variable anyway; how can you isolate the one bad access out of a million … especially using the typically crummy breakpoint resources offered by most bit-wiggling BDMs?

Globals lead to strong coupling, a fundamental no-no in computer science. eXtreme Programming's belief that "everything changes all of the time" rings true. When a global's type, size, or meaning changes it's likely the software team will have to track down and change every reference to that variable. That's hardly the basis of highly productive development.

Multitasking systems and those handling interrupts suffer from severe reentrancy problems when globals pepper the code. An 8-bit processor might have to generate several instructions to do a simple 16-bit integer assignment. Inevitably an interrupt will occur between those instructions. If the ISR or another task then tries to read or set that variable, the Four Horsemen of the Apocalypse will ride through the door. Reentrancy problems aren't particularly reproducible so that once a week crash, which is quite impossible to track using most debugging tools, will keep you working plenty of late hours.

Or then there's the clever team member who thoughtlessly adds recursion to a routine which manipulates a global. If such recursion is needed, changing the variable to emulate a stack-based automatic may mean ripping up vast amounts of other code that shares the same global.

Globals destroy reuse. The close coupling between big parts of the code means everything is undocumentably interdependent. The software isn't a collection of packages; it's a web of ineffably interconnected parts.

Finally, globals are addictive. Late at night, tired, crashing from the uber-caffeinated drink consumed hours ago, we sneak one in. Yeah, that's poor design, but it's just this once. That lack of discipline cracks open the door of chaos. It's a bit easier next time to add yet another global; after all, the software already has this mess in it. Iterated, this dysfunctional behavior becomes habitual.

Why do we stop at a red light on a deserted street at 3 AM? The threat of a cop hiding behind a billboard is a deterrent, as is the ever-expanding network of red-light cameras. But breaking the rules leads to rule-breaking. Bad habits replace the good ones far too easily, so a mature driver carefully stops to avoid practicing dangerous behavior. In exceptional circumstances, of course (the kid is bleeding), we toss the rules and speed to the hospital.

The same holds true in software development. We don't use globals as a lazy alternative to good design. But in some exceptional conditions there's no alternative.

### *3.5.2 Alternatives to Globals*

Encapsulation is the anti-global pattern. Shared resources of all stripes should cower behind the protection of a driver. The resource—be it a global variable or an I/O device—is private to that driver.

A function like `void variable_set(int data)` sets the global (in this case for an int), and a corresponding `int variable_get()` reads the data. Clearly, in C at least, the global is still not really local to a single function; it's filescopic to both driver routines (more on this later) so both of these functions can access it.

The immediate benefit is that the variable is hidden. Only those two functions can read or change it. It's impossible to accidentally choose the wrong variable name while programming. Errant code can't stomp on the now non-global.

But there are some additional perks that come from this sort of encapsulation.

In embedded systems it's often impossible due to memory or CPU cycle constraints to range-check variables. When one is global then *every* access to that variable requires a check, which quickly burns ROM space and programmer patience. The result is, again, we form the habit of range-checking nothing. Have you seen the picture of the parking meter displaying a total due of 8.1 E+6 dollars? Or the electronic billboard showing a 505 degree temperature? Ariane 5 was lost, to the tune of several hundred million dollars, in part because of unchecked variables whose values were insane. I bet those developers wish they had checked the range of critical variables.

An encapsulated variable requires but a single test, one `if` statement, to toss an exception if the data is whacky. If CPU cycles are in short supply it might be possible to eliminate even that overhead with a compile-time switch that at least traps such errors at debug time.

Encapsulation yields another cool debugging trick. Use a `#define` to override the call to `variable_set(data)` as follows:

```
#define variable_set(data) variable_set_debug(data, __FILE__,
  __LINE__)
```

… and modify the driver routine to stuff the extra two parameters into a log file, circular buffer, or to a display. Or only save that data if there's an out-of-range error. This little bit of extra information tracks the error to its source.

Add code to the encapsulated driver to protect variables subject to reentrancy corruption. For instance:

```
int variable_get(void){
  int temp;
  push_interrupt_state;
  disable_interrupts;
  temp=variable;
  pop_interrupt_state;
  return temp;
}
```

Turning interrupts off locks the system down until the code extracts the no-longer-global variable from memory. Notice the code to push and pop the interrupt state; there's no guarantee that this routine won't be called with interrupts already disabled. The additional two lines preserve the system's context.

An RTOS offers better reentrancy-protection mechanisms like semaphores. If using Micrium's uC/OS-II, for instance, use the OS calls `OSSemPend` and `OSSemPost` to acquire and release semaphores. Other RTOSes have similar mechanisms.

The ex-global is not really private to a single function. Consider a more complicated example, like handling receive data from a UART, which requires three data structures and four functions:

- `UART_buffer`—a circular buffer which stores data from the UART
- `UART_start_ptr`—the pointer to the beginning of data in the circular buffer
- `UART_end_ptr`—pointer to the end of the buffer
- `UART_init()`—which sets up the device's hardware and initializes the data structures
- `UART_rd_isr()`—the ISR for incoming data
- `UART_char_avail()`—tests the buffer to see if a character is available
- `UART_get()`—retrieves a character from the buffer if one is available

One file—UART.C—contains these functions (though I'd also add the functions needed to send data to the device to create a complete UART handler) and nothing else. Define the filescopic data structures using the `static` keyword to keep them invisible outside the file. Only this small hunk of code has access to the data. Though this approach does create variables that are not encapsulated by functions, it incurs less overhead than a more rigorous adoption of encapsulation would and carries few perils. Once debugged, the rest of the system only sees the driver entry points so cannot muck with the data.

Note that the file that handles the UART is rather small. It's a package that can be reused in other applications.

### *3.5.3 Wrinkles*

Encapsulation isn't free. It consumes memory and CPU cycles. Terribly resource-constrained applications might not be able to use it at all.

Even in a system with plenty of headroom there's nothing faster for passing data around than globals. It's not always practical to eliminate them altogether. But their use (and worse, their abuse) does lead to less reliable code. My rule, embodied in my firmware standard, is "no global variables! But … if you really need one … get approval from the team leader." In other words, globals are a useful asset managed very, very carefully.

When a developer asks for permission to use a global, ask "Have you profiled the code? What makes you think you need those clock cycles? Give me solid technical justification for making this decision."

Sometimes it's truly painful to use encapsulation. I've seen people generate horribly contorted code to avoid globals. Use common sense; strive for a clean design that's maintainable.

Encapsulation has its own yin and yang. Debugging is harder. You're at a breakpoint with nearly all the evidence at hand needed to isolate a problem, except for the value of the encapsulated system status! What now?

It's not too hard to reference the link map, find the variable's address, and look at the hex. But it's hard to convert a floating point number's hex representation to human-speak. One alternative is to have a debug mode such that the encapsulated `variable_set()` function stores a copy, *which no other code accesses*, in a real global somewhere. Set this inside the driver's reentrancy-protection code so interrupt corruption is impossible.

### 3.5.4 The Other Side of the Story

I mentioned a program that had 5000 globals. When it came time to do a major port and clean-up of the code we made a rule: no globals. But this was a big real-time system pushing a lot of data around very fast; globals solved some difficult technical problems.

But at the end of the port there were only five global variables. That's a lot more tractable than derailing with 5000.

CHAPTER 4

# Real Time

## 4.1 Real Time Means Right Now

We're taught to think of our code in the procedural domain: that of actions and effects—IF statements and control loops that create a logical flow to implement algorithms and applications. There's a not-so-subtle bias in college toward viewing *correctness* as being nothing more than stringing the right statements together.

Yet embedded systems are the realm of real time, where getting the result on-time is just as important as computing the correct answer.

A hard real-time task or system is one where an activity simply must be completed—always—by a specified deadline. The deadline may be a particular time or time interval or may be the arrival of some event. Hard real-time tasks fail, by definition, if they miss such a deadline.

Notice this definition makes no assumptions about the frequency or period of the tasks. A microsecond or a week—if missing the deadline induces failure, then the task has hard real-time requirements.

"Soft" real time, though, has a definition as weak as its name. By convention it's the class of systems that are not hard real time, though generally there are some sort of timeliness requirements. If missing a deadline won't compromise the integrity of the system, if getting the output in a timely manner is acceptable, then the application's real-time requirements are "soft." Sometimes soft real-time systems are those where multi-valued timeliness is acceptable: bad, better, and best responses are all within the scope of possible system operation.

### 4.1.1 Interrupts

Most embedded systems use at least one or two interrupting devices. Few designers manage to get their product to market without suffering metaphorical scars from battling interrupt service routines (ISRs). For some incomprehensible reason—perhaps because "real time" gets little more than lip service in academia—most of us leave college without the slightest idea of how to design, code, and debug these most important parts of our systems. Too many of us become experts at ISRs the same way we picked up the secrets of the birds and the bees—from quick conversations in the halls and on the streets with our pals. There's got to be a better way!

New developers rail against interrupts because they are difficult to understand. However, just as we all somehow shattered our parents' nerves and learned to drive a stick shift, it just takes a bit of experience to become a certified "master of interrupts."

Before describing the "how," let's look at why interrupts are important and useful. Somehow peripherals have to tell the CPU that they require service. On a UART, perhaps a character arrived and is ready inside the device's buffer. Maybe a timer counted down and must let the processor know that an interval has elapsed.

Novice embedded programmers naturally lean toward polled communication. The code simply looks at each device from time to time, servicing the peripheral if needed. It's hard to think of a simpler scheme.

An interrupt-serviced device sends a signal to the processor's dedicated interrupt line. This causes the processor to screech to a stop and invoke the device's unique ISR, which takes care of the peripheral's needs. There's no question that setting up an ISR and associated control registers is a royal pain. Worse, the smallest mistake causes a major system crash that's hard to troubleshoot.

Why, then, not write polled code? The reasons are legion:

1. Polling consumes a lot of CPU horsepower. Whether the peripheral is ready for service or not, processor time—usually a lot of processor time—is spent endlessly asking "do you need service yet?"

2. Polled code is generally an unstructured mess. Nearly every loop and long complex calculation has a call to the polling routines so that a device's needs

never remain unserviced for long. ISRs, on the other hand, concentrate all of the code's involvement with each device into a single area.

3. Polling leads to highly variable latency. If the code is busy handling something else (just doing a floating point add on an 8-bit CPU might cost hundreds of microseconds) the device is ignored. Properly managed interrupts can result in predictable latencies of no more than a handful of microseconds.

Use an ISR pretty much anytime a device can asynchronously require service. I say "pretty much" because there are exceptions. As we'll see, interrupts impose their own sometimes unacceptable latencies and overhead. I did a tape interface once, assuming the processor was fast enough to handle each incoming byte via an interrupt. Nope. Only polling worked. In fact, tuning the five instruction polling loops' speeds ate up 3 weeks of development time.

#### 4.1.1.1 Vectoring

Though interrupt schemes vary widely from processor to processor, most modern chips use a variation of *vectoring*. Peripherals, both external to the chip and internal (like on-board timers), assert the CPU's interrupt input.

The processor generally completes the current instruction and stores the processor's state (current program counter and possibly flag register) on the stack. The entire rationale behind ISRs is to accept, service, and return from the interrupt all with no visible impact on the code. This is possible only if the hardware and software save the system's context before branching to the ISR.

It then *acknowledges* the interrupt, issuing a unique interrupt acknowledge cycle recognized by the interrupting hardware. During this cycle the device places an interrupt code on the data bus that tells the processor where to find the associated vector in memory.

Now the CPU interprets the vector and creates a pointer to the interrupt vector table, a set of ISR addresses stored in memory. It reads the address and branches to the ISR.

Once the ISR starts, you the programmer must preserve the CPU's context (such as saving registers, restoring them before exiting). The ISR does whatever it must, then returns with all registers intact to the normal program flow. The main line application never knows that the interrupt occurred.

Figures 4.1 and 4.2 show two views of how an ×86 processor handles an interrupt. When the interrupt request line goes high, the CPU completes the instruction it's executing (in this case at address 0100) and pushes the return address (two 16-bit words) and the contents of the flag register. The interrupt acknowledge cycle—wherein the CPU reads an interrupt number supplied by the peripheral—is unique as there's no read pulse. Instead, `intack` going low tells the system that this cycle is unique.

To create the address of the vector, ×86 processors multiply the interrupt number by four (left shifted two bits). A pair of 16-bit reads extracts the 32-bit ISR address.

Important points:

- The CPU chip's hardware, once it sees the interrupt request signal, does everything automatically, pushing the processor's state, reading the interrupt number, extracting a vector from memory, and starting the ISR.

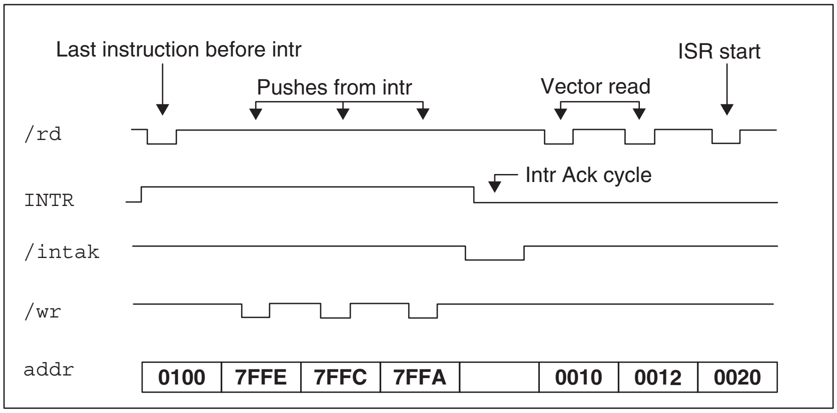


**Figure 4.1: Logic analyzer view of an interrupt**

```
0100    NOP  Fetch     <--- INTR REQ asserted
7FFE    0102 Write     <--- Return address pushed
7FFC    0000 Write
7FFA    ---- Write     <--- Flags pushed
xxxx    0010 INTA      <--- Vector inserted
0010    0020 Read      <--- ISR Address (low) read
0012    0000 Read      <--- ISR Address (high) read
0020    PUSH Fetch     <--- ISR starts
```

**Figure 4.2: Real-time trace view of an interrupt**

- The interrupt number supplied by the peripheral during the acknowledge cycle might be hardwired into the device's brain, but more commonly it's set up by the firmware. Forget to initialize the device and the system will crash as the device supplies a bogus number.
- Some peripherals and interrupt inputs will skip the acknowledge cycle since they have predetermined vector addresses.
- All CPUs let you disable interrupts via a specific instruction. Further, you can generally enable and disable interrupts from specific devices by appropriately setting bits in peripheral or interrupt control registers.
- Before invoking the ISR the hardware disables or reprioritizes interrupts. Unless your code explicitly reverses this, you'll see no more interrupts at this level.

At first glance the vectoring seems unnecessarily complicated. Its great advantage is support for many varied interrupt sources. Each device inserts a different vector; each vector invokes a different ISR. Your `USB_Data_Ready` ISR is called independently of the `USB_Transmit_Buffer_Full` routine. The vectoring scheme also limits pin counts, since it requires just one dedicated interrupt line.

Some CPUs sometimes directly invoke the ISR without vectoring. This greatly simplifies the code, but, unless you add a lot of manual processing it limits the number of interrupt sources a program can conveniently handle.

#### *4.1.1.2 Interrupt design guidelines*

While crummy code is just hard to debug, crummy ISRs are virtually undebuggable. The software community knows it's just as easy to write good code as it is to write bad. Give yourself a break and design hardware and software that ease the debugging process.

Poorly coded ISRs are the bane of our industry. Most ISRs are hastily thrown together, tuned at debug time to work, and tossed in the "oh my god it works" pile and forgotten. A few simple rules can alleviate many of the common problems.

First, don't even consider writing a line of code for your new embedded system until you lay out an interrupt map. List each interrupt, and give an English description of what the routine should do. Include your estimate of the interrupt's frequency. Figure the maximum, worst case time available to service each. This is your guide: exceed this number, and the system stands no chance of functioning properly.

The map is a budget. It gives you an assessment of where interrupting time will be spent. Just as your own personal financial budget has a degree of flexibility (spend too much on dinner this month and, assuming you don't abuse the credit cards, you'll have to reduce spending somewhere else). Like any budget, it's a condensed view of a profound reality whose parameters your system must meet. One number only is cast in stone: *there's only one second worth of compute time per second to get everything done.* You can tune execution time of any ISR, but be sure there's enough time overall to handle every device (Figure 4.3).

Approximate the complexity of each ISR. Given the interrupt rate, with some idea of how long it'll take to service each, you can assign priorities (assuming your hardware includes some sort of interrupt controller). Give the highest priority to things that must be done in staggeringly short times to satisfy the hardware or the system's mission (like, to accept data coming in from a 1 MB/s source).

The cardinal rule of ISRs is to keep the handlers short. A long ISR simply reduces the odds you'll be able to handle all time-critical events in a timely fashion. If the interrupt starts something truly complex, have the ISR spawn off a task that can run independently. This is an area where an RTOS is a real asset, as task management requires nothing more than a call from the application code.

Short, of course, is measured in *time*, not in code size. Avoid loops. Avoid long complex instructions (repeating moves, hideous math, and the like). Think like an optimizing compiler: does this code *really* need to be in the ISR? Can you move it out of the ISR into some less critical section of code?

For example, if an interrupt source maintains a time-of-day clock, simply accept the interrupt and increment a counter. Then return. Let some other chunk of code—perhaps a non-real-time task spawned from the ISR—worry about converting counts to time and day of the week.

| | Latency | Max-time | Freq | Description |
|---|---|---|---|---|
| INT1 | | 1000 µsec | 1000 µsec | timer |
| INT2 | | 100 µsec | 100 µsec | send data |
| INT3 | | 250 µsec | 250 µsec | Serial data in |
| INT4 | | 15 µsec | 100 µsec | write tape |
| NMI | 200 µsec | 500 µs | once! | System crash |

**Figure 4.3: An interrupt map**

Ditto for command processing. I see lots of systems where an ISR receives a stream of serial data, queues it to RAM, and then executes commands or otherwise processes the data. Bad idea! The ISR should simply queue the data. If time is really pressing (i.e., you need real-time response to the data), consider using another task or ISR, one driven via a timer that interrupts at the rate you consider "real time," to process the queued data.

An analogous rule to keeping ISRs short is to keep them simple. Complex ISRs lead to debugging nightmares, especially when the tools may be somewhat less than adequate. Debugging ISRs with a simple BDM-like debugger is going to hurt—bad. Keep the code so trivial there's little chance of error.

An old rule of software design is to use one function (in this case the serial ISR) to do one thing. A real-time analogy is to do things *only when they need to get done*, not at some arbitrary rate.

Re-enable interrupts as soon as practical in the ISR. Do the hardware-critical and non-reentrant things up front, then execute the interrupt enable instruction. Give other ISRs a fighting chance to do their thing.

Fill all of your unused interrupt vectors with a pointer to a null routine. During debug, *always* set a breakpoint on this routine. Any spurious interrupt, due to hardware problems or misprogrammed peripherals, will then stop the code cleanly and immediately, giving you a prayer of finding the problem in minutes instead of weeks (Figure 4.4).

```
Vect_table:
            dl    start_up          ; power up vector
            dl    null_isr          ; unused vector
            dl    null_isr          ; unused vector
            dl    timer_isr         ; main tick timer ISR
            dl    serial_in_isr     ; serial receive ISR
            dl    serial_out_isr    ; serial transmit ISR
            dl    null_isr          ; unused vector
            dl    null_isr          ; unused vector

null_isr:                           ; spurious intr routine
            jmp   null_isr          ; set BP here!
```

**Figure 4.4: Fill unused vectors with a pointer to `null_isr`, and set a breakpoint there while debugging**

### 4.1.1.3 Hardware issues

Lousy hardware design is just as deadly as crummy software. Modern high integration CPUs include a wealth of internal peripherals—USB controllers, timers, DMA controllers, etc. Interrupts from these sources pose no hardware design issues, since the chip vendors take care of this for you. All of these chips, though, do permit the use of external interrupt sources. There's trouble in them thar external interrupts!

The biggest issue is the generation of the INTR signal itself. Don't simply pulse an interrupt input. Though some chips do permit edge-triggered inputs, the vast majority of them require you to assert and hold INTR until the processor issues an acknowledgment, such as from the interrupt ACK pin. Sometimes it's a signal to drop the vector on the bus; sometimes it's nothing more than a "hey, I got the interrupt—you can release INTR now."

As always, be wary of timing. A slight slip in asserting the vector can make the chip wander to an erroneous address. If the INTR must be externally synchronized to clock, do *exactly* what the spec sheet demands.

If your system handles a really fast stream of data consider adding hardware to supplement the code. A data acquisition system I worked on accepted data at a 20 μsec rate. Each generated an interrupt, causing the code to stop what it was doing, vector to the ISR, push registers like wild, and then reverse the process at the end of the sequence. If the system was busy servicing another request, it could miss the interrupt altogether.

A cheap 256-byte-deep FIFO chip eliminated all of the speed issues. The hardware filled the FIFO without CPU intervention. It generated an interrupt at the half-full point (modern FIFOs often have Empty, Half-Full, and Full bits), at which time the ISR read data from the FIFO until it was sucked dry. During this process additional data might come along and be written to the FIFO, but this happened transparently to the code.

Most designs seem to connect FULL to the interrupt line. Conceptually simple, this results in the processor being interrupted only after the entire buffer is full. If a little extra latency causes a short delay before the CPU reads the FIFO, then an extra data byte arriving before the FIFO is read will be lost.

An alternative is EMPTY going not-true. A single byte arriving will cause the micro to read the FIFO. This has the advantage of keeping the FIFOs relatively empty, minimizing the chance of losing data. It also makes a big demand on CPU time, generating interrupts with practically every byte received.

Instead, connect HALF-FULL, if the signal exists on the FIFOs you've selected, to the interrupt line. HALF-FULL is a nice compromise, deferring processor cycles until a reasonable hunk of data is received, yet leaving free buffer space for more data during the ISR cycles.

Some processors do amazing things to service an interrupt, stacking addresses and vectoring indirectly all over memory. The ISR itself no doubt pushes lots of registers, perhaps also preserving other machine information. If the HALF-FULL line generates the interrupt, then you have the a priori information that lots of other data is already queued and waiting for processor time. Save overhead by making the ISR read the FIFOs until the EMPTY flag is set. You'll have to connect the EMPTY flag to a parallel port, so the software can read it, but the increase in performance is well worth it.

In mission-critical systems it might also make sense to design a simple circuit that latches the combination of FULL and an incoming new data item. This overflow condition could be disastrous and should be signaled to the processor.

A few bucks invested in a FIFO may allow you to use a much slower, and cheaper, CPU. Total system cost is the only price issue in embedded design. If a \$2 8-bit chip with a \$1 FIFO does the work of a \$5 32-bitter that also needs gobs of memory chips, it's foolish not to add the extra part.

Figure 4.5 shows the result of an Intel study of serial receive interrupts coming to a 386EX processor. At 530,000 baud—or around 53,000 characters per second—the CPU is almost completely loaded servicing interrupts.

Add a 16-byte FIFO and CPU loading declines to a mere 10%. That's a stunning performance improvement!

#### *4.1.1.4 C or assembly?*

If you've followed my suggestions you have a complete interrupt map with an estimated maximum execution time for the ISR. You're ready to start coding … right?

If the routine will be in assembly language, convert the time to a rough number of instructions. If an average instruction takes x microseconds (depending on clock rate, wait states, and the like), then it's easy to get this critical estimate of the code's allowable complexity.

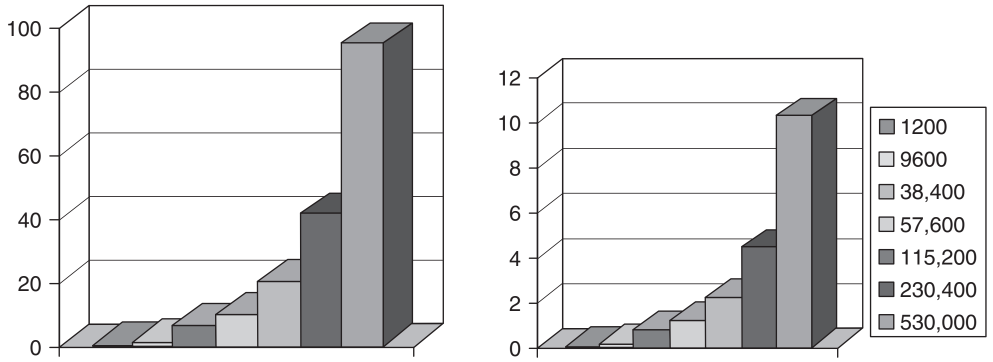


**Figure 4.5: Baud rates versus CPU utilization. On the left, a conventional connection uses 90% of the CPU to service 530K baud inputs. On the right, with a FIFO the processor is 10% loaded at the same rate**

C is more problematic. In fact, there's no way to scientifically write an interrupt handler in C! You have no idea how long a line of C will take. You can't even develop an estimate as each line's time varies wildly. A string compare may result in a run-time library call with totally unpredictable results. A FOR loop may require a few simple integer comparisons or a vast amount of processing overhead.

And so, we write our C functions in a fuzz of ignorance, having no concept of execution times until we actually run the code. If it's too slow, well, just change something and try again!

I'm not recommending not coding ISRs in C. Rather, this is more a rant against the current state of compiler technology. Years ago assemblers often produced T-state counts on the listing files, so you could easily figure how long a routine ran. Why don't compilers do the same for us? Though there are lots of variables (that string compare will take a varying amount of time depending on the data supplied to it), certainly many C operations will give deterministic results. It's time to create a feedback loop that tells us the cost, in time and bytes, for each line of code we write, before burning ROMs and starting test.

Until compilers improve, use C if possible, but look at the code generated for a typical routine. Any call to a run-time routine should be immediately suspect, as that routine may be slow or non-reentrant, two deadly sins for ISRs. Look at the processing overhead—how much pushing and popping takes place? Does the compiler spend a lot of time manipulating the stack frame? You may find one compiler pitifully slow at interrupt handling. Either try another or switch to assembly.

Despite all of the hype you'll read in magazine ads about how vendors understand the plight of the embedded developer, the plain truth is that the compiler vendors all uniformly miss the boat. Modern C and C++ compilers are poorly implemented in that they give us no feedback about the real-time nature of the code they're producing.

The way we write performance bound C code is truly astounding. Write some code, compile and run it … and if it's not fast enough, change something—anything—and try again. The compiler has so distanced us from the real-time nature of the code that we're left to make random changes in a desperate attempt to get the tool to produce faster code.

A much more reasonable approach would be to get listings from the compiler with typical per-statement execution times. An ideal listing might resemble:

```
250-275 nsec  for(i=0; i<count; ++i)
508-580 nsec   {if (start_count != end_count)
250 nsec          end_point=head;
              }
```

Where a range of values covers possible differences in execution paths depending on how the statement operates (for example, if the `for` statement iterates or terminates).

To get actual times, of course, the compiler needs to know a lot about our system, including clock rates and wait states. Another option is to display T-states, or even just number of instructions executed (since that would give us at least some sort of view of the code's performance in the time domain).

Vendors tell me that cache, pipelines, and prefetchers make modeling code performance too difficult. I disagree. Most small embedded CPUs don't have these features, and, of them, only cache is truly tough to model.

Please, Mr. Compiler Vendor, give us some sort of indication about the sort of performance we can expect! Give us a clue about how long a run-time routine or floating point operation takes.

A friend told me how his DOD project uses an antique language called CMS. The compiler is so buggy they have to look for bugs in the assembly listing after each and every compile—and then make a more or less random change and recompile, hoping to lure the tool into creating correct code. I laughed until I realized that's exactly the situation we're in when using a high quality C compiler in performance-bound applications.

Be especially wary of using complex data structures in ISRs. Watch what the compiler generates. You may gain an enormous amount of performance by sizing an array at an even power of two, perhaps wasting some memory, but avoiding the need for the compiler to generate complicated and slow indexing code.

An old software adage recommends coding for functionality first, and speed second. Since 80% of the speed problems are usually in 20% of the code, it makes sense to get the system working and then determine where the bottlenecks are. Unfortunately, real-time systems by their nature usually don't work at all if things are slow. You often *have* to code for speed up front.

If the interrupts are coming fast—a term that is purposely vague and qualitative, measured by experience and gut feel—then I usually just take the plunge and code the silly thing in assembly. Why cripple the entire system due to a little bit of interrupt code? If you have broken the ISRs into small chunks, so the real-time part is small, then little assembly will be needed. Code all of the slower ISRs in C.

### 4.1.2 Debugging INT/INTA Cycles

Lots of things can and will go wrong long before your ISR has a chance to exhibit buggy behavior. Remember that most processors service an interrupt with the following steps:

- The device hardware generates the interrupt pulse.
- The interrupt controller (if any) prioritizes multiple simultaneous requests and issues a single interrupt to the processor.
- The CPU responds with an interrupt acknowledge cycle.
- The controller drops an interrupt vector on the data bus.

- The CPU reads the vector and computes the address of the user-stored vector in memory. It then fetches this value.
- The CPU pushes the current context, disables interrupts, and jumps to the ISR.

Interrupts from internal peripherals (those on the CPU itself) will generally not generate an external interrupt acknowledge cycle. The vectoring is handled internally and invisibly to the wary programmer, tools in hand, trying to discover his system's faults.

A generation of structured programming advocates has caused many of us to completely design the system and write all of the code before debugging. Though this is certainly a nice goal, it's a mistake for the low level drivers in embedded systems. I believe in an early wrestling match with the system's hardware. Connect an emulator, and exercise the I/O ports. They never behave quite how you expected. Bits might be inverted or transposed, or maybe there's a dozen complex configuration registers needing setup. Work with your system, understand its quirks, and develop notes about how to drive each I/O device. Use these notes to write your code.

Similarly, start prototyping your interrupt handlers with a hollow shell of an ISR. You've got to get a lot of things *right* just to get the ISR to start. Don't worry about what the handler should do until you have it at least being called properly.

Set a breakpoint on the ISR. If your shell ISR never gets called, and the system doesn't crash and burn, most likely the interrupt never makes it to the CPU. If you were clever enough to fill the vector table's unused entries with pointers to a null routine, watch for a breakpoint on that function. You may have misprogrammed the table entry or the interrupt controller, which would then supply a wrong vector to the CPU.

If the program vectors to the wrong address, then use a logic analyzer or debugger's trace (if it has trace; a lot of BDM/JTAG units don't) to watch how the CPU services the interrupt. Trigger collection on the interrupt itself, or on any read from the vector table in RAM. You should see the interrupt controller drop a vector on the bus. Is it the right one? If not, perhaps the interrupt controller is misprogrammed.

Within a few instructions (if interrupts are on) look for the read from the vector table. Does it access the right table address? If not, and if the vector was correct, then you are either looking at the wrong system interrupt or there's a timing problem in the interrupt acknowledge cycle. Break out the logic analyzer, and check this carefully.

Hit the databooks and check the format of the table's entries. On an ×86-style processor, four bytes represent the ISR's offset and segment address. If these are in the wrong order—and they often are—there's no chance your ISR will execute.

Frustratingly often the vector is fine; the interrupt just does not occur. Depending on the processor and peripheral mix, only a handful of things could be wrong:

- Did you enable interrupts in the main routine? Without an EI instruction, no interrupt will ever occur. One way of detecting this is to sense the CPU's INTR input pin. If it's asserted all of the time, then generally the chip has all interrupts disabled.
- Does your I/O device generate an interrupt? It's easy to check this with external peripherals.
- Have you programmed the device to allow interrupt generation? Most CPUs with internal peripherals allow you to selectively disable each device's interrupt generation; quite often you can even disable parts of this (e.g., allow interrupts on "received data" but not on "data transmitted").

Modern peripherals are often incredibly complex. Motorola's TPU, for example, has an entire book dedicated to its use. Set one bit in one register to the wrong value, and it won't generate the interrupt you are looking for.

It's not uncommon to see an interrupt work perfectly once and then never work again. The only general advice is to be sure your ISR re-enables interrupts before returning. Then look into the details of your processor and peripherals.

You may need to service the peripherals as well before another interrupt comes along. Depending on the part, you may have to read registers in the peripheral to clear the interrupt condition. UARTs and Timers usually require this. Some have peculiar requirements for clearing the interrupt condition, so be sure to dig deeply into the databook.

### 4.1.3 Finding Missing Interrupts

A device that parses a stream of incoming characters will probably crash very obviously if the code misses an interrupt or two. One that counts interrupts from an encoder to measure position may only exhibit small precision errors, a tough thing to find and troubleshoot.

Having worked on a number of systems using encoders as position sensors, I've developed a few tricks over the years to find these missing pulses.

You can build a little circuit using a single up/down counter that counts every interrupt, and that decrements the count on each interrupt acknowledge. If the counter always shows a value of zero or one, everything is fine.

Most engineering labs have counters—test equipment that just accumulates pulse counts. I have a scope that includes a counter. Use two of these, one on the interrupt pin and another on the interrupt acknowledge pin. The counts should always be the same.

You can build a counter by instrumenting the ISR to increment a variable each time it starts. Either show this value on a display, or probe the variable using your debugger.

If you know the maximum interrupt rate, use a performance analyzer to measure the maximum time in the ISR. If this exceeds the fastest interrupts, there's very likely a latent problem waiting to pounce.

Most of these sorts of difficulties stem from slow ISRs or from code that leaves interrupts off for too long. Be wary of any code that executes a disable-interrupt instruction. There's rarely a good need for it; this is usually an indication of sloppy software.

It's rather difficult to find a chunk of code that leaves interrupts off. The ancient 8080 had a wonderful pin that showed interrupt state all of the time. It was easy to watch this on the scope and look for interrupts that came during that period. Now, having advanced so far, we have no such easy troubleshooting aids. About the best one can do is watch the INTR pin. If it stays asserted for long periods of time, and if it's properly designed (i.e., stays asserted until INTA), then interrupts are certainly off.

One design rule of thumb will help minimize missing interrupts: re-enable interrupts in ISRs at the earliest safe spot.

### *4.1.4 Avoid NMI*

Reserve NMI—the non-maskable interrupt—for a catastrophe like the apocalypse. Power-fail, system shutdown, and imminent disaster all good things to monitor with NMI. Timer or UART interrupts are not.

When I see an embedded system with the timer tied to NMI, I know, for sure, that the developers found themselves missing interrupts. NMI may alleviate the symptoms, but only masks deeper problems in the code that *must* be cured.

NMI will break even well-coded interrupt handlers, since most ISRs are non-reentrant during the first few lines of code where the hardware is serviced. NMI will thwart your stack management efforts as well.

If using NMI, watch out for electrical noise! NMI is usually an edge-triggered signal. Any bit of noise or glitching will cause perhaps hundreds of interrupts. Since it cannot be masked, you'll almost certainly cause a reentrancy problem. I make it a practice to always properly terminate the CPU's NMI input via an appropriate resistor network.

NMI mixes poorly with most tools. Debugging any ISR—NMI or otherwise—is exasperating at best. Few tools do well with single stepping and setting breakpoints inside of the ISR.

### 4.1.5 Breakpoint Problems

Using any sort of debugging tool, suppose you set a breakpoint where the ISR starts and then start single stepping through the code. All is well, since by definition interrupts are off when the routine starts. Soon, though, you'll step over an EI instruction or its like. Suddenly, all hell breaks lose.

A regularly occurring interrupt like a timer tick comes along steadily, perhaps dozens or hundreds of times per second. Debugging at human speeds means the ISR will start over while you're working on a previous instantiation. Pressing the "single step" button might make the ISR start, but then itself be interrupted and restarted, with the final stop due to your high-level debug command coming from this second incarnation.

Oddly, the code seems to execute backward. Consider the case of setting two breakpoints—the first at the start of the ISR and the second much later into the routine. Run to the first breakpoint, stop, and then resume execution. The code may very well stop at the same point, the same first breakpoint, without ever going to the second. Again, this is simply due to the human-speed debugging that gives interrupting hardware a chance to issue yet another request while the code's stopped at the breakpoint.

In the case of NMI, though, disaster strikes immediately since there is no interrupt-safe state. The NMI is free to reoccur at any time, even in the most critical non-reentrant parts of the code, wreaking havoc and despair.

A lot of applications now just can't survive the problems inherent in using breakpoints. After all, stopping the code stops everything; your entire system shuts down. If your code controls a moving robot arm, for example, and you stop the code as the arm starts moving, it will keep going and going and going … until something breaks or a limit switch is actuated. Years ago I worked on a 14-ton steel gauge; a Z80 controlled the motion of this monster on railroad tracks. Hit a breakpoint and the system ran off the end of the tracks!

Datacomm is another problem area. Stop the code via a breakpoint, with data packets still streaming in, and there's a good chance the receiving device will time out and start transmitting retry requests.

Though breakpoints are truly wonderful debugging aids, they are like Heisenberg's uncertainty principle: the act of looking at the system changes it. You can cheat Heisenberg—at least in debugging embedded code!—by using real-time trace, a feature available on all emulators and some smart logic analyzers.

### *4.1.6 Easy ISR Debugging*

What's the fastest way to debug an ISR?

Don't.

If your ISR is only 10 or 20 lines of code, debug by inspection. Don't fire up all kinds of complex and unpredictable tools.

Keep the handler simple and short. If it fails to operate correctly, a few minutes reading the code will usually uncover the problem.

After 35 years of building embedded systems I've learned that long ISRs are a bad thing and are a symptom of poor code. If something complex needs to happen, have the ISR spawn a task to do the work.

Keep 'em short, and keep 'em simple.

### *4.1.7 The RTOS*

Millions of words have been written about RTOSs so there's no need to reiterate them here. I will make a few short points that seem mostly neglected by other authors.

How does one decide whether or not to use an RTOS? Surveys show about 30% of projects don't have any sort of OS, which is entirely reasonable for a lot of embedded applications. An RTOS adds a number of costs. Commercial RTOSs require either a per-unit royalty payment or an up-front fee; sometimes both, and these can be significant. Even the smallest operating system bumps up memory needs, and many offerings are a long way from small. And the time required to learn the RTOS and its associated toolchain, and get it installed and working, can suck weeks from the schedule.

Yet once you're familiar with the operating system and it's installed and working, you'll reap some important benefits. The program will be more logically constructed since a polling loop doesn't have to be tortured into some awful shape where it can manage many different activities, some of which must run at particular intervals. It loosens system coupling, which eases maintenance. And sequencing activities becomes utterly trivial. A decent OS comes with powerful communications resources like semaphores, mailboxes, and queues that let you pass data around the system in a reentrant-safe manner.

Here are my rules of thumb for making the RTOS/no-RTOS decision:

- Any big system, where "big" is defined like pornography (that is, "I know it when I see it"), gets either an RTOS or, if appropriate, a conventional operating system like Linux. Big systems always get bigger, big systems are inherently complex, and big systems manage many different activities, so benefit from having a multitasking environment.
- Most small systems, where "small" means we're counting pennies, do not get an operating system of any sort.
- Where lives are at stake if the system is not lightly loaded we do a very careful analysis to see if multitasking could create determinism problems (see below). If determinism can't be guaranteed, and if there is no redundant safety mechanism, avoid an RTOS. Note, though, that moving tasks into interrupt handlers does not solve determinism. A different mechanism, such as time-triggered sequencing, is probably more appropriate.
- If a regulatory body must certify the product, then use an RTOS only if a certifiable version is available. For instance, at this writing several vendors sell commercial operating systems certifiable to DO-178B Level A, but such products are not available for all processors.

- If there are multiple independent activities going on, such as a need to update a display while also taking data and scanning a keyboard, I prefer to use an RTOS.
- If a simple loop can handle all of the application's needs, and we foresee no future need to add functionality beyond what the loop can manage, skip the RTOS.

Then there's the determinism issue, which few vendors care to discuss. Fact is, normal preemptive round-robin multitasking yields a system which no one can prove will work! It has been shown that if the perfect storm of interrupts and/or task scheduling requests occurs, tasks may fail to run as designed.

Add cache memory and the determinism problems skyrocket.

The problem can be mitigated, but not eliminated, by lowering CPU utilization, which adds timing margins.

A variety of alternative scheduling algorithms exist. Probably the best known is Rate Monotonic Scheduling, in which the task that runs most often gets the highest priority. Then figure:

$$x = \sum_{1}^{n} \frac{E_i}{P_i}$$

where

$E_i$ is the execution time of each task, and

$P_i$ is the period of each task

$n$ is the number of tasks

Then, if:

$$x \leq n(\sqrt[n]{2} - 1)$$

all tasks will be scheduled in a timely manner. Simplifying somewhat, for a typical busy system with more than a few tasks, if the sum of (worst-case execution time of each task)/(task's period) is less than about 69%, you'll meet all deadlines if tasks are assigned priorities based on how often they run. The fastest gets the highest priorities; those that run the least often get the lowest.

RMS has been around since the 1970s, and plenty of articles in *Embedded Systems Design* and elsewhere advocate its use. "Rate Monotonic Scheduling" gets over 8000 hits in Google. We're all using it … right?

Maybe not.

RMS scheduling absolutely requires you to know how often each task runs. Do you? That can be awfully hard to determine in a real-world application responding to unpredictable inputs from the real world.

To guarantee RMS will work in any particular application, you've got to insure the sum of execution time/task period is under about 69%. That implies we know each task's real-time behavior. Few of us do. In assembly, with simple processors devoid of cache and pipelines, it's tedious but not difficult to compute execution time. In C that's impossible. Even looking at the generated assembly sheds little light on the subject since calls to the run-time package are opaque at best.

One could instrument the code to measure execution times quantitatively. Change the code, though, and it's critical to repeat the measurements. Base a design on RMS and there are significant long-term maintenance issues where perhaps plenty of changes and enhancements can be anticipated.

## 4.2 Reentrancy

Virtually every embedded system uses interrupts; many support multitasking or multithreaded operations. These sorts of applications can expect the program's control flow to change contexts at just about any time. When that interrupt comes, the current operation is put on hold and another function or task starts running. What happens if functions and tasks share variables? Disaster surely looms if one routine corrupts the other's data.

By carefully controlling how data is shared, we create "reentrant" functions, those that allow multiple concurrent invocations that do not interfere with each other. The word "pure" is sometimes used interchangeably with "reentrant."

Reentrancy was originally invented for mainframes, in the days when memory was a valuable commodity. System operators noticed that a dozen to hundreds of identical copies of a few big programs would be in the computer's memory array at any time. At the University of Maryland, my old hacking grounds, the monster Univac 1108 had one of the early reentrant FORTRAN compilers. It burned up a (for those days) breathtaking 32 kW of system memory, but being reentrant, it required only 32K even if 50 users were running it. Everyone executed the same code, from the same set of addresses. Each person had his or her own data area, yet everyone running the compiler quite literally executed identical code. As the operating system changed contexts from user to user it swapped data areas so one person's work didn't affect any other. Share the code, but not the data.

In the embedded world a routine must satisfy the following conditions to be reentrant:

1. It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
2. It does not call non-reentrant functions.
3. It does not use the hardware in a non-atomic way.

### 4.2.1 Atomic Variables

Both the first and last rules use the word "atomic," which comes from the Greek word meaning "indivisible." In the computer world "atomic" means an operation that cannot be interrupted. Consider the assembly language instruction:

```
mov    ax,bx
```

Since nothing short of a reset can stop or interrupt this instruction, it's atomic. It will start and complete without any interference from other tasks or interrupts.

The first part of rule 1 requires the atomic use of shared variables. Suppose two functions each share the global variable "foobar." Function A contains:

```
temp=foobar;
temp+=1;
foobar=temp;
```

This code is not reentrant, because foobar is used non-atomically. That is, it takes three statements to change its value, not one. The foobar handling is not indivisible; an interrupt can come between these statements and switch context to the other function, which then may also try to change foobar. Clearly there's a conflict; foobar will wind up with an incorrect value, the autopilot will crash, and hundreds of screaming people will wonder “why didn’t they teach those developers about reentrancy?”

Suppose, instead, Function A looks like:

```
foobar+=1;
```

Now the operation is atomic; an interrupt will not suspend processing with foobar in a partially changed state, so the routine is reentrant.

Except … do you really know what your C compiler generates? On an ×86 processor the code might look like:

```
mov  ax,[foobar]
inc  ax
mov  [foobar],ax
```

which is clearly not atomic, and so not reentrant. The atomic version is:

```
inc    [foobar]
```

The moral is to be wary of the compiler; assume it generates atomic code and you may find *60 Minutes* knocking at your door.

But even the seemingly safe `inc [foobar]` will exhibit non-reentrant behavior in some multiprocessor systems which share a memory bus. The increment instruction actually does three things in sequence:

1. Read from variable `foobar` into an internal CPU register
2. Increment the register
3. Write the register back to `foobar`

If one processor reads the variable (step 1), and then another acquires the bus and also accesses `foobar`, the two processors will be working with different values for the same variable! The solution is to insure your compiler locks bus accesses, as follows:

```
lock inc [foobar]
```

… which dedicates the bus to one processor for the entire execution of the instruction.

The second part of the first reentrancy rule reads "… unless each is allocated to a specific instance of the function." This is an exception to the atomic rule that skirts the issue of shared variables.

An "instance" is a path through the code. There's no reason a single function can't be called from many other places. In a multitasking environment it's quite possible that several copies of the function may indeed be executing concurrently. (Suppose the routine is a driver that retrieves data from a queue; many different parts of the code may want queued data more or less simultaneously). Each execution path is an "instance" of the code.

Consider:

```
int foo;
void some_function(void){
foo++;
}
```

`foo` is a global variable whose scope exists beyond that of the function. Even if no other routine uses `foo`, `some_function` can trash the variable in more than one instance if it runs at any time.

C and C++ can save us from this peril. Use automatic variables. That is, declare `foo` inside of the function. Then, each instance of the routine will use a new version of `foo` created from the stack, as follows:

```
void some_function(void){
int foo;
foo++;
}
```

Another option is to dynamically assign memory (using malloc), again so each incarnation uses a unique data area. The fundamental reentrancy problem is thus avoided, as it's impossible for multiple instances to stamp on a common version of the variable.

### *4.2.2 Two More Rules*

The rest of the rules are very simple.

Rule 2 tells us a calling function inherits the reentrancy problems of the callee. That makes sense; if other code inside the function trashes shared variables, the system is going to crash. Using a compiled language, though, there's an insidious problem. Are you sure—really sure—that the run-time package is reentrant? Obviously string operations and a lot of other complicated things use run-time calls to do the real work. An awful lot of compilers also generate run-time calls to do, for instance, long math, or even integer multiplications and divisions.

If a function must be reentrant, talk to the compiler vendor to insure that the entire run-time package is pure. If you buy software packages (like a protocol stack) that may be called from several places, take similar precautions to insure the purchased routines are also reentrant.

Rule 3 is a uniquely embedded caveat. Hardware looks a lot like a variable; if it takes more than a single I/O operation to handle a device, reentrancy problems can develop.

Consider Zilog's SCC serial controller. Accessing any of the device's internal registers requires two steps: first write the register's address to a port, then read or write the register from the same port, the same I/O address. If an interrupt comes between setting the port and accessing the register, another function might take over and access the device. When control returns to the first function the register address you set will be incorrect.

## *4.2.3 Keeping Code Reentrant*

What are our best options for eliminating non-reentrant code? The first rule of thumb is to avoid shared variables. Globals are the source of no end of debugging woes and failed code. Use automatic variables or dynamically allocated memory.

Yet globals are also the fastest way to pass data around. It's not entirely possible to eliminate them from real-time systems. So, when using a shared resource (variable or hardware) we must take a different sort of action.

The most common approach is to disable interrupts during non-reentrant code. With interrupts off, the system suddenly becomes a single-process environment. There will be no context switches. Disable interrupts, do the non-reentrant work, and then turn interrupts back on.

Most times this sort of code looks like:

```
long i;
void do_something(void){
  disable_interrupts();
  i+=0×1234;
  enable_interrupts();
}
```

This solution *does not work*. If `do_something()` is a generic routine, perhaps called from many places, and is invoked with interrupts disabled, it returns after turning them back on. The machine's context is changed, probably in a very dangerous manner.

Don't use the old excuse "yeah, but I wrote the code and I'm careful. I'll call the routine only when I know that interrupts will be on." A future programmer probably does not know about this restriction and may see `do_something()` as just the ticket needed to solve some other problem … perhaps when interrupts are off.

Better code looks like:

```
long i;
void do_something(void){
  push interrupt state;
  disable_interrupts();
  i+=0×1234;
  pop interrupt state;
}
```

Shutting interrupts down does increase system latency, reducing its ability to respond to external events in a timely manner. A kinder, gentler approach is to use a semaphore to indicate when a resource is busy. Semaphores are simple on-off state indicators whose processing is inherently atomic, often used as "in-use" flags to have routines idle when a shared resource is not available.

Nearly every commercial real-time operating system includes semaphores; if this is your way of achieving reentrant code, by all means use an RTOS.

Don’t have an RTOS? Sometimes I see code that assigns an in-use flag to protect a shared resource, like this:

```
while (in_use);  //wait till resource free
in_use=TRUE;     //set resource busy
Do non-reentrant stuff
in_use=FALSE;    //set resource available
```

If some other routine has access to the resource it sets `in_use` true, causing this routine to idle until `in_use` gets released. Seems elegant and simple … but it does not work. An interrupt that occurs after the `while` statement will preempt execution. This routine feels it now has exclusive access to the resource, yet hasn’t had a chance to set `in_use` true. Some other routine can now get access to the resource.

Some processors have a test-and-set instruction, which acts like the in-use flag, but which is interrupt-safe. It’ll always work. The instruction looks something like:

```
Tset variable   ;  if(variable==0){
                ;     variable=1;
                ;     returns TRUE;}
                ;  else{returns FALSE;}
```

If you’re not so lucky to have a test-and-set, try the following:

```
loop:     mov  al,0     ;0 means "in use"
          xchg al,variable
          cmp  al,0
          je   loop     ;loop if in use
```

If `al=0`, we swapped `0` with zero; nothing changed, but the code loops since someone else is using the resource. If `al=1`, we put a `0` into the “`in use`” variable, marking the resource as busy. We fall out of the loop, now having control of the resource. It’ll work every time.

### *4.2.4 Recursion*

No discussion of reentrancy is complete without mentioning recursion, if only because there’s so much confusion between the two.

A function is recursive if it calls itself. That’s a classic way to remove iteration from many sorts of algorithms. Given enough stack space this is a perfectly valid—though tough to debug—way to write code. Since a recursive function calls itself, clearly it must be reentrant to avoid trashing its variables. So all recursive functions must be reentrant … but not all reentrant functions are recursive.

### *4.2.5 Asynchronous Hardware/Firmware*

But there are subtler issues that result from the interaction of hardware and software. These may not meet the classical definition of reentrancy but pose similar risks and require similar solutions.

We work at that fuzzy interface between hardware and software, which creates additional problems due to the interactions of our code and the device. Some cause erratic and quite impossible-to-diagnose crashes that infuriate our customers. The worst bugs of all are those that appear infrequently, that can't be reproduced. Yet a reliable system just cannot tolerate any sort of defect, especially the random one that passes our tests, perhaps dismissed with the "ah, it's just a glitch" behavior.

Potential evil lurks whenever hardware and software interact asynchronously. That is, when some physical device runs at its own rate, sampled by firmware running at some different speed.

I was poking through some open-source code and came across a typical example of asynchronous interactions. The RTEMS real-time operating system provided by OAR Corporation (ftp://ftp.oarcorp.com/pub/rtems/releases/4.5.0/ for the code) is a nicely written, well-organized product with a lot of neat features. But the timer handling routines, at least for the 68302 distribution, is flawed in a way that will fail infrequently but possibly catastrophically. This is just one very public example of the problem I constantly see buried in proprietary firmware.

The code is simple and straightforward, and looks much like any other timer handler.

```
int timer_hi;
interrupt timer(){
  ++timer_hi;}

long timer_read(void){
  unsigned int low, high;
  low =inword(hardware_register);
  high=timer_hi;
  return (high<<16+low);}
```

There's an ISR invoked when the 16-bit hardware timer overflows. The ISR services the hardware, increments a global variable named `timer_hi`, and returns. So `timer_hi` maintains the number of times the hardware counted to 65536.

Function `timer_read` returns the current "time" (the elapsed time in microseconds as tracked by the ISR and the hardware timer). It, too, is delightfully free of complications. Like most of these sorts of routines it reads the current contents of the hardware's timer register, shifts `timer_hi` left 16 bits, and adds in the value read from the timer. That is, the current time is the concatenation of the timer's current value and the number of overflows.

Suppose the hardware rolled over five times, creating five interrupts. `timer_hi` equals 5. Perhaps the internal register is, when we call `timer_read`, 0×1000. The routine returns a value of 0×51,000; simple enough and seemingly devoid of problems.

## *4.2.6 Race Conditions*

But let's think about this more carefully. There are really two things going on at the same time. *Not* concurrently, which means "apparently at the same time," as in a multitasking environment where the RTOS doles out CPU resources so all tasks *appear* to be running simultaneously. No, in this case the code in `timer_read` executes whenever called, and the clock-counting timer runs at its own rate. The two are asynchronous.

A fundamental rule of hardware design is to panic whenever asynchronous events suddenly synchronize. For instance, when two different processors share a memory array there's quite a bit of convoluted logic required to insure that only one gets access at any time. If the CPUs use different clocks the problem is much trickier, since the designer may find the two requesting exclusive memory access within fractions of a nanosecond of each other. This is called a "race" condition and is the source of many gray hairs and dramatic failures.

One of `timer_read`'s race conditions might be:

- It reads the hardware and gets, let's say, a value of `0Xffff`.
- Before having a chance to retrieve the high part of the time from variable `timer_hi`, the hardware increments again to 0×0000.

- The overflow triggers an interrupt. The ISR runs. `timer_hi` is now 0×0001, not 0 as it was just nanoseconds before.
- The ISR returns; our fearless `timer_read` routine, with no idea an interrupt occurred, blithely concatenates the new 0×0001 with the previously read timer value of `0Xffff`, and returns `0X1ffff`—a hugely incorrect value.

Or, suppose `timer_read` is called during a time when interrupts are disabled—say, if some other ISR needs the time. One of the few perils of writing encapsulated code and drivers is that you're never quite sure what state the system is in when the routine gets called. In this case:

- `Timer_read` starts. The timer is `0xffff` with no overflows.
- Before much else happens it counts to 0×0000. With interrupts off the pending interrupt gets deferred.
- `Timer_read` returns a value of 0×0000 instead of the correct 0×10000, or the reasonable `0Xffff`.

So the algorithm that seemed so simple has quite subtle problems, necessitating a more sophisticated approach. The RTEMS RTOS, at least in its 68k distribution, will likely create infrequent but serious errors.

Sure, the odds of getting a mis-read are small. In fact, the chance of getting an error plummets as the frequency we call `timer_read` decreases. How often will the race condition surface? Once a week? Monthly?

Many embedded systems run for years without rebooting. Reliable products must *never* contain fragile code. Our challenge as designers of robust systems is to identify these sorts of issues and create alternative solutions that work correctly, everytime.

### 4.2.7 Options

Fortunately a number of solutions do exist. The easiest is to stop the timer before attempting to read it. There will be no chance of an overflow putting the upper and lower halves of the data out of sync. This is a simple and guaranteed solution.

We will lose time. Since the hardware generally counts the processor's clock, or clock divided by a small number, it may lose quite a few ticks during the handful of instructions executed to do the reads. The problem will be much worse if an interrupt causes a context switch after disabling the counting. Turning interrupts off during this period will eliminate unwanted tasking, but increases both system latency and complexity.

I just *hate* disabling interrupts; system latency goes up and sometimes the debugging tools get a bit funky. When reading code a red flag goes up if I see a lot of disable interrupt instructions sprinkled about. Though not necessarily bad, it's often a sign that either the code was beaten into submission (made to work by heroic debugging instead of careful design) or there's something quite difficult and odd about the environment.

Another solution is to read the `timer_hi` variable, then the hardware timer, and then reread `timer_hi`. An interrupt occurred if both variable values aren't identical. Iterate until the two variable reads are equal. The upside: correct data, interrupts stay on, and the system doesn't lose counts.

The downside: in a heavily loaded, multitasking environment, it's possible that the routine could loop for rather a long time before getting two identical reads. The function's execution time is non-deterministic. We've gone from a very simple timer reader to somewhat more complex code that could run for milliseconds instead of microseconds.

Another alternative might be to simply disable interrupts around the reads. This will prevent the ISR from gaining control and changing `timer_hi` after we've already read it, but creates another issue.

We enter `timer_read` and immediately shut down interrupts. Suppose the hardware timer is at our notoriously problematic 0×ffff, and `timer_hi` is zero. Now, before the code has a chance to do anything else, the overflow occurs. With context switching shut down we miss the rollover. The code reads a zero from both the timer register and from `timer_hi`, returning zero instead of the correct 0×10000, or even a reasonable 0×0ffff.

Yet disabling interrupts is probably indeed a good thing to do, despite my rant against this practice. With them on there's always the chance our reading routine will be suspended by higher priority tasks and other ISRs for perhaps a very long time. Maybe long enough for the timer to roll over several times. So let's try to fix the code. Consider the following:

```
long timer_read(void){
  unsigned int low, high;
  push_interrupt_state;
  disable_interrupts;
  low=inword(Timer_register);
  high=timer_hi;
  if(inword(timer_overflow))
    {++high;
     low=inword(timer_register);}
  pop_interrupt_state;
  return (((ulong)high)<<16+(ulong)low);
}
```

We've made three changes to the RTEMS code. First, interrupts are off, as described.

Second, you'll note that there's no explicit interrupt re-enable. Two new pseudo-C statements have appeared which push and pop the interrupt state. Trust me for a moment—this is just a more sophisticated way to manage the state of system interrupts.

The third change is a new test that looks at something called "`timer_overflow`", an input port that is part of the hardware. Most timers have a testable bit that signals an overflow took place. We check this to see if an overflow occurred between turning interrupts off and reading the low part of the time from the device. With an inactive ISR variable `timer_hi` won't properly reflect such an overflow.

We test the status bit and reread the hardware count if an overflow had happened. Manually incrementing the high part corrects for the suspended ISR. The code then concatenates the two fixed values and returns the correct result. Every time.

With interrupts off we have increased latency. However, there are no loops; the code's execution time is entirely deterministic.

### *4.2.8 Other RTOSs*

Unhappily, race conditions occur anytime we need more than one read to access data that's changing asynchronously to the software. If you're reading X and Y coordinates, even with just 8 bits of resolution, from a moving machine there's some peril they could be seriously out of sync if two reads are required. A 10-bit encoder managed through byte-wide ports potentially could create a similar risk.

Having dealt with this problem in a number of embedded systems over the years, I wasn't too shocked to see it in the RTEMS RTOS. It's a pretty obscure issue, after all, though terribly real and potentially deadly. For fun I looked through the source of uC/OS, another very popular operating system whose source is on the net (see www.ucos-ii.com). uC/OS never reads the timer's hardware. It only counts overflows as detected by the ISR, as there's no need for higher resolution. There's no chance of an incorrect value.

Some of you, particularly those with hardware backgrounds, may be clucking over an obvious solution I've yet to mention. Add an input capture register between the timer and the system; the code sets a "lock the value into the latch" bit, then reads this safely unchanging data. The register is nothing more than a parallel latch, as wide as the input data. A single clock line drives each flip-flop in the latch; when strobed it locks the data into the register. The output is fed to a pair of processor input ports.

When it's time to read a safe, unchanging value the code issues a "hold the data now" command which strobes encoder values into the latch. So all bits are stored and can be read by the software at any time, with no fear of things changing between reads.

Some designers tie the register's clock input to one of the port control lines. The I/O read instruction then automatically strobes data into the latch, assuming one is wise enough to insure the register latches data on the leading edge of the clock.

The input capture register is a very simple way to suspend moving data during the duration of a couple of reads. At first glance it seems perfectly safe. But a bit of analysis shows that for asynchronous inputs it *is not reliable*. We're using hardware to fix a software problem, so we must be aware of the limitations of physical logic devices.

To simplify things for a minute, let's zoom in on that input capture register and examine just one of its bits. Each gets stored in a flip-flop, a bit of logic that might have only three connections: data in, data out, and clock. When the input is a one, strobing clock puts a one at the output.

But suppose the input changes at about the same time clock cycles? What happens? The short answer is that no one knows.

### 4.2.9 Metastable States

Every flip-flop has two critical specifications we violate at our peril. "Set-up time" is the minimum number of nanoseconds that input data must be stable *before* clock comes. "Hold time" tells us how long to keep the data present *after* clock transitions. These specs vary depending on the logic device. Some might require a few nanoseconds of set-up and/or hold time; others need an order of magnitude less (Figure 4.6).

If we tend to our knitting we'll respect these parameters and the flip-flop will always be totally predictable. But when things are asynchronous—say, the wrist rotates at its own rate and the software does a read whenever it needs data—there's a chance that we'll violate set-up or hold time.

Suppose the flip-flop requires 3 ns of set-up time. Our data changes within that window, flipping state perhaps a single nanosecond before clock transitions. The device will go into a metastable state where the output gets very strange indeed.

By violating the spec the device really doesn't know if we presented a zero or a one. Its output goes, not to a logic state, but to either a half-level (in between the digital norms) or it will oscillate, toggling wildly between states. The flip-flop is metastable (Figure 4.7).

This craziness doesn't last long; typically after a few to 50 ns the oscillations damp out or the half-state disappears, leaving the output at a valid one or zero. But which one is it? This is a digital system, and we expect ones to be ones, and zeros zeros.

The output is *random*. Bummer, that. You cannot predict which level it will assume. That sure makes it hard to design predictable digital systems!

Hardware folks feel that the random output isn't a problem. Since the input changed at almost exactly the same time the clock strobed, either a zero or a one is reasonable.

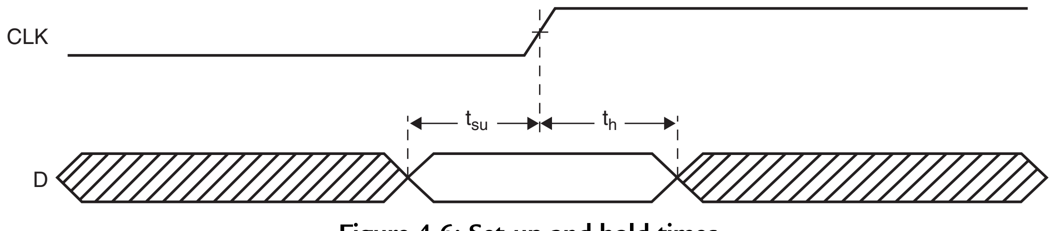


**Figure 4.6: Set-up and hold times**

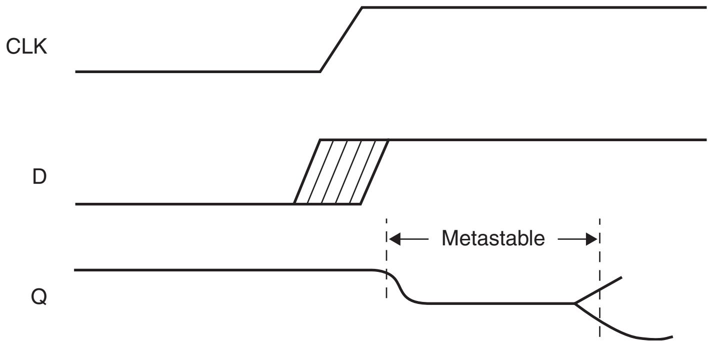


**Figure 4.7: A metastable state**

If we had clocked just a hair ahead or behind we'd have gotten a different value, anyway. Philosophically, who knows which state we measured? Is this really a big deal? Maybe not to the EEs, but this impacts our software in a big way, as we'll see shortly.

Metastability occurs only when clock and data arrive almost simultaneously; the odds increase as clock rates soar. An equally important factor is the type of logic component used; slower logic (like 74HCxx) has a much wider metastable window than faster devices (say, 74FCTxx). Clearly at reasonable rates the odds of the two asynchronous signals arriving closely enough in time to cause a metastable situation are low—measurable, yes; important, certainly. With a 10 MHz clock and 10 KHz data rate, using typical but not terribly speedy logic, metastable errors occur about once a minute. Though infrequent, no reliable system can stand that failure rate.

The classic metastable fix uses two flip-flops connected in series. Data goes to the first; its output feeds the data input of the second. Both use the same clock input. The second flop's output will be "correct" after two clocks, since the odds of two metastable events occurring back-to-back are almost nil. With two flip-flops, at reasonable data rates, errors occur millions or even billions of years apart. Good enough for most systems.

But "correct" means the second stage's output will not be metastable: it's not oscillating, nor is it at an illegal voltage level. There's still an equal chance the value will be in either legal logic state.

### 4.2.10 Firmware, Not Hardware

To my knowledge there's no literature about how metastability affects software, yet it poses very real threats to building a reliable system.

Hardware designers smugly cure their metastability problem using the two stage flops described. Their domain is that of a single bit, whose input changed just about the same time as the clock transition. Thinking in such narrow terms it's indeed reasonable to accept the inherent random output the flops generate.

But we software folks are reading parallel I/O ports, each perhaps 8 bits wide. That means there are 8 flip-flops in the input capture register, all driven by the same clock pulse.

Let's look at what might happen. The encoder changes from `0Xff` to 0×100. This small difference might represent just a tiny change in angle. We request a read at just about the same time the data changes; our input operation strobes the capture register's clock creating a violation of set-up or hold time. Every input bit changes; each of the flip-flops inside the register goes metastable. After a short time the oscillations die out, but now every bit in the register is random. Though the hardware folks might shrug and complain that no one knows what the right value was, since everything changed as clock arrived, in fact the data was around `0Xff` or 0×100. A random result of, say, 0×12 is absurd and totally unacceptable and may lead to crazy system behavior.

The case where data goes from `0Xff` to 0×100 is pathological since every bit changes at once. The system faces the same peril whenever lots of bits change. `0X0f` to 0×10. `0X1f` to 0×20. The upper, unchanging data bits will always latch correctly; but every changing bit is at risk.

Why not use the multiple flip-flop solution? Connect two input capture registers in series, both driven by the same clock. Though this will eliminate the illegal logic states and oscillations, the second stage's output will be random as well.

One option is to ignore metastability and hope for the best. Or use very fast logic with very narrow set-up/hold time windows to reduce the odds of failure. If the code samples in the inputs infrequently it's possible to reduce metastability to one chance in millions or even billions. Building a safety critical system? Feeling lucky?

It is possible to build a synchronizer circuit that takes a request for a read from the processor, combines it with a data available bit from the I/O device, responding with a data-OK signal back to the CPU. This is non-trivial and prone to errors.

An alternative is to use a different coding scheme for the I/O device. Buy an encoder with Gray Code output, for example (if you can find one). Gray Code is a counting scheme where only a single bit changes between numbers, as follows:

| | |
|---|---|
| 0 | 000 |
| 0 | 001 |
| 2 | 011 |
| 3 | 010 |
| 4 | 110 |
| 5 | 111 |
| 6 | 101 |
| 7 | 100 |

Gray Code makes sense if, and only if, your code reads the device faster than it's likely to change, and if the changes happen in a fairly predictable fashion—like counting up. Then there's no real chance of more than a single bit changing between reads; if the inputs go metastable only one bit will be wrong. The result will still be reasonable.

Another solution is to compute a parity or checksum of the input data before the capture register. Latch that, as well, into the register. Have the code compute parity and compare it to that read; if there's an error do another read.

Though I've discussed adding an input capture register, please don't think that this is the root cause of the problem. Without that register—if you just feed the asynchronous inputs directly into the CPU—it's quite possible to violate the processor's innate set-up/hold times. There's no free lunch; all logic has physical constraints we must honor.

Some designs will never have a metastability problem. It always stems from violating set-up or hold times, which in turn comes from either poor design or asynchronous inputs.

All of this discussion has revolved around asynchronous inputs, when the clock and data are unrelated in time. Be wary of anything not slaved to the processor's clock. Interrupts are a notorious source of problems. If caused by, say, someone pressing a button, be sure that the interrupt itself and the vector-generating logic don't violate the processor's set-up and hold times.

But in computer systems most things do happen synchronously. If you're reading a timer that operates from the CPU's clock, it is inherently synchronous to the code. From a metastability standpoint it's totally safe.

Bad design, though, can plague any electronic system. Every logic component takes time to propagate data; when a signal traverses many devices the delays can add up significantly. If the data then goes to a latch it's quite possible that the delays may cause the input to transition at the same time as the clock. Instant metastability.

Designers are pretty careful to avoid these situations, though. Do be wary of FPGAs and other components where the delays vary depending on how the software routes the device. And when latching data or clocking a counter it's not hard to create a metastability problem by using the wrong clock edge. Pick the edge that gives the device time to settle before it's read.

What about analog inputs? Connect a 12-bit A/D converter to two 8-bit ports and we'd seem to have a similar problem: the analog data can wiggle all over, changing during the time we read the two ports. However, there's no need for an input capture register because the converter itself generally includes a "sample and hold" block, which stores the analog signal while the A/D digitizes. Most A/Ds then store the digital value until we start the next conversion.

Other sorts of inputs we use all share this problem. Suppose a robot uses a 10-bit encoder to monitor the angular location of a wrist joint. As the wrist rotates the encoder sends back a binary code, 10 bits wide, representing the joint's current position. An 8-bit processor requires two distinct I/O instructions—two byte-wide reads—to get the data. No matter how fast the computer might be there's a finite time between the reads during which the encoder data may change.

The wrist is rotating. A `"position_read()"` routine reads 0xff from the low part of the position data. Then, before the next instruction, the encoder rolls over to 0×100. `"position_read()"` reads the high part of the data—now 0×1—and returns a position of `0X1ff`, clearly in error and perhaps even impossible.

This is a common problem. Handling input from a two axis controller? If the hardware continues to move during our reads, then the X and Y data will be slightly uncorrelated, perhaps yielding impossible results. One friend tracked a rare autopilot failure to the way the code read a flux-gate compass, whose output is a pair of related quadrature signals. Reading them at disparate times, while the vessel continued to move, yielded impossible heading data.

## 4.3 eXtreme Instrumenting

In 1967 Keuffel & Esser (the greatest of the slide rule companies) commissioned a study of the future. They predicted by 2067 we'd see three-dimensional TVs and cities covered by majestic domes. The study somehow missed the demise of the slide rule (their main product) within 5 years.

Our need to compute, to routinely deal with numbers, led to the invention of dozens of clever tools, from the abacus to logarithm tables to the slide rule. All worked in concert with the user's brain, in an iterative, back and forth process that only slowly produced answers.

Now even grade school children routinely use graphing calculators. The device assumes the entire job of computation and sometimes even data analysis. What a marvel of engineering! Powered by nothing more than a stream of photons, pocket-sized, and costing virtually nothing, our electronic creations gives us astonishing new capabilities.

Those of us who spend our working lives parked in front of computers have even more powerful computational tools. The spreadsheet is a multidimensional version of the hand calculator, manipulating thousands of formulas and numbers with a single keystroke. Excel is one of my favorite engineering tools. It lets me model weird systems without writing a line of code and tune the model almost graphically. Computational tools have evolved to the point where we no longer struggle with numbers; instead, we ask complex "what-if" questions.

Network computing lets us share data. We pass spreadsheets and documents among coworkers with reckless abandon. In my experience widely shared big spreadsheets are usually incorrect. Someone injects a row or column, forgetting to adjust a summation or other formula. The data at the end is so complex, based on so many intermediate steps, that it's hard to see if it's right or wrong … so we assume it's right. This is the dark side of a spreadsheet: no other tool can make so many incorrect calculations so fast.

Mechanical engineers now use finite element analysis to predict the behavior of complex structures under various stresses. The computer models a spacecraft vibrating as it is boosted to orbit, giving the designers insight into its strength without running expensive tests on shakers. Yet, finite element analysis is so complex, with millions of interrelated calculations! How do they convince themselves that a subtle error isn't lurking in the model? Like subtle errors lurking hidden in large spreadsheets, the complexity of the calculations removes the element of "feel." Is that complex carbon-fiber structure strong enough when excited at 20 Hz? Only the computer knows for sure.

The modern history of engineering is one of increasing abstraction from the problem at hand. The C language insulates us from the tedium of assembly, which itself removes us from machine code. Digital ICs protect us from the very real analog behavior of each of the millions of transistors encapsulated in the chip. When we embed an operating system into a product we're given a wealth of services we can use without really understanding the how and why of their operation.

Increasing abstraction is both inevitable and necessary. An example is the move to object-oriented programming, and more importantly, software reuse, which will—someday—lead to "software ICs" whose operation is as mysterious as today's giant LSI devices, yet that elegantly and cheaply solve some problem.

But, abstraction comes at a price. In too many cases we're losing the "feel" of the problem. Engineering has always been about building things, in the most literal of contexts. Building, touching, and experiencing failure are the tactile lessons that burn themselves into the wiring of our brains. When we delve deeply into how and why things work, when we get burned by a hot resistor, when we've had a tantalum capacitor installed backward explode in our face, when a CMOS device fails from excessive undershoot on an input, we develop our own rules of thumb that give us a new understanding of electronics. Book learning tells us what we need to know. Handling components and circuits builds a powerful subconscious knowledge of electronics.

A friend who earns his keep as a consultant sometimes has to admit that a proposed solution looks good on paper but just does not feel right. Somehow we synthesize our experience into an emotional reaction as powerful and immediate as any other feeling.

I've learned to trust that initial impression and to use that bit of nausea as a warning that something is not quite right. The ground plane on that PCB just doesn't look heavy enough. The capacitors seem a long way from the chips. That sure seems like a long cable for those fast signals. Gee, there's a lot of ringing on that node.

Practical experience has always been an engineer's stock-in-trade. We learn from our successes and our failures. This is nothing new. According to *Cathedral, Forge and Waterwheel* (Frances and Joseph Gies, 1994, HarperCollins, NY), in the Middle Ages "[e]ngineers had some command of geometry and arithmetic. What they lacked was engineering theory, in place of which they employed their own experience, that of their colleagues, and rule of thumb."

The flip side of a "feel" for a problem is an ability to combine that feeling with basic arithmetic skills to very quickly create a first approximation to a solution, something often called "guesstimating." This wonderful word combines "guess"—based on our engineering feel for a problem—and "estimate"—a partial analytical solution.

Guesstimates are what keep us honest; "200,000 bits per second seems kind of fast for an 8 bit micro to process" (this is the guess part), "why, that's 1/200,000 or 5 μsec per bit" (the estimate part). Maybe there's a compelling reason why this guesstimate is incorrect, but it flags an area that needs study.

In 1995 an Australian woman swam the 110 miles from Havana to Key West in 24 hours. Public Radio reported this information in breathless excitement, while I was left baffled. My guesstimate said this is unlikely. That's a 4.5 mph average, a pace that's hard to beat even with a brisk walk, yet the she maintained this for a solid 24 hours.

Maybe swimmers are speedier than I'd think. Perhaps the Gulf Stream spun off a huge gyre, a rotating current that gave her a remarkable boost in the right direction. I'm left puzzled, as the data fails my guesstimating sense of reasonableness. And so, though our sense of "feel" can and should serve as a measure against which we can evaluate the mounds of data tossed our way each day, it is imperfect at best.

The art of "guesstimating" was once the engineer's most basic tool. Old engineers love to point to the demise of the slide rule as the culprit. "Kids these days," they grumble. Slide rules forced one to estimate the solution to every problem. The slide rule did force us to have an easy familiarity with numbers and with making coarse but rapid mental calculations.

We forget, though, just how hard we had to work to get anything done! Nothing beats modern technology for number crunching, and I'd never go back. Remember that the slide rule *forced* us to estimate all answers; the calculator merely *allows* us to accept any answer as gospel without doing a quick mental check.

We need to grapple with the size of things, every day and in every avenue. A million times a million is, well, $10^{12}$. The gigahertz is a period of 1 ns. For a swimmer 4.5 miles per hour seems fast. It's unlikely your ISR will complete in 2 μsec.

We're building astonishing new products, the simplest of which have hundreds of functions requiring millions of transistors. Without our amazing tools and components, those things that abstract us from the worries of biasing each individual transistor, we'd never be able to get our work done. Though the abstraction distances us from how things work, it enables us to make things work in new and wondrous ways.

The art of guesstimating fails when we can't or don't understand the system. Perhaps in the future we'll need computer-aided guesstimating tools, programs that are better than feeble humans at understanding vast interlocked systems. Perhaps this will be a good thing. Maybe, like double-entry bookkeeping, a computerized guesstimater will at least allow a cross-check on our designs.

As a nerdy kid in the 1960s I was steered by various mentors to vacuum tubes long before I ever understood semiconductors. A tube is wonderfully easy to understand. Sometimes you can quite literally see the blue glow of electrons splashing off the plate onto the glass. The warm glow of the filaments, the visible mesh of the control grids, always conjured a crystal clear mental image of what was going on.

A 100,000 gate ASIC is neither warm nor clear. There's no emotional link between its operation and your understanding of it. It's a platonic relationship at best.

So, what's an embedded engineer to do? How can we re-establish this "feel" for our creations, this gut-level understanding of what works and what doesn't?

The first part of learning to guesstimate is to gain an intimate understanding of how things work. We should encourage kids to play with technology and science. Help them get their hands greasy. It matters little if they work on cars, electronics, or in the sciences. Nurture that odd human attribute that couples doing with learning.

The second part of guesstimation is a quick familiarity with math. Question engineers (and your kids) deeply about things. "Where did that number come from?" "Do you believe it … and why?"

Work on your engineers' understanding of orders of magnitude. It's astonishing how hard some people work to convert frequency to period, yet this is the most common calculation we do in computer design. If you know that a microsecond is a megahertz, a millisecond is a 1000 Hz, you'll never spend more than a second getting a first approximation conversion.

The third ingredient is to constantly question everything. As the bumper sticker says, "question authority." As soon as the local expert backs up his opinion with numbers, run a quick mental check. He's probably wrong.

In *To Engineer is Human* (1982, Random House, NY), author Henry Petroski says "magnitudes come from a *feel* for the problem, and do not come automatically from machines or calculating contrivances." Well put, and food for thought for all of us.

A simple CPU has very predictable timing. Add a prefetcher or pipeline and timing gets fuzzier, but still is easy to figure within 10% or 20%. Cache is the wild card, and as cache size increases determinism diminishes. Thankfully, today few small embedded CPUs have even the smallest amount of cache.

Your first weapon in the performance arsenal is developing an understanding of the target processor. What can it do in 1 µsec? One instruction? Five? Some developers use very, very slow clocks when not much has to happen—one outfit I know runs the CPU (in a spacecraft) at 8 KHz until real speed is needed. At 8 KHz they get maybe 5000 instructions per second. Even small loops become a serious problem. Understanding the physics—a perhaps fuzzy knowledge of just what the CPU can do at this clock rate—means the big decisions are easy to make.

Estimation is one of engineering's most important tools. Do you think the architect designing a house does a finite element analysis to figure the size of the joists? No! He refers to a manual of standards. A 15 foot unsupported span typically uses joists of a certain size. These estimates, backed up with practical experience, insure that a design, while perhaps not optimum, is adequate.

We do the same in hardware engineering. Electrons travel at about 1 or 2 feet/nsec, depending on the conductor. It's hard to make high frequency first harmonic crystals, so use a higher order harmonic. Very small PCB tracks are difficult to manufacture reliably. All of these are ingredients of the "practice" of the art of hardware design. None of these is tremendously accurate: you can, after all, create one mil tracks on a board for a ton of money. The exact parameters are fuzzy, but the general guidelines are indeed correct.

So too for software engineering. We need to develop a sense of the art. A 68HC16, at 16 MHz, runs so many instructions per second (plus or minus). With this particular compiler you can expect (more or less) this sort of performance under these conditions.

Data, even fuzzy data, lets us bound our decisions, greatly improving the chances of success. The alternative is to spend months and years generating a mathematically precise solution—which we won't do—or to burn incense and pray… the usual approach.

Experiment. Run portions of the code. Use a stopwatch—metaphorical or otherwise—to see how it executes. Buy a performance analyzer or simply instrument sections of the firmware to understand the code's performance.

The first time you do this you'll think "this is so cool," and you'll walk away with a clear number: xxx microseconds for this routine. With time you'll develop a sense of speed. "You know, integer compares are pretty darn fast on this system." Later—as you develop a sense of the art—you'll be able to bound things. "Nah, there's no way that loop can complete in 50 μsec."

This is called experience, something that we all too often acquire haphazardly. We plan our financial future, work daily with our kids on their homework, even remember to service the lawnmower at the beginning of the season, yet neglect to proactively improve our abilities at work.

Experience comes from exposure to problems and from learning from them. A fast, useful sort of performance expertise comes from extrapolating from a current product to the next. Most of us work for a company that generally sells a series of similar products. When it's time to design a new one we draw from the experience of the last, and from the code and design base. Building version 2.0 of a widget? Surely you'll use algorithms and ideas from 1.0. Use 1.0 as a testbed. Gather performance data by instrumenting the code.

Always close the feedback loop. When any project is complete, spend a day learning about what you did. Measure the performance of the system to see just how accurate your processor utilization estimates were. The results are always interesting and sometimes terrifying. If, as is often the case, the numbers bear little resemblance to the original goals, then figure out what happened and use this information to improve your estimating ability. Without feedback, you work forever in the dark. Strive to learn from your successes as well as your failures.

Track your system's performance all during the project's development, so you're not presented with a disaster 2 weeks before the scheduled delivery. It's not a bad idea to assign CPU utilization specifications to major routines during overall design, and then track these targets like you do the schedule. Avoid surprises with careful planning.

A lot of projects eventually get into trouble by overloading the processor. This is always discovered late in the development, during debugging or final integration, when the cost of correcting the problem is at the maximum. Then a mad scramble to remove machine cycles begins.

We all know the old adage that 90% of the processor burden lies in 10% of the code. It's important to find and optimize that 10%, not some other section that will have little impact on the system's overall performance. Nothing is worse than spending a week optimizing the wrong routine.

If you understand the design, if you have a sense of the CPU, you'll know where that 10% of the code is before you write a line. Knowledge is power.

Learn about your hardware. Pure software types often have no idea that the CPU is actively working against them. I talked to an engineer who was moaning about how slow his new 386EX-based instrument runs. He didn't know that the 386EX starts with 31 wait states … and so had never reprogrammed it to a saner value.

### 4.3.1 Performance Measurements

The county was repaving my street. Workers brought in a big, noisy machine that ate away 2 inches of the old surface in a single pass, feeding a few cubic meters of ground-up rubble to a succession of dump trucks each second. Then an even larger contraption pulled hot and oily asphalt from other trucks and put a pool-table-flat road down in a single pass. A small army of acolytes milled around the machine as it crept slowly along. A couple fed raw material into it, another tickled mysterious levers to control the beast, and some workers directed the procession of traffic. I walked along with them for quite a while, as it's always interesting to see how people do their job.

I watch embedded developers do their thing, too. Joe Coder is hunched in front of his monitor, furiously single-stepping, examining watchpoints, and using a debugger to insure his code executes correctly. When the thing finally works he breathes a huge sigh of relief and moves on to the next project.

That approach makes me shudder.

Joe builds real-time code. He's doing the *code* part of that just fine since traditional debugging gives us great insight into the program's procedural domain. That's the `if-then`, `do-while` bit that represents the visible components of all software.

But he's totally ignoring the *real-time* part of his job. How long does an interrupt handler take? A microsecond … or a week? Is the unit idle 90% of the time … or 1%?

A management maxim states "if you can't measure it, you can't manage it," which certainly holds true for developing embedded systems. We can and do measure everything about the procedural nature of the code; similarly we can and must measure and manage the nature of our systems in the time domain.

That was easy in the olden days. A dozen or more in-circuit emulator vendors offered tools that handled both code's time and procedural nature. Sophisticated trace caught every aspect of system operation at full execution speed. Time stamps logged what happened when. Performance analyzers isolated bottlenecks.

Then processors got deep pipelines, making it difficult to know what the core chip was doing. Caches followed, changing difficult to impossible. Now the processor is buried inside an FPGA or ASIC. The Age of Emulators faded as surely as the Pleistocene, replaced now by the JTAG Epoch. Ironically, as the size and scope of embedded apps exploded our tools' capabilities imploded. Most debuggers today offer little help with managing the time domain. But we *must* measure time to understand what the systems are doing, to find those complex real-time bugs, and to understand where the microseconds go.

## 4.3.2 Output Bits

Since the tools aren't up to snuff, stride into the EEs' offices and demand, at least on the prototypes, one or more parallel outputs dedicated to debugging. It's astonishing how much insight one can glean from a system by simply instrumenting the code to drive these bits.

Want to know the execution time of any routine? Drive one of the bits high when the function starts, and low as it exits. Monitor the bit with an oscilloscope—which every lab has—and you can measure time to any precision you'd like. The cost: pretty much zero. Insight: a lot.

The top trace in Figure 4.8 monitors an interrupt request line. On the bottom we see the output bit, driven high by the code when the corresponding ISR starts, and low immediately before exit. The time from the leading edge of the request to the assertion of the bit is the interrupt latency, a critical parameter that lets us know that the system will respond to interrupts in a timely manner. Then, duration of that bit's assertion tells us the ISR's execution time.

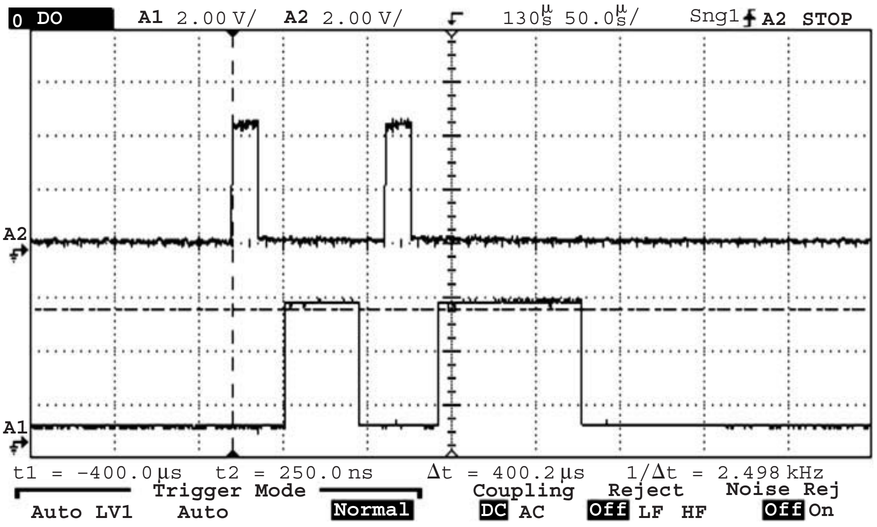


Figure 4.8: Top: Interrupt request. Bottom: ISR execution time

Note that the same ISR has been invoked twice, with different responses each time. No doubt there's a decision in the code that changes the execution path. Or a pipeline might be filling, or any of a dozen other factors might affect the routine's performance. Trigger a digital scope on the bottom trace's rising edge, as shown in Figure 4.9. The time from the bit being asserted to the beginning of the hash is the fastest the thing ever runs; to the end of the hash is the slowest.

Wow—two lines of code and one output bit gives a *ton* of quantitative information!

Wise managers demand parametric data about firmware performance at the end of a project. How much free flash space do we have? How about RAM? Without that data it's impossible to know if it's possible to enhance features in the future. The same goes for performance numbers. If the system is 99.9% loaded, adding even the simplest functionality will have you emulating Sisyphus for a very long time.

Instrument the idle loop or create a low priority task that just toggles the output bit, as shown in Figure 4.10. Where there's hash, it's idle. Where there's not, the system is busy.

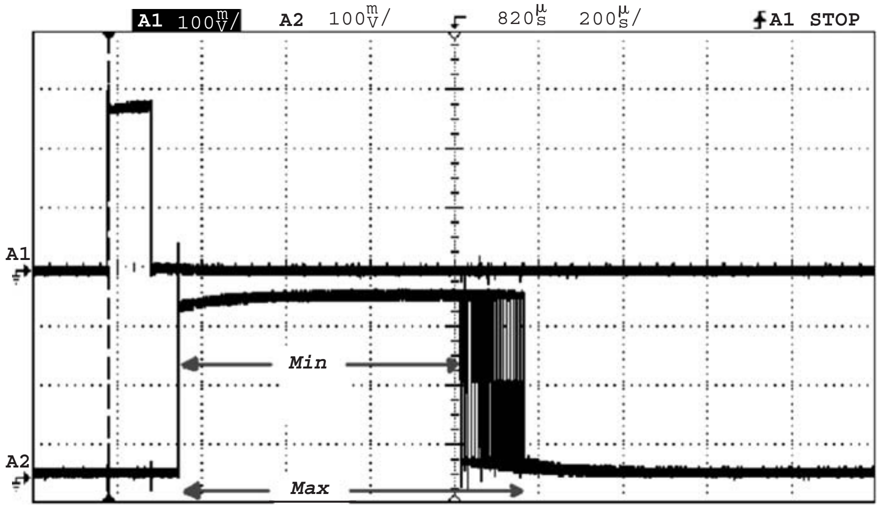


**Figure 4.9: Measuring min and max times**

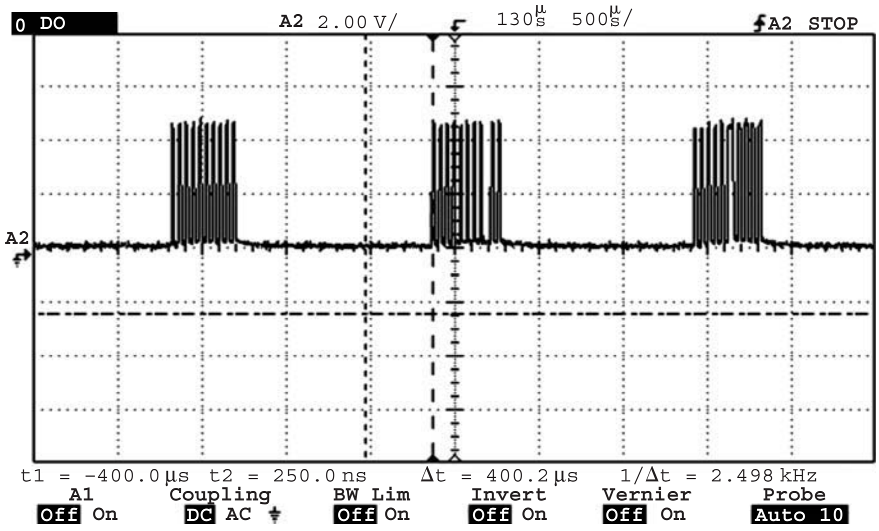


**Figure 4.10: Measuring idle time**

This is easy to do since many operating systems have a built-in hook that's called whenever the system has nothing to do. Micrium's uC/OS-II, for instance, invokes the following routine, to which I added instructions to toggle the output bit:

```
/**************************************************
*                                    IDLE TASK HOOK
**************************************************/
void OSTaskIdleHook (void)
{
   outportb(test_port, 1);    // Assert instrumentation pin
   outportb(test_port, 0);    // ... but just for a moment
}
```

The cost: 480 nsec on a 33 MHz 186.

### 4.3.3 The VOM Solution

What we really want is a performance analyzer, an instrument that's always connected to the system to constantly monitor idle time. The tool then immediately alerts you if new or changed code suddenly sucks cycles like a Hummer goes through gas. But plan on

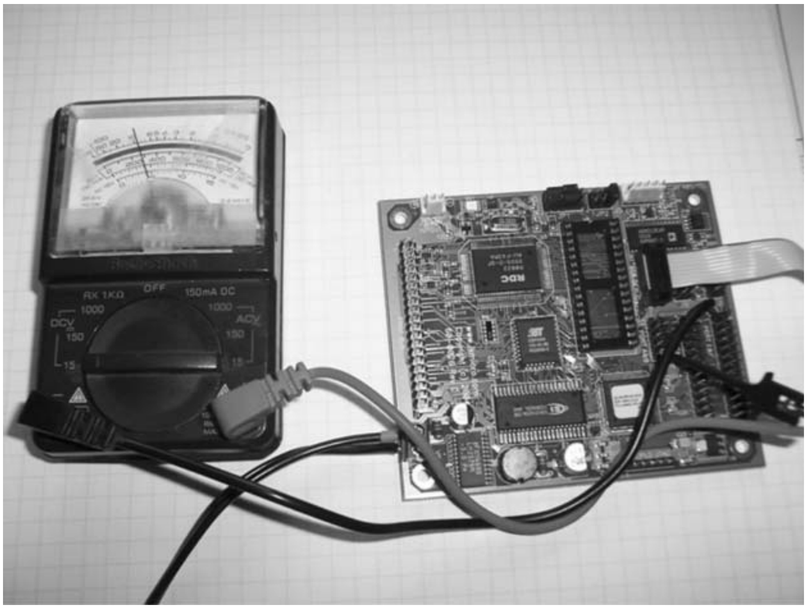


Figure 4.11: A poor person's performance analyzer

spending a few tens of thousands for the tool, assuming such a tool even exists for the CPU you're using.

Or not. Remove all load from your system so the idle hook runs nearly all of the time. Use the scope to figure the duty cycle of our trusty output bit. On my 33 MHz 186 system running uC/OS-II the duty cycle is 8.6%.

Now get Radio Shack's 22–218A voltmeter (about $15) and remove the back cover. Find the 29.2 K resistor and change it to one whose value is:

$$R = \frac{\text{DutyCycle}}{100} \times \text{MaxVolts} \times 2000$$

where DutyCycle is in percent and MaxVolts is the system's power supply voltage.

Monitor the output bit with the meter as shown in Figure 4.11. Your cheap VOM is now a $10k performance analyzer. It'll show the percentage of time the system is idle. Leave it hooked up all the time to see the effect of new code and different stimuli to the system.

The Radio Shack salesperson tried to sell me an extended service plan, but I'm pretty sure this mod will void the warranty.

It's fun to watch colleagues' jaws drop when you explain what you're doing.

The needle's response won't keep up with millisecond-level changes in system loading. Not a problem; modify the idle hook to increment a variable called `Idle_Counts` and invoke the following task every second:

```
static void Compute_Percent_Idle_Time (void *p_arg)
{
float Num_Idles_Sec=178571.0;      // In fully idle system we
                                   // get this many
                                   // counts/sec
float Idle;                        // Percent idle time
while(1){
    Idle= 100.0 * (((float) Idle_Counts)/Num_Idles_Sec);
    printf("\nIdle time in percent= %f", Idle);
    Idle_Counts=0;
    OSTimeDly(OS_TICKS_PER_SEC);  // Go to sleep
  }
}
```

Obviously the `Num_Idles_Sec` parameter is system-dependent. Run the code in a totally idle system and look at `Idle_Counts` to see how many counts you get in your configuration.

Modify the routine to suit your requirements. Add a line or two to compute min and max limits. I naughtily used `printf` to display the results, which on my 186 test system burns 40 msec. This is a Heisenberg effect: making the measurement changes the system's behavior. Better, log the results to a variable or send them to whatever output device you have. Consider using longs instead of floats. But you get the idea.

## 4.3.4 R-2R

Some lucky readers work with a hardware designer who has a fond spot for the firmware crowd. Buy him a beer. Wash her car. Then ask for more than a single output bit—maybe even three.

Construct the following circuit and connect the three points with PIO designations to the output bits. Often called an R-2R latter network, this is an inexpensive and not-terribly-accurate digital to analog converter (Figure 4.12).

Now we can look at the real-time behavior of tasks … in real time. Want to know which task executes when? Again using uC/OS-II, change the Micrium-supplied hook that executes whenever a task is invoked as follows:

```
/*********************************
*                    TASK SWITCH HOOK
*********************************/
void OSTaskSwHook (void)
{
  outportb(test_port,
          OSTCBCur-<OSTCBId);
}
```

This sends the next task's ID (a number from 0 to whatever) to those output bits. Probe the R-2R ladder with the scope and you'll see something that looks like Figure 4.13.

Task 0 is running when the scope reads zero volts. Task 1 when the voltage is 1/8th of the power supply, etc. By adding less than a microsecond of overhead we can watch how our system runs dynamically. Sometimes it's useful to trigger the scope on some event—say, a

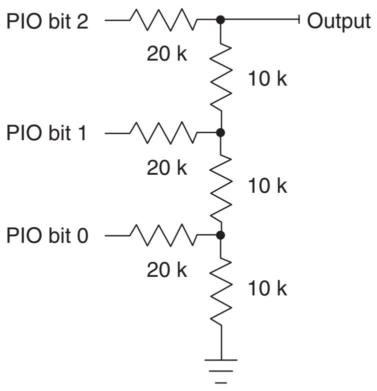


**Figure 4.12: A $0.05 D/A converter**

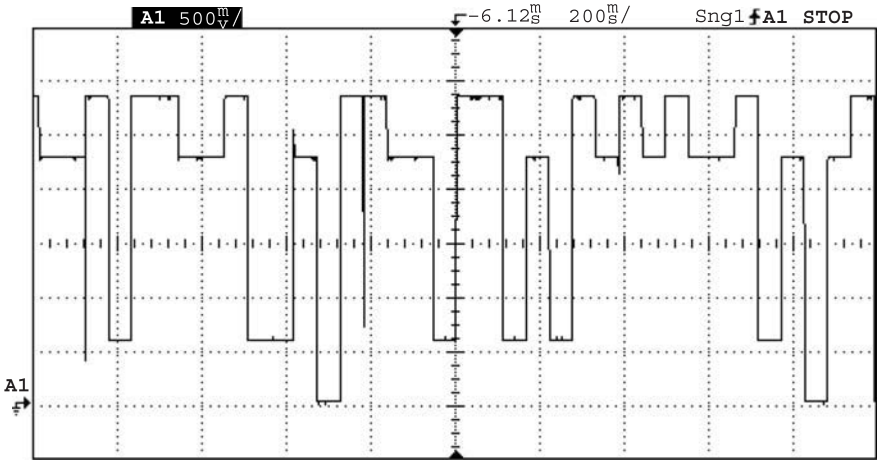


Figure 4.13: Executing tasks, in real time

button press or the start of incoming data. The scope will show how the system schedules tasks to deal with that event.

Three bits lets us monitor eight tasks. More bits, more tasks. But the resistor network isn't very accurate, so after about four bits use a real D/A converter instead.

The same technique can monitor the system's mode, if you maintain mode or status info in a few bits. Or the size of a stack or the depth of a queue. Compute the data structure's size, correct for wrap on circular queues, and output the three or so MSBs. In real time, with little Heisenberging, you'll see system behavior—dramatically. It's simple, quick, inexpensive … and way cool.

## 4.4 Floating Point Approximations

Most embedded processors don't know how to compute trig and other complex functions. Programming in C we're content to call a library routine that does all of the work for us. Unhappily this optimistic approach often fails in real-time systems where size, speed, and accuracy are all important issues.

The compiler's run-time package is a one-size-fits-all proposition. It gives a reasonable tradeoff of speed and precision. But every embedded system is different, with different requirements. In some cases it makes sense to write our own approximation routines. Why?

*Speed*: Many compilers have very slow run-time packages. A clever approximation may eliminate the need to use a faster CPU.

*Predictability*: Compiler functions vary greatly in execution time depending on the input argument. Real-time systems must be *predictable* in the time domain. The alternative is to always assume worst case execution time, which again may mean your CPU is too slow, too loaded, for the application.

*Accuracy*: Read the compilers' manuals carefully! Some explicitly do not support the ASNI C standard, which requires all trig to be double precision. (8051 compilers are notorious for this.) Alternatively, why pay the cost (in time) for double precision math when you only need 5 digits of accuracy?

*Size*: When memory is scarce, using one of these approximations may save much code space. If you only need a simple cosine, why include the entire floating point trig library?

Most of the complicated math functions we use compute what is inherently not computable. There's no precise solution to most trig functions, square roots, and the like, so libraries employ approximations of various kinds. We, too, can employ various approximations to meet peculiar embedded requirements.

College taught us that any differentiable function can be expressed by a Taylor series, such as:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Like all approximations used in run-time packages, this is simply a polynomial that's clear, simple, and easy to implement. Unhappily it takes dozens of terms to achieve even single precision float accuracy, which is an enormous amount of computation.

Plenty of other series expansions exist, many of which converge to an accurate answer much faster than does the Taylor series. Entire books have been written on the subject and one could spend years mastering the subject.

But we're engineers, not mathematicians. We need cookbook solutions now. Happily computer scientists have come up with many.

The best reference on the subject is the suitably titled *Computer Approximations*, by John Hart et al., Second Edition, 1978, Robert Krieger Publishing Company, Malabar, FL. It contains thousands of approximations for all of the normal functions like roots, exponentials, and trig, as well as approaches for esoteric relations like Bessel Functions and Elliptic Integrals. It's a must-have for those of us working on real-time resource-constrained systems.

But it's out of print. You can pry my copy out of my cold, dead fingers when my wetwear finally crashes for good. As I write this there are six copies from Amazon Marketplace for about $250 each. Long ago I should have invested in Google, Microsoft … and *Computer Approximations*.

Be warned: it's written by eggheads for eggheads. The math behind the derivation of the polynomials goes over my head.

Tables of polynomial coefficients form the useful meat of the book. Before each class of function (trig, roots, etc.) there's a handful of pages with suggestions about using the coefficients in a program. Somewhere, enigmatically buried in the text, like an ancient clue in a Dan Brown thriller, an important point about actually using the coefficients lies lost. So here's how to interpret the tables.

An "index" vectors you to the right set of tables. An index for square roots looks like this:

$$\begin{array}{llll} P(x) & & & \\ \left[\frac{1}{\sqrt{100}}, 1\right] & 2.56 & 4 & 0033 \end{array}$$

which means: the set of coefficients labeled "0033" is for a polynomial of degree 4, accurate to 2.56 decimal digits for inputs between $1/\sqrt{100}$ and 1. Outside that range all bets are off. The polynomial is of the form `P(x)`. (Several other forms are used, like `P(x)/Q(x)`, so that bit of what seems like pedantry is indeed important.)

This is the corresponding table of coefficients:

| | | |
|---|---|---|
| P00 | (+ 0) | + 0.14743 837 |
| P01 | (+ 1) | + 0.19400 802 |
| P02 | (+ 1) | − 0.26795 117 |
| P03 | (+ 1) | + 0.25423 691 |
| P04 | (+ 0) | − 0.95312 89 |

The number in parentheses is the power of 10 to apply to each number. Entry P02, prefixed by (+1), is really −2. 6795117.

It took me a long time to figure that out.

Knowing this the polynomial for the square root of x, over that limited range, accurate to 2.56 digits is:

$$\sqrt{x} = 0.14743\,837 + 1.9400\,802 \times x - 2.6795117 \times x^2 + 2.5423\,691 \times x^3 - 0.95312\,89 \times x^4$$

It looks like a lot of trouble for a not particularly accurate way to figure roots. The genius of this book, though, is that he lists so many variations (90 for square roots alone) that you can make tradeoffs to select the right polynomial for your application.

Three factors influence the decision: required accuracy, speed, and range. Like all good engineering predicaments, these conspire against each other. High speed generally means lower accuracy. Extreme accuracy requires a long polynomial that will burn plenty of CPU cycles, or a microscopic range. For instance, the following coefficients express $2^x$ to 24.78 decimal digits of accuracy:

| | | |
|---|---|---|
| P00 | (+1) | +0.72134 75314 61762 84602 46233 635 |
| P01 | (−1) | +0.57762 26063 55921 17671 75 |
| Q00 | (+2) | +0.20813 69012 79476 15341 50743 885 |
| Q01 | (+1) | +0.1 |

… using the polynomial:

$$2^x = \frac{(Q(x^2) + x \times P(x^2))}{(Q(x^2) - x \times P(x^2))}$$

A bit of clever programming yields a routine that is surprisingly fast for such accuracy. Except that the range of *x* is limited to 0 to 1/256, which isn't particularly useful in most applications.

The trick is to do a bit of magic to the input argument to reduce what might be a huge range to one that satisfies the requirements of the polynomial. Since the range reduction process consumes CPU cycles it is another factor to consider when selecting an approximation.

Hart's book does describe range reduction math … in a manner an academic would love. But by puzzling over his verbiage and formulas, with some doodling on paper, it's not too hard to construct useful range reduction algorithms.

### 4.4.1 Roots

For example, I find the most useful square root polynomials require an input between 0.25 and 1. Since $\sqrt{x} = 2^k \times \sqrt{x \times 2^{-2k}}$ we can keep increasing $k$ till $x \times 2^{-2k}$ is in the required range. Then use the polynomial to take the square root, and adjust the answer by $2^k$, as follows:

```
// reduce_sqrt-The square root routines require an input argument in
// the range[0.25, 1]. This routine reduces the argument to that range.
//
// Return values:
// - "reduced_arg", which is the input arg/2^(2k)
// - "scale", which is the sqrt of the scaling factor, or 2^k
//
// Assumptions made:
// - The input argument is > zero
// - The input argument cannot exceed +2^31 or be under 2^-31.
//
// Possible improvements:
// - To compute (a* 2^(-2k)) we do the division (a/(2^2k)). It's
//   much more efficient to decrement the characteristic of "a" (i.e.,
//   monkey with the floating point representation of "a") the
//   appropriate number of times. But that's far less portable than
//   the less efficient approach shown here.
//
// How it works:
//  The algorithm depends on the following relation:
// sqrt(arg)=(2^k) * sqrt(arg * 2^(-2k))
//
```

```
// We pick a "k" such that 2^(-2k) * arg is less than 1 and greater
//   than 0.25.
//
// The ugly "for" loop shifts a one through two_to_2k while shifting
//  a long version of the input argument right two times, repeating
//  till the long version is all zeroes. Thus, two_to_2k and the input
//  argument have this relationship:
//
//       two_to_2k       input argument        sqrt(two_to_2k)
//       4               1to 4                  2
//       16              5 to 16                4
//       64              17 to 64               8
//       256             65 to 256             16
//
// Note that when we're done, and then divide the input arg by
//  two_to_2k, the result is in the desired [0.25, 1] range.
//
// We also must return "scale", which is sqrt(two_to_2k), but we
// prefer not to do the computationally expensive square root. Instead
// note (as shown in the table above) that scale simply doubles with
// each iteration.
//
// There's a special case for the input argument being less than
// the [0.25,1]range. In this case we multiply by a huge number, one
// sure to bring virtually every argument up to 0.25, and then adjust
// "scale" by the square root of that huge number.
//


void reduce_sqrt(double arg, double *reduced_arg, double *scale){
  long two_to_2k;         // divisor we're looking for: 2^(2k)
  long l_arg;             // 32 bit version of input argument
  const double huge_num=1073741824;     // 2**30
  const double sqrt_huge_num=32768;     // sqrt(2**30)

  if(arg>=0.25){
    // shift arg to zero while computing two_to_2k as described above
    l_arg=(long) arg;      // l_arg is long version of input arg
    for(two_to_2k=1, *scale=1.0; l_arg!=0;l_arg>>=2, two_to_2k<<=2,
     *scale*=2.0);
    // normalize input to [0.25, 1]
    *reduced_arg=arg/(double) two_to_2k;
  }else
   {
   // for small arguments:
   arg=arg*huge_num;          // make the number big
   l_arg=(long) arg;          // l_arg is long version of input arg
   for(two_to_2k=1, *scale=1.0; l_arg!=0; l_arg>>=2, two_to_2k<<=2,
    *scale*=2.0);
   *scale=*scale/sqrt_huge_num;
   // normalize input argument to [0.25, 1]
   *reduced_arg=arg/(double) two_to_2k;
   };
};
```

Here's a sampling of interesting polynomials for square roots. Note that they can all be used with the range reduction scheme I've described.

With just two terms, using the form $P(x)$ over the range [1/100, 1], this one is accurate to 0.56 decimal digits and is very fast:

| | | |
|---|---|---|
| P00 (+ 0) | + 0.11544 | 2 |
| P01 (+ 1) | + 0.11544 | 2 |

A different form $(P(x)/Q(x))$ gives 3.66 digits over 1/4 to 1:

| | |
|---|---|
| P00 (− 1) | + 0.85805 2283 |
| P01 (+ 1) | + 0.10713 00909 |
| P02 (+ 0) | + 0.34321 97895 |
| Q00 (+ 0) | + 0.50000 08387 |
| Q01 (+ 1) | + 0.1 |

The following yields 8.95 digits accuracy using $(P(x)/Q(x))$, but works over the narrower 1/2 to 1 range. Use the usual range reduction algorithm to narrow $x$ to 0.25

to 1, and then double the result if it's less than 1/2. Scale the result by the square root of two:

| | | |
|---|---|---|
| P00 | (+ 0) | + 0.29730 27887 4025 |
| P01 | (+ 1) | + 0.89403 07620 6457 |
| P02 | (+ 2) | + 0.21125 22405 69754 |
| P03 | (+ 1) | + 0.59304 94459 1466 |
| Q00 | (+ 1) | + 0.24934 71825 3158 |
| Q01 | (+ 2) | + 0.17764 13382 80541 |
| Q02 | (+ 2) | + 0.15035 72331 29921 |
| Q03 | (+ 1) | + 0.1 |

Hart's book also contains coefficients for cube roots. Reduce the argument's range to 0.5 to 1 (generally useful with his cube root approximations) using the relation:

$$\sqrt[3]{x} = 2^k \times \sqrt[3]{x \times 2^{-3k}},$$

as follows:

```
//
// reduce_cbrt-The cube root routines require an input argument in
// the range[0.5, 1].This routine reduces the argument to that range.
//
// Return values:
// - "reduced_arg", which is the input arg/2^(3k)
// - "scale", which is the cbrt of the scaling factor, or 2^k
//
// Assumptions made:
// - The input argument cannot exceed +/-2^31 or be under +/-2^-31.
//
// Possible improvements:
// - To compute a*2^(-3k)) we do the division (a/(2^3k)). It's
//   much more efficient to decrement the characteristic of "a"
//   (i.e., monkey with the floating point representation of "a") the
//   appropriate number of times. But that's far less portable than the
//   less efficient approach shown here.
// - Corrections made for when the result is still under 1/2 scale by
//   two. It's more efficient to increment the characteristic.
//
// How it works:
//  The algorithm depends on the following relation:
//  cbrt(arg)=(2^k) * cbrt(arg * 2^(−3k))
//
//  We pick a "k" such that 2^(-3k) * arg is less than 1 and greater
//  than 0.5.
//
//  The ugly "for" loop shifts a one through two_to_3k while shifting a
//  long version of the input argument right three times, repeating
//  till the long version is all zeros. Thus, two_to_3k and the input
//  argument have this relationship:
//
//      two_to_3k        input argument          cbrt(two_to_3k)
//      8                1 to 7                   2
//      64               8 to 63                  4
//      512              64 to 511                8
//      4096             512 to 4095             16
//
//  Note that when we're done, and then divide the input arg by
//  two_to_3k, the result is between [1/8,1]. The following algorithm
//  reduces it to [1/2, 1]:
//
//  if (reduced_arg is between [1/4, 1/2])
//           multiply it by two and correct the scale by the cube root
//           of two.
//  if (reduced_arg is between [1/4, 1/2])
//           multiply it by two and correct the scale by the cube root
//           of two.
//
//  Note that the if the argument was between [1/8, 1/4] both of those
//  "if"s will execute.
//
//  We also must return "scale", which is cbrt(two_to_3k), but we
//  prefer not to do the computationally expensive cube root. Instead
//  note (as shown in the table above) that scale simply doubles with
//  each iteration.
//
```

```
//  There's a special case for the input argument being less than the
//  [0.5,1] range. In this case we multiply by a huge number, one sure
//  to bring virtually every argument up to 0.5, and then adjust
//  "scale" by the cube root of that huge number.
//
//  The code takes care of the special case that the cube root of a
//  negative number is negative by passing the absolute value of the
//  number to the approximating polynomial, and then making "scale"
//  negative.
//


void reduce_cbrt(double arg, double *reduced_arg, double *scale){
  long two_to_3k;             // divisor we're looking for: 2^(3k)
  long l_arg;                 // 32 bit version of input argument
  const double huge_num=1073741824;          // 2**30
  const double cbrt_huge_num=1024;           // cbrt(2**30)
  const double cbrt_2= 1.25992104989487;     // cbrt(2)



  *scale=1.0;
  if(arg<0){                  // if negative arg, abs(arg) and set
    arg=fabs(arg);            // scale to -1 so the polynomial
routine
    *scale=-1.0;              // will give a negative result
    };

  if(arg>=0.5){
    // shift arg to zero while computing two_to_3k as described above
    l_arg=(long) arg;         // l_arg is long version of input arg
    for(two_to_3k=1; l_arg!=0; l_arg>>=3, two_to_3k<<=3, *scale*=2.0);
    *reduced_arg=arg/(double) two_to_3k; // normalize input to [0.5, 1]
    if(*reduced_arg<0.5){     // if in the range [1/8,1/2] correct it
      *reduced_arg*=2;
      *scale/=cbrt_2;
    };
    if(*reduced_arg<0.5){     // if in the range [1/4,1/2] correct it
      *reduced_arg*=2;
      *scale/=cbrt_2;
  };


}else
  {
  // for small arguments:
  arg=arg*huge_num;            // make the number big
  l_arg=(long) arg;            // l_arg is long version of input arg
  for(two_to_3k=1; l_arg!=0; l_arg>>=3, two_to_3k<<=3, *scale*=2.0);
      *scale=*scale/cbrt_huge_num;
      *reduced_arg=arg/(double) two_to_3k; // normalize to [0.25, 1]
      if(*reduced_arg<0.5){    // if in the range [1/8,1/2]
                               // correct it
        *reduced_arg*=2;
        *scale/=cbrt_2;
      };
      if(*reduced_arg<0.5){    // if in the range [1/4,1/2]
                               // correct it
        *reduced_arg*=2;
        *scale/=cbrt_2;
      };
  };
};
```

With just two terms in the form *P*(*x*) these coefficients give a cube root accurate to 1.24 digits over the range 1/8 to 1:

| | | |
|---|---|---|
| P00 | (+ 0) | + 0.45316 35 |
| P01 | (+ 0) | + 0.60421 81 |

Adding a single term improves accuracy to 3.20 digits over 1/2 to 1:

| | | |
|---|---|---|
| P00 | (+ 0) | + 0.49329 5663 |
| P01 | (+ 0) | + 0.69757 0456 |
| P02 | (+ 0) | − 0.19150 216 |

Kick accuracy up by more than a notch, to 11.75 digits over 1/2 to 1 using the form $(P(x)/Q(x))$:

| | | |
|---|---|---|
| P00 | (+ 0) | + 0.22272 47174 61818 |
| P01 | (+ 1) | + 0.82923 28023 86013 7 |
| P02 | (+ 2) | + 0.35357 64193 29784 39 |
| P03 | (+ 2) | + 0.29095 75176 33080 76 |
| P04 | (+ 1) | + 0.37035 12298 99201 9 |
| Q00 | (+ 1) | + 0.10392 63150 11930 2 |
| Q01 | (+ 2) | + 0.16329 43963 24801 67 |
| Q02 | (+ 2) | + 0.39687 61066 62995 25 |
| Q03 | (+ 2) | + 0.18615 64528 78368 42 |
| Q04 | (+ 1) | + 0.1 |

## *4.4.2 Exponentiations*

The three most common exponentiations we use are powers of 2, e, and 10. In a binary computer it's trivial to raise an integer to a power of two, but floating point is much harder. Approximations to the rescue!

Hart's power-of-two exponentials expect an argument in the range `[0.0, 0.5]`. That's not terribly useful for most real-world applications so we must use a bit of code to reduce the input `"x"` to that range.

Consider the equation $2^x=2^a\times 2^{(x-a)}$. Well, duh. That appears both contrived and trite. Yet it's the basis for our range reduction code. The trick is to select an integer "`a`" so that `"x"` is between 0 and 1. An integer, because it's both trivial and fast to raise two to any int. Simply set `"a"` to the integer part of `"x,"` and feed `(x-int(x)`$^{a}$`)` in to the approximating polynomial. Multiply the result by the easily computed $2^a$ to correct for the range reduction.

But I mentioned that Hart wants inputs in the range `[0.0, 0.5]`, not `[0.0, 1.0]`. If `(x-int(a))` is greater than 0.5 use the relation:

$$2^x = 2^a \times 2^{1/2} \times 2^{(x-a-(1/2))}$$

instead of the one given earlier. The code is simpler than the equation. Subtract 0.5 from "x" and then compute the polynomial for (x-int(x)) as usual. Correct the answer by $2^a \times \sqrt{2}$.

Exponents can be negative as well as positive. Remember that $2^{-x} = 1/2^x$. Solve for the absolute value of "x" and then take the reciprocal.

Though I mentioned it's easy to figure $2^a$, that glib statement slides over a bit of complexity. The code listed below uses the worst possible strategy: it laboriously does floating point multiplies. That very general solution is devoid of confusing chicanery so is clear, though slow.

Other options exist. If the input "x" never gets very big, use a look-up table of pre-computed values. Even a few tens of entries will let the resulting $2^x$ assume huge values.

Another approach simply adds "a" to the number's mantissa. Floating point numbers are represented in memory using two parts, one fractional (the characteristic), plus an implied two raised to the other part, the mantissa. Apply "a" to the mantissa to create $2^a$. That's highly non-portable but very fast.

Or, one could shift a "one" left through an int or long "a" times, and then convert that to a float or double.

Here's the range reduction code:

```
//
// reduce_expb-The expb routines require an input argument in
// the range[0.0, 0.5].This routine reduces the argument to that range.
//
// Return values:
// - "arg", which is the input argument reduced to that range
// - "two_int_a", which is 2**(int(arg))
// - "adjustment", which is a flag set to zero if the fractional
//               part of arg is <=0.5; set to one otherwise
//
// How this range reduction code works:
//  (1)2**x=2**a * 2**(x-a)
// and,
//  (2)2**x=2**(1/2) * 2**a * 2**(x-a-1/2)
//
// Now, this all looks contrived. The trick is to pick an integer "a"
// such that (x-a) is in the range [0.0, 1.0]. If the result is in
// range we use equation 1. If it is in [0.5, 1.0], use equation (2)
// which will get it down to our desired [0.0, 0.5] range.
//
// The value "adjustment" tells the calling function if we are using
// the first or the second equation.
//

void reduce_expb(double *arg, double *two_int_a, int *adjustment){
  int int_arg;                // integer part of the input argument


  *adjustment=0;              // Assume we're using equation (2)
  int_arg=(int) *arg;
  if((*arg-int_arg) > 0.5)    // if frac(arg) is in [0.5, 1.0]...
     {
       *adjustment=1;
       *arg=*arg-0.5;         // . . . then change it to [0.0, 0.5]
     }
  *arg=*arg-(double) int_arg; // arg is now just fractional part


// Now compute 2** (int) arg.
  *two_int_a=1.0;
   for(; int_arg!=0; int_arg--)*two_int_a=*two_int_a * 2.0;
};
```

*Computer Approximations* lists polynomials for 46 variants of $2^x$, with precisions ranging from 4 to 25 decimal digits. Room (and sanity) prohibits listing them all. But here are two of the most useful examples. Both assume the input argument is between 0 and 0.5, and both use the following ratio of two polynomials:

$$2^x = \frac{Q(x^2) + xP(x^2)}{Q(x^2) - xP(x^2)}$$

The following coefficients yield a precision of 6.36 decimal digits. Note that $P(x)$ is simply an offset, and Q01 is 1, making this a very fast and reasonably accurate approximation:

| | | |
|---|---|---|
| P00 | (+1) | +0.86778 38827 9 |
| Q00 | (+2) | +0.25039 10665 03 |
| Q01 | (+1) | +0.1 |

For 9.85 digits of precision use the somewhat slower:

| | | |
|---|---|---|
| P00 | (+1) | +0.72151 89152 1493 |
| P01 | (−1) | +0.57690 07237 31 |
| Q00 | (+2) | +0.20818 92379 30062 |
| Q01 | (+1) | +0.1 |

Here's an example of code that implements this:

```
//
// expb1063-compute 2**x to 9.85 digits accuracy
//

double expb1063(double arg){

  const double P00 = + 7.2152891521493;
  const double P01 = + 0.0576900723731;
  const double Q00 = +20.8189237930062;
  const double Q01 = + 1.0;

  const double sqrt2= 1.4142135623730950488; // sqrt(2) for scaling
  double two_int_a;               // 2**(int(a)); used to scale
                                  // result
  int adjustment;                 // set to 1 by reduce_expb if
                                  // must adjust the answer by
                                  // sqrt(2)
  double answer;                  // The result
  double Q;                       // Q(x**2)
  double x_P;                     // x*P(x**2)
  int    negative=0;              // 0 if arg is +; 1 if negative

// Return an error if the input is too large. "Too large" is entirely
// a function of the range of your float library and expected inputs
// used in your application.
```

```
  if(abs(arg>100.0)){
     printf("\nYikes! %d is a big number, fella. Aborting.", arg);
     return 0;
  }
//   If the input is negative, invert it. At the end we'll take
// the reciprocal, since n**(−1)=1/(n**x).
  if(arg<0.0)
  {
    arg=-arg;
    negative=1;
  }
  reduce_expb(&arg, &two_int_a, &adjustment); // reduce to [0.0, 0.5]

// The format of the polynomial is:
//  answer=(Q(x**2)+x*P(x**2))/(Q(x**2)-x*P(x**2))
//
//  The following computes the polynomial in several steps:
  Q=    Q00+Q01 * (arg * arg);
  x_P=arg * (P00+P01 * (arg * arg));
  answer= (Q+x_P)/(Q-x_P);

//  Now correct for the scaling factor of 2**(int(a))
  answer= answer * two_int_a;

//  If the result had a fractional part > 0.5, correct for that
  if(adjustment == 1)answer=answer * sqrt2;

// Correct for a negative input
  if(negative == 1) answer=1.0/answer;

  return(answer);
};
```

If you're feeling a need for serious accuracy try the following coefficients, which are good to an astonishing 24.78 digits but need a more complex range reduction algorithm to keep the input between `[0, 1/256]`:

| | |
|---|---|
| P00 (+1) | +0.72134 75314 61762 84602 46233 635 |
| P01 (−1) | +0.57762 26063 55921 17671 75 |
| Q00 (+2) | +0.20813 69012 79476 15341 50743 885 |
| Q01 (+1) | +0.1 |

Note that the polynomials, both of degree 1 only, quickly plop out an answer.

## 4.4.3 Other Exponentials

What about the more common natural and decimal exponentiations? Hart lists many, accurate over a variety of ranges. The most useful seem to be those that work between `[0.0, 0.5]` and `[0.0, 1.0]`. Use a range reduction approach much like that described above, substituting the appropriate base: $10^x = 10^a \times 10^{(x-a)}$ or $e^x = e^a \times e^{(x-a)}$.

Using the same ratio of polynomials listed above the following coefficients give $10^x$ to 12.33 digits:

| | | |
|---|---|---|
| P00 | (+2) | +0.41437 43559 42044 8307 |
| P01 | (+1) | +0.60946 20870 43507 08 |
| P02 | (−1) | +0.76330 97638 32166 |
| Q00 | (+2) | +0.35992 09924 57256 1042 |
| Q01 | (+2) | +0.21195 92399 59794 679 |
| Q02 | (+1) | +0.1 |

The downside of using this or similar polynomials is you've got to solve for that pesky $10^a$. Though a look-up table is one fast possibility if the input stays in a reasonable range, there aren't many other attractive options. If you're working with an antique IBM 1602 BCD machine perhaps it's possible to left shift by factors of 10 in a way analogous to that outlined above for exponents of 2. In today's binary world there are no left-shift decimal instructions so one must multiply by 10.

Instead, use one of the following relations:

$$10^x = 2^{x/\log_{10}2}$$

or

$$e^x = 2^{2/\log_e 2}$$

```
//
// expd_based_on_expb1063-compute 10**x to 9.85 digits accuracy
//
//  This is one approach to computing 10**x. Note that:
// 10**x=2** (x/log_base_10(2)), so
// 10**x=expb(x/log_base_10(2))
//
```

```
double expd_based_on_expb1063(double arg){
   const double log10_2=0.30102999566398119521;

   return(expb1063(arg/log10_2));
}
```

### 4.4.4 Logs

There's little magic to logs, other than a different range reduction algorithm. Again, it's easiest to develop code that works with logarithms of base 2. Note that:

$$\log_2(x) = \log_2(f \times 2^n)$$
$$\log_2(x) = \log_2(f) + \log_2(2^n)$$
$$\log_2(x) = \log_2(f) + n$$

We pick an `n` that keeps `f` between `[0.5, 1.0]`, since Hart's most useful approximations require that range. The reduction algorithm is very similar to that described last issue for square roots, and is here:

```
//  reduce_log2-The log2 routines require an input argument to
// the range[0.5, 1.0].
//
// Return values:
// - "arg", which is the input argument reduced to that range
// - "n", which is n from the discussion below
// - "adjustment", set to 1 if the input was < 0.5
//
// How this range reduction code works:
//  If we pick an integer n such that x=f x 2**n, and 0.5<= f < 1, and
// assume all "log" functions here means log(base 2), then:
// log(x) = log(f x 2**n)
//        = log(f)+ log(2**n)
//        = log(f)+ n
//
//  The "for" loop shifts a one through two_to_n while shifting a long
// version of the input argument right, repeating till the long
// version is all zeros. Thus, two_to_n and the input argument
// have this relationship:
//
```

```
//       two_to_n         input argument       n
//       1                   0.5 to 1          0
//       2                    1 to 2           1
//       4                    2 to 4           2
//       8                    4 to 8           3
// etc.
//
//  There’s a special case for the argument being less than 0.5.
// In this case we use the relation:
//  log(1/x)=log(1)-log(x)
//          =      -log(x)
// i.e., take the reciprocal of x (which is bigger than 0.5) and solve
// for the above. To tell the caller to do this, set “adjustment”=1.
//

void reduce_log2(double *arg, double *n, int *adjustment){

  long two_to_n;                //divisor we’re looking for: 2^(2k)
  long l_arg;                   //long (32 bit) version of input

  *adjustment=0;
  if(*arg<0.5){                 //if small arg use the reciprocal
     *arg=1.0/(*arg);
     *adjustment=1;
  }

  //shift arg to zero while computing two_to_n as described above
  l_arg=(long) *arg;            //l_arg is long version of input
  for(two_to_n=1, *n=0.0; l_arg!=0; l_arg>>=1, two_to_n<<=1, *n+=1.0);
  *arg=*arg/(double)two_to_n;   //normalize input to [0.5, 1]

};
```

This code uses the above to compute log(base 2)(*x*) to 4.14 digits of accuracy:

```
//
// LOG2_2521-compute log(base 2)(x) to 4.14 digits accuracy
//
  double log2_2521(double arg){
  const double P00=-1.45326486;
  const double P01=+0.951366714;
  const double P02=+0.501994886;
  const double Q00=+0.352143751;
  const double Q01=+1.0;

  double n;                              // used to scale result
  double poly;                           // result from polynomial
  double answer;                         // The result
  int adjustment;                        // 1 if argument was < 0.5

// Error if the input is <=0 since log is not real at and below 0.
  if(arg<=0.0){
     printf("\nHoly smokes! %d is too durn small. Aborting.", arg);
     return 0;
  }
  reduce_log2(&arg, &n, &adjustment); // reduce input to [0.5, 1.0]

// The format of the polynomial is P(x)/Q(x)
  poly= (P00+arg * (P01+arg * P02))/(Q00+Q01 * arg);

// Now correct for the scaling factors
  if(adjustment)answer=-n-poly;
  else answer=poly+n;
  return(answer);
};
```

Need better results? The following coefficients are good to 8.32 digits over the same range:

| | | |
|---|---|---|
| P00 | (+1) | −0.20546 66719 51 |
| P01 | (+1) | −0.88626 59939 1 |
| P02 | (+1) | +0.61058 51990 15 |
| P03 | (+1) | +0.48114 74609 89 |
| Q00 | (+0) | +0.35355 34252 77 |
| Q01 | (+1) | +0.45451 70876 29 |
| Q02 | (+1) | +0.64278 42090 29 |
| Q03 | (+1) | +0.1 |

Though Hart's book includes plenty of approximations for common and natural logs, it's probably more efficient to compute $\log_2$ and use the change of base formulas:

$$\log_{10}(x) = \frac{\log_2(x)}{\log_2(10)}$$

and

$$\ln(x) = \frac{\log_2(x)}{\log_2(\mathrm{e})}$$

### 4.4.5 Trig: General Notes

We generally work in radians rather than degrees. The 360° in a circle are equivalent to $2\pi$ radians; thus, one radian is $360/(2\pi)$, or about 57.3°. This may seem a bit odd until you think of the circle's circumference, which is $2\pi r$; if $r$ (the circle's radius) is one, the circumference is indeed $2\pi$.

The conversions between radians and degrees are:

Angle in radians = angle in degrees × $2\pi/360$

Angle in degrees = angle in radians × $360/(2\pi)$

| Degrees | Radians | Sine | Cosine | Tangent |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 45 | $\pi/4$ | $\sqrt{2}/2$ | $\sqrt{2}/2$ | 1 |
| 90 | $\pi/2$ | 1 | 0 | Infinity |
| 135 | $3\pi/4$ | $\sqrt{2}/2$ | $-\sqrt{2}/2$ | −1 |
| 180 | $\pi$ | 0 | −1 | Infinity |
| 225 | $5\pi/4$ | $-\sqrt{2}/2$ | $-\sqrt{2}/2$ | 1 |
| 270 | $3\pi/2$ | −1 | 0 | Infinity |
| 315 | $7\pi/4$ | $-\sqrt{2}/2$ | $\sqrt{2}/2$ | −1 |
| 360 | $2\pi$ | 0 | 1 | 0 |

### 4.4.6 Cosine and Sine

The following examples all approximate the cosine function; sine is derived from cosine via the relationship:

$$\sin(x) = \cos(\pi/2 - x)$$

In other words, the sine and cosine are the same function, merely shifted 90º in phase. The sine code is (assuming we're calling `cos_32`, the lowest accuracy cosine approximation):

```
//   The sine is just cosine shifted a half-pi, so
// we'll adjust the argument and call the cosine approximation.
//
float sin_32(float x){
      return cos_32(halfpi-x);
}
```

All of the cosine approximations in this chapter compute the cosine accurately over the range of 0 to $\pi/2$ (0–90º). That surely denies us most of the circle! Approximations in general work best over rather limited ranges; it's up to us to reduce the input range to something the approximation can handle accurately.

Therefore, before calling any of the following cosine approximations we assume the range has been reduced to 0 to $\pi/2$ using the following code:

```
// Math constants
double const pi=3.1415926535897932384626433;// pi
double const twopi=2.0*pi;          // pi times 2
double const halfpi=pi/2.0;         // pi divided by 2
//
//   This is the main cosine approximation "driver"
// It reduces the input argument's range to [0, pi/2],
// and then calls the approximator.
//
float cos_32(float x){
      int quad;                     // what quadrant are we in?

      x=fmod(x, twopi);             // **Get rid of values > 2* pi
      if(x<0)x=-x;                  // **cos(-x)=cos(x)
```

```
        quad=int(x/halfpi);       // Get quadrant # (0 to 3)    switch (quad){
        case 0: return cos_32s(x);
        case 1: return -cos_32s(pi-x);
        case 2: return -cos_32s(x-pi);
        case 3: return cos_32s(twopi-x);
        }
}
```

This code is configured to call `cos_32s`, which is the approximation (detailed shortly) for computing the cosine to 3.2 digits accuracy. Use this same code, though, for all cosine approximations; change `cos_32s` to `cos_52s`, `cos_73s`, or `cos_121s`, depending on which level of accuracy you need. See the complete listing for a comprehensive example.

If you can guarantee that the input argument will be greater than zero and less than $2\pi$, delete the two lines in the listing above which have "`**`" in the comments to get even faster execution.

Be clever about declaring variables and constants. Clearly, working with the `cos_32` approximation nothing must be declared "`double`". Use `float` for more efficient code. Reading the complete listing you'll notice that for `cos_32` and `cos_52` we used floats everywhere; the more accurate approximations declare things as doubles.

One trick that will speed up the approximations is to compute $x^2$ by incrementing the characteristic of the floating point representation of $x$. You'll have to know exactly how the numbers are stored, but you can save hundreds of microseconds over performing the much clearer "`x*x`" operation.

How does the range reduction work? Note that the code divides the input argument into one of four "quadrants"—the very same quadrants of the circle shown in Figure 4.14.

### *4.4.6.1 Quadrants 0 to 3 of the circle*

- For the first quadrant (0 to $\pi/2$) there's nothing to do since the cosine approximations are valid over this range.
- In quadrant 1 the cosine is symmetrical with quadrant 0, if we reduce its range by subtracting the argument from $\pi$. The cosine, though, is negative for quadrants 1 and 2 so we compute `-cos(`$\pi$`-x)`.

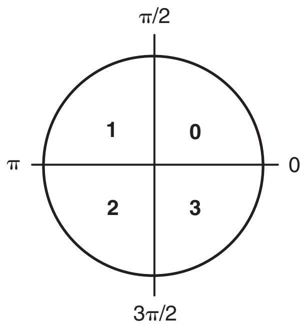


**Figure 4.14: Quadrants 0 to 3 of the circle**

- Quadrant 2 is similar to 1.
- Finally, in 3 the cosine goes positive again; if we subtract the argument from 2p it translates back to something between 0 and p/2.

The approximations do convert the basic polynomial to a simpler, much less computationally expensive form, as described in the comments. All floating point operations take appreciable amounts of time, so it's important to optimize the design. For the trig functions you'll find the calling functions as well as the range reduction functions to illustrate ways of optimizing.

```
//              cos_32s computes cosine (x)
//
// Accurate to about 3.2 decimal digits over the range [0, pi/2].
// The input argument is in radians.
//
// Algorithm:
//              cos(x)= c1+c2*x**2+c3*x**4
//  which is the same as:
//              cos(x)= c1+x**2(c2+c3*x**2)
//
float cos_32s(float x)
{
const float c1= 0.99940307;
const float c2=-0.49558072;
const float c3= 0.03679168;
float x2;                          // The input argument squared
x2=x * x;
return (c1+x2*(c2+c3 * x2));
}
```

cos_32 computes a cosine to about 3.2 decimal digits of accuracy. Use the range reduction code (listed earlier) if the range exceeds 0 to π/2. The plotted errors are absolute (not percent error) (Figure 4.15).

cos_52 computes a cosine to about 5.2 decimal digits of accuracy. Use the range reduction code (listed earlier) if the range exceeds 0 to π/2. The plotted errors are absolute (not percent error) (Figure 4.16).

```
//        cos_52s computes cosine (x)
//
//  Accurate to about 5.2 decimal digits over the range [0, pi/2].
//  The input argument is in radians.
//
//  Algorithm:
//        cos(x)= c1+c2*x**2+c3*x**4+c4*x**6
//  which is the same as:
//        cos(x)= c1+x**2(c2+c3*x**2+c4*x**4)
//        cos(x)= c1+x**2(c2+x**2(c3+c4*x**2))
//
float cos_52s(float x)
{
const float c1= 0.9999932946;
const float c2=-0.4999124376;
const float c3= 0.0414877472;
const float c4=-0.0012712095;
float x2;                             // The input argument squared
x2=x * x;
return (c1+x2*(c2+x2*(c3+c4*x2)));
}
```

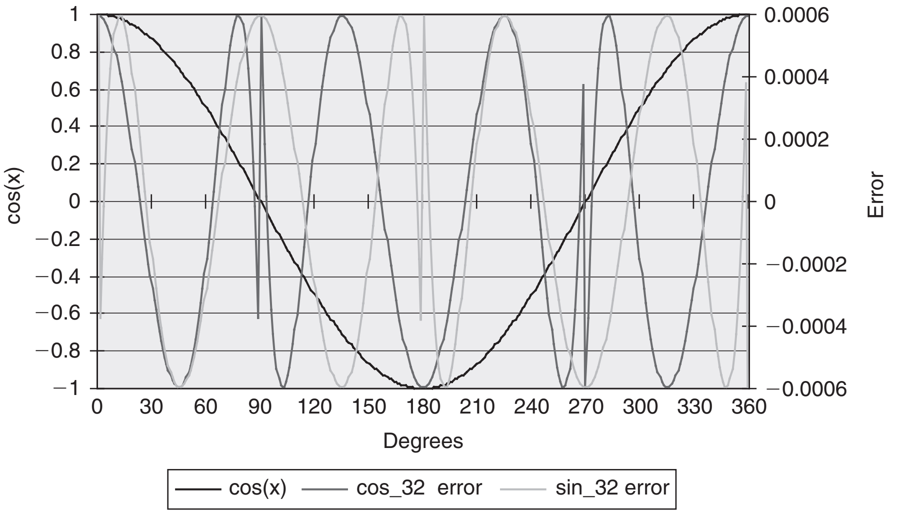


Figure 4.15: cos_32 and sin_32 error

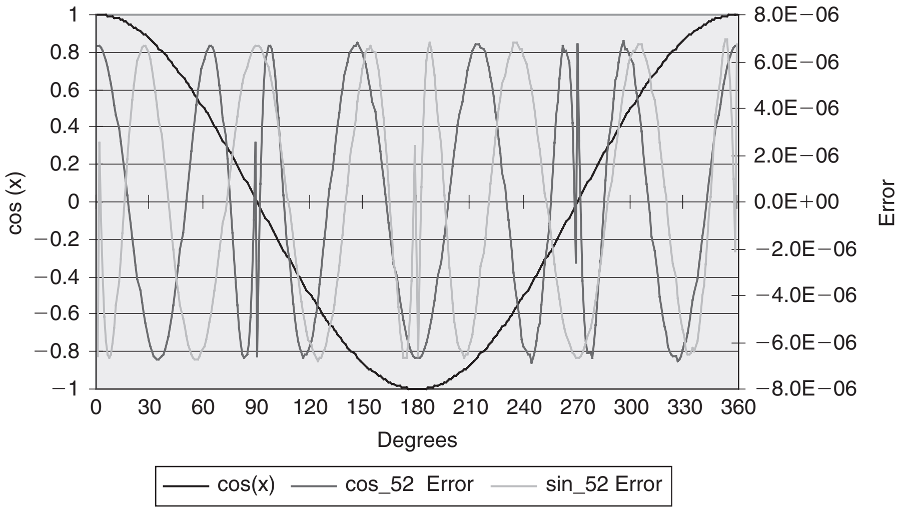


Figure 4.16: cos_52 and sin_52 error

```
//          cos_73s computes cosine (x)
//
// Accurate to about 7.3 decimal digits over the range [0, pi/2].
// The input argument is in radians.
//
// Algorithm:
//          cos(x)= c1+c2*x**2+c3*x**4+c4*x**6+c5*x**8
//  which is the same as:
//          cos(x)= c1+x**2(c2+c3*x**2+c4*x**4+c5*x**6)
//          cos(x)= c1+x**2(c2+x**2(c3+c4*x**2+c5*x**4))
//          cos(x)= c1+x**2(c2+x**2(c3+x**2(c4+c5*x**2)))
//
double cos_73s(double x)
{
const double c1= 0.999999953464;
const double c2=-0.4999999053455;
const double c3= 0.0416635846769;
const double c4=-0.0013853704264;
const double c5= 0.000023233 ;  // Note: this is a better coefficient
                                // than Hart's
                                // from Steven Perkins
double x2;                      // The input argument squared
x2=x * x;
return (c1+x2*(c2+x2*(c3+x2*(c4+c5*x2))));
}
```

`cos_73` computes a cosine to about 7.3 decimal digits of accuracy. Use the range reduction code (listed earlier) if the range exceeds 0 to $\pi/2$. Also plan on using double precision math for the range reduction code to avoid losing accuracy. The plotted errors are absolute (not percent error) (Figure 4.17).

```
//           cos_121s computes cosine (x)
//
// Accurate to about 12.1 decimal digits over the range [0, pi/2].
// The input argument is in radians.
//
```

```
// Algorithm:
//      cos(x)= c1+c2*x**2+c3*x**4+c4*x**6+c5*x**8+c6*x**10+c7*x**12
//  which is the same as:
//      cos(x)= c1+x**2(c2+c3*x**2+c4*x**4+c5*x**6+c6*x**8+c7*x**10)
//      cos(x)= c1+x**2(c2+x**2(c3+c4*x**2+c5*x**4+c6*x**6+c7*x**8 ))
//      cos(x)= c1+x**2(c2+x**2(c3+x**2(c4+c5*x**2+c6*x**4+c7*x**6 )))
//      cos(x)= c1+x**2(c2+x**2(c3+x**2(c4+x**2(c5+c6*x**2+c7*x**4 ))))
//      cos(x)= c1+x**2(c2+x**2(c3+x**2(c4+x**2(c5+x**2(c6+c7*x**2 )))))
//
double cos_121s(double x)
{
const double c1= 0.99999999999925182;
const double c2=-0.49999999997024012;
const double c3= 0.041666666473384543;
const double c4=-0.001388888418000423;
const double c5= 0.0000248010406484558;
const double c6=-0.0000002752469638432;
const double c7= 0.0000000019907856854;
double x2;                          // The input argument squared

x2=x * x;
return (c1+x2*(c2+x2*(c3+x2*(c4+x2*(c5+x2*(c6+c7*x2))))));
}
```

`cos_121` computes a cosine to about 12.1 decimal digits of accuracy. Use the range reduction code (listed earlier) if the range exceeds 0 to $\pi/2$. Also plan on using double precision math for the range reduction code to avoid losing accuracy. The plotted errors are absolute (not percent error) (Figure 4.18).

## *4.4.7 Higher Precision Cosines*

Given a large enough polynomial there's no limit to the possible accuracy. A few more algorithms are listed here. These are all valid for the range of 0 to p/2, and all can use the previous range reduction algorithm to change any angle into one within this range. All take an input argument in radians.

No graphs are included because these exceed the accuracy of the typical compiler's built-in cosine function … so there's nothing to plot the data against.

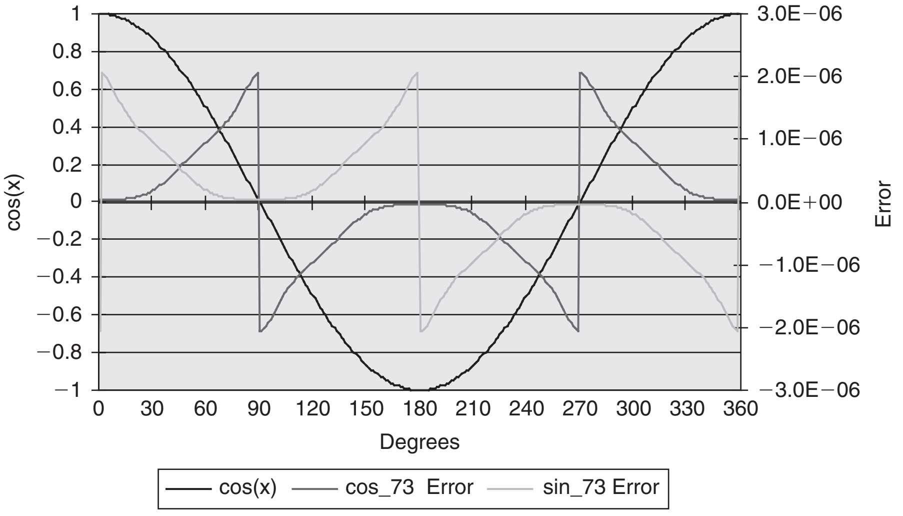


**Figure 4.17: cos_73 and sin_73 error**

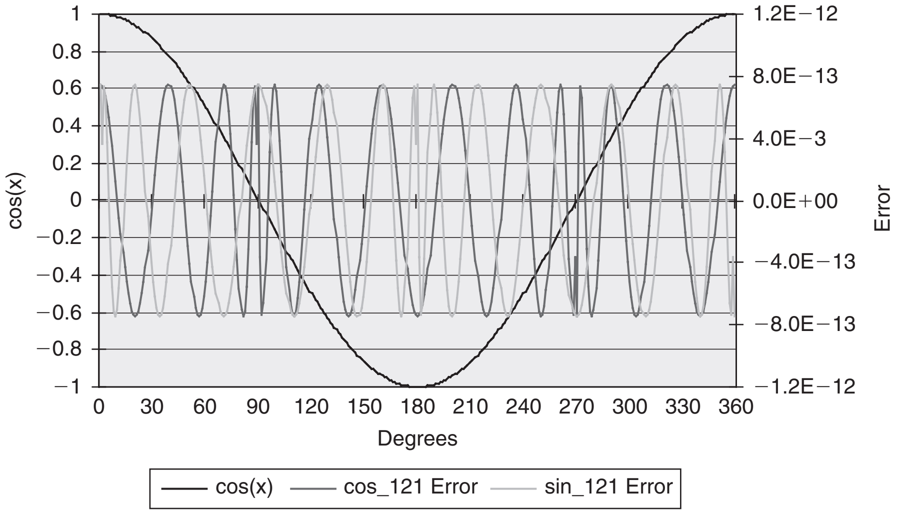


**Figure 4.18: cos_131 and sin_121 error**

Note that C's `double` type on most computers carries about 15 digits of precision. So for these algorithms, especially for the 20.2 and 23.1 digit versions, you'll need to use a data type that offers more bits. Some C's support a `long double`. But check the manual carefully! Microsoft's Visual C++, for instance, while it does support the `long double` keyword, converts all of these to `double`.

Accurate to about 14.7 decimal digits over the range [0, $\pi/2$]:

```
c1= 0.99999999999999806767
c2=-0.4999999999998996568
c3= 0.04166666666581174292
c4=-0.001388888886113613522
c5= 0.000024801582876042427
c6=-0.0000002755693576863181
c7= 0.0000000020858327958707
c8=-0.000000000011080716368
cos(x)= c1+x^2(c2+x^2(c3+x^2(c4+x^2(c5 +
           x^2(c6+x^2(c7+x^2*c8))))))
```

Accurate to about 20.2 decimal digits over the range [0, $\pi/2$]:

```
 c1= 0.9999999999999999999936329
 c2=-0.49999999999999999948362843
 c3= 0.04166666666666665975670054
 c4=-0.00138888888888885302082298
 c5= 0.000024801587301492746422297
 c6=-0.00000027557319209666748555
 c7= 0.0000000020876755667423458605
 c8=-0.0000000000114706701991777771
 c9= 0.0000000000000477687298095717
c10=-0.00000000000000015119893746887
cos(x)= c1+x^2(c2+x^2(c3+x^2(c4+x^2(c5+x^2(c6 +
           x^2(c7+x^2(c8+x^2(c9+x^2*c10))))))))
```

Accurate to about 23.1 decimal digits over the range [0, $\pi/2$]:

```
c1= 0.9999999999999999999999914771
c2=-0.4999999999999999999991637437
c3= 0.04166666666666666665319411988
c4=-0.00138888888888888880310186415
```

```
 c5= 0.00002480158730158702330045157
 c6=-0.000000275573192239332256421489
 c7= 0.000000002087675698165412591559
 c8=-0.0000000000114707451267755432394
 c9= 0.00000000000004779454394066499917
c10=-0.0000000000000001561226342882778l
c11= 0.00000000000000000039912654507924
cos(x)=c1+x^2(c2+x^2(c3+x^2(c4+x^2(c5+x^2(c6 +
            x^2(c7+x^2(c8+x^2(c9+x^2(c10+x^2*c11)))))))))
```

## 4.4.8 Tangent

The tangent of an angle is defined as `tan(x)=sin(x)/cos(x)`. Unhappily this is not the best choice, though, for doing an approximation. As cos($x$) approaches zero the errors propagate rapidly. Further, at some points like $\pi/4$ (see the previous graphs of sine and cosine errors) the errors of sine and cosine reinforce each other; both are large and have the same sign.

So we're best off using a separate approximation for the tangent. All of the approximations we'll use generate a valid tangent for angles in the range of 0 to $\pi/4$ (0–45º), so once again a range reduction function will translate angles to this set of values.

```
//
// This is the main tangent approximation "driver"
// It reduces the input argument's range to [0, pi/4],
// and then calls the approximator.
// Enter with positive angles only.
//
// WARNING: We do not test for the tangent approaching infinity,
// which it will at x=pi/2 and x=3*pi/2. If this is a problem
// in your application, take appropriate action.
//
float tan_32(float x){
      int octant;                       // what octant are we in?

      x=fmod(x, twopi);                 // Get rid of values .2 *pi
      octant=int(x/qtrpi);              // Get octant # (0 to 7)
      switch (octant){
      case 0: return      tan_32s(x            *four_over_pi);
      case 1: return  1.0/tan_32s((halfpi-x)    *four_over_pi);
```

```
        case 2: return  -1.0/tan_32s((x-halfpi)      *four_over_pi);
        case 3: return  -    tan_32s((pi-x)          *four_over_pi);
        case 4: return       tan_32s((x-pi)          *four_over_pi);
        case 5: return   1.0/tan_32s((threehalfpi-x  *four_over_pi);
        case 6: return  -1.0/tan_32s((x-threehalfpi) *four_over_pi);
        case 7: return  -    tan_32s((twopi-x)       *four_over_pi);
        }
}
```

The code above does the range reduction and then calls `tan_32`. When using the higher precision approximations substitute the appropriate function name for `tan_32`.

The reduction works much like that for cosine, except that it divides the circle into octants and proceeds from there. One quirk is that the argument is multiplied by $4/\pi$. This is because the approximations themselves actually solve for $\tan((\pi/4)x)$.

The listings that follow give the algorithms needed.

Remember that tan(90) and tan(270) both equal infinity. As the input argument gets close to 90 or 270 the value of the tangent skyrockets, as illustrated in Figure 4.19. *Never take a tangent close to 90º or 270º!*

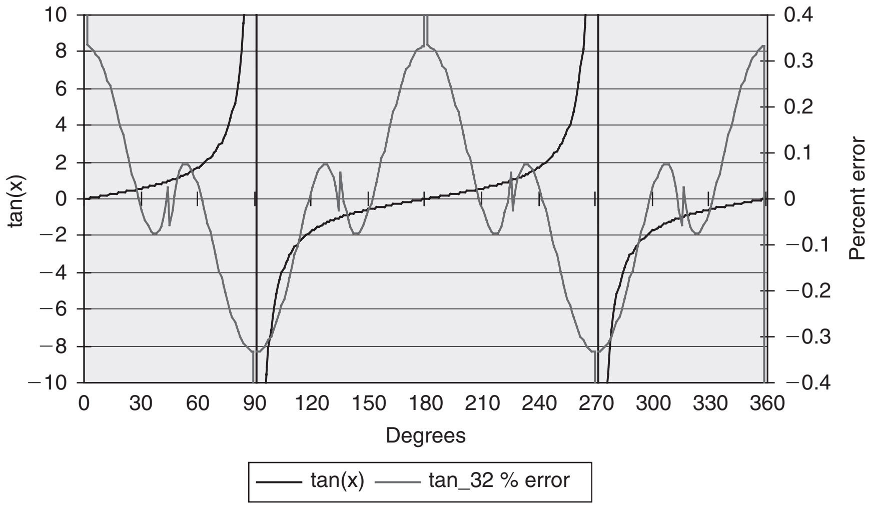


**Figure 4.19: tan_32 error**

```
// ********************************************************
// ***
// ***    Routines to compute tangent to 3.2 digits
// ***   of accuracy.
// ***
// ********************************************************
//
//              tan_32s computes tan(pi*x/4)
//
// Accurate to about 3.2 decimal digits over the range [0, pi/4].
// The input argument is in radians. Note that the function
// computes tan(pi*x/4), NOT tan(x); it's up to the range
// reduction algorithm that calls this to scale things properly.
//
// Algorithm:
//             tan(x)= x*c1/(c2+x**2)
//
float tan_32s(float x)
{
const float c1=-3.6112171;
const float c2=-4.6133253;
float x2;                                  // The input argument squared

x2=x * x;
return (x*c1/(c2+x2));
}
```

`tan_32` computes the tangent of $\pi/4 \times x$ to about 3.2 digits of accuracy. Use the range reduction code to translate the argument to 0 to $\pi/4$, and of course to compensate for the peculiar "$\pi/4$" bias required by this routine. Note that the graphed errors are percentage error, not absolute.

```
// ********************************************************
// ***
// ***    Routines to compute tangent to 5.6 digits
// ***   of accuracy.
// ***
// ********************************************************
```

```
//
//              tan_56s computes tan(pi*x/4)
//
// Accurate to about 5.6 decimal digits over the range [0, pi/4].
// The input argument is in radians. Note that the function
// computes tan(pi*x/4), NOT tan(x); it's up to the range
// reduction algorithm that calls this to scale things properly.
//
//  Algorithm:
//              tan(x)= x(c1+c2*x**2)/(c3+x**2)
//
float tan_56s(float x)
{
const float c1=-3.16783027;
const float c2= 0.134516124;
const float c3=-4.033321984;
float x2;                                // The input argument squared

x2=x * x;
return (x*(c1+c2 * x2)/(c3+x2));
}
```

`tan_56` computes the tangent of $\pi/4 \times x$ to about 5.6 digits of accuracy. Use the range reduction code to translate the argument to 0 to $\pi/4$, and of course to compensate for the peculiar "$\pi/4$" bias required by this routine. Note that the graphed errors are percentage error, not absolute (Figure 4.20).

```
// *********************************************************
// ***
// ***    Routines to compute tangent to 8.2 digits
// ***   of accuracy.
// ***
// *********************************************************
//
//              tan_82s computes tan(pi*x/4)
//
// Accurate to about 8.2 decimal digits over the range [0, pi/4].
// The input argument is in radians. Note that the function
// computes tan(pi*x/4), NOT tan(x); it's up to the range
// reduction algorithm that calls this to scale things properly.
//
// Algorithm:
//             tan(x)= x(c1+c2*x**2)/(c3+c4*x**2+x**4)
//
double tan_82s(double x)
{
const double c1= 211.849369664121;
const double c2=- 12.5288887278448 ;
const double c3= 269.7350131214121;
const double c4=- 71.4145309347748;

double x2;                               // The input argument squared

x2=x * x;
return (x*(c1+c2 * x2)/(c3+x2*(c4+x2)));
}
```

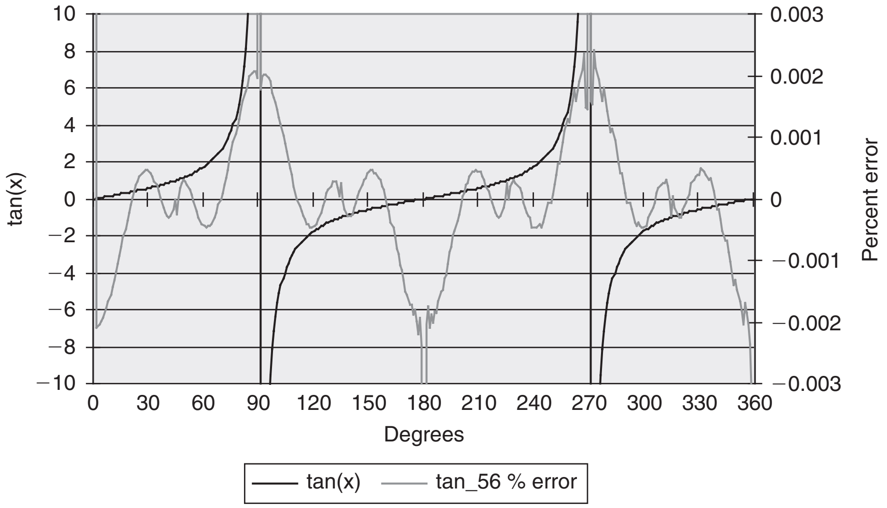


**Figure 4.20: tan_56 error**

`tan_82` computes the tangent of $\pi/4 \times x$ to about 8.2 digits of accuracy. Use the range reduction code to translate the argument to 0 to $\pi/4$, and of course to compensate for the peculiar "$\pi/4$" bias required by this routine. Note that variables are declared as "double." The graphed errors are percentage error, not absolute (Figure 4.21).

```
// *****************************************************
// ***
// ***    Routines to compute tangent to 14 digits
// ***   of accuracy.
// ***
// *****************************************************
//
//             tan_14s computes tan(pi*x/4)
//
// Accurate to about 14 decimal digits over the range [0, pi/4].
// The input argument is in radians. Note that the function
// computes tan(pi*x/4), NOT tan(x); it's up to the range
// reduction algorithm that calls this to scale things properly.
//
// Algorithm:
//             tan(x)= x(c1+c2*x**2+c3*x**4)/(c4+c5*x**2+c6*x**4+x**6)
//
double tan_14s(double x)
{
const double c1=-34287.4662577359568109624;
const double c2= 2566.7175462315050423295;
const double c3=-26.5366371951731325438;
const double c4=-43656.1579281292375769579;
const double c5= 12244.4839556747426927793;
const double c6=-336.611376245464339493;
double x2;                              // The input argument squared

x2=x * x;
return (x*(c1+x2*(c2+x2*c3))/(c4+x2*(c5+x2*(c6+x2))));
}
```

`tan_141` computes the tangent of $\pi/4\times x$ to about 14.1 digits of accuracy. Use the range reduction code to translate the argument to 0 to $\pi/4$, and of course to compensate for the peculiar "$\pi/4$" bias required by this routine. Note that variables are declared as "double." The graphed errors are percentage error, not absolute (Figure 4.22).

### 4.4.9 Higher Precision Tangents

Given a large enough polynomial there's no limit to the possible accuracy. A few more algorithms are listed here. These are all valid for the range of 0 to $\pi/4$, and all should use

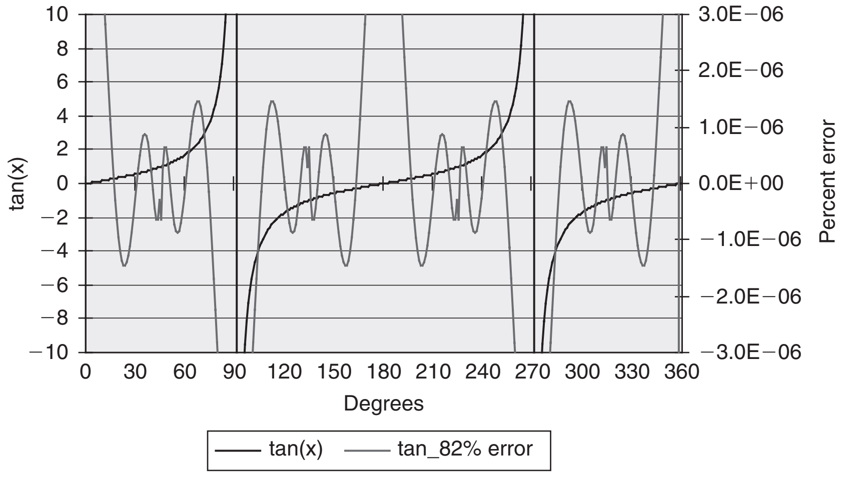


Figure 4.21: tan_82 error

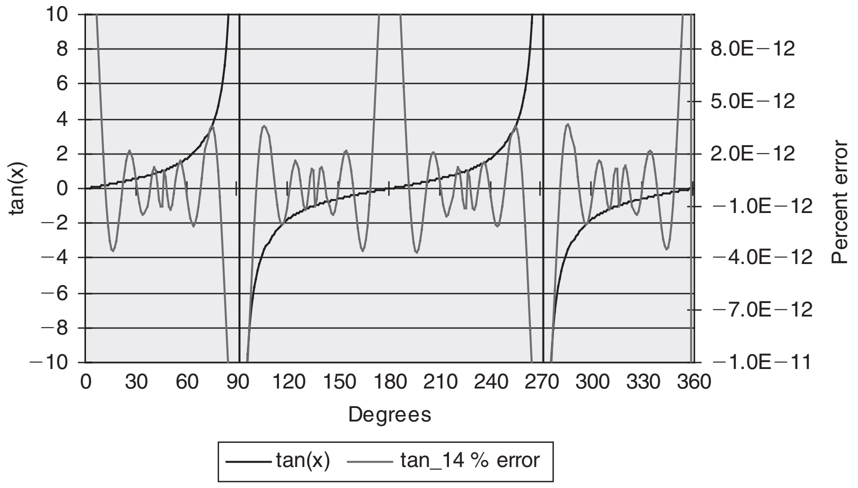


Figure 4.22: tan_14 error

the previous range reduction algorithm to change any angle into one within this range. All take an input argument in radians, though it is expected to be mangled by the $\pi/4$ factor. The prior range reducer will correct for this.

No graphs are included because these exceed the accuracy of the typical compiler's built-in cosine function … so there's nothing to plot the data against.

As noted before, C's `double` type on most computers carries about 15 digits of precision. So for these algorithms, especially for the 20.2 and 23.1 digit versions, you'll need to use a data type that offers more bits. Some C's support a `long double`. But check the manual carefully: Microsoft's Visual C++, for instance, while it does support the `long double` keyword, converts all of these to `double`.

Accurate to about 20.3 digits over the range of 0 to π/4:

```
c1= 10881241.46289544215469695742
c2=-  895306.0870564145957447087575
c2=    14181.99563014366386894487566
c3=-      45.63638305432707847378129653
c4= 13854426.92637036839270054048
c5=- 3988641.468163077300701338784
c6=   135299.4744550023680867559195
c7=-    1014.19757617656429288596025
tan(xπ/4)=x(c1+x²(c2+x²(c31x²*c4)))
           /(c5+x²(c6+x²(c7+x²)))
```

Accurate to about 23.6 digits over the range of 0 to π/4:

```
c1= 4130240.558996024013440146267
c2=- 349781.8562517381616631012487
c3=    6170.317758142494245331944348
c4=-     27.94920941480194872760036319
c5=       0.0175143807040383602666563058
c6= 5258785.647179987798541780825
c7=-1526650.549072940686776259893
c8=   54962.51616062905361152230566
c9=-    497.495460280917265024506937
tan(xπ/4)=x(c1+x²(c2+x²(c3+x²(c4+x²*c5))))
          /(c6+x²(c7+x²(c8+x²(c9+x²))))
```

### 4.4.10 Arctangent, Arcsine, and Arccosine

The arctangent is the same as the inverse tangent, so arctan(tan(x))=x. It's often denoted as "atan(x)" or "$\tan^{-1}$(x)".

In practice the approximations for inverse sine and cosine aren't too useful; mostly we derive these from the arctangent as follows:

```
Arcsine(x)    = atan(x/√(1-x²))
Arccosine(x)  = π/2—arcsine(x)
              = π/2—atan(x/√(1—x²))
```

The approximations are valid for the range of 0 to $\pi/12$. The following code, based on that by Jack Crenshaw in his *Math Toolkit for Real-Time Programming*, reduces the range appropriately:

```
//
// This is the main arctangent approximation "driver"
// It reduces the input argument's range to [0, pi/12],
// and then calls the approximator.
//
//
double atan_66(double x){
double y;                                 // return from atan__s function
int complement= FALSE;                    // true if arg was >1
int region= FALSE;                        // true depending on region arg is in
int sign= FALSE;                          // true if arg was < 0

if (x <0 ){
      x=-x;
      sign=TRUE;                          // arctan(-x)=-arctan(x)
}
if (x > 1.0){
      x=1.0/x;                            // keep arg between 0 and 1
      complement=TRUE;
}
if (x > tantwelfthpi){
      x=(x-tansixthpi)/(1+tansixthpi*x);  // reduce arg to under tan(pi/12)
      region=TRUE;
}

y=atan_66s(x);                            // run the approximation
if (region) y+=sixthpi;                   // correct for region we're in
if (complement)y=halfpi-y;                // correct for 1/x if we did that
if (sign)y=-y;                            // correct for negative arg
return (y);
}
```

```
// ********************************************************
// ***
// ***    Routines to compute arctangent to 6.6 digits
// ***   of accuracy.
// ***
// ********************************************************
//
//              atan_66s computes atan(x)
//
// Accurate to about 6.6 decimal digits over the range [0, pi/12].
//
// Algorithm:
//              atan(x)= x(c1+c2*x**2)/(c3+x**2)
//
double atan_66s(double x)
{
const double c1=1.6867629106;
const double c2=0.4378497304;
const double c3=1.6867633134;

double x2;                               // The input argument squared

x2=x * x;
return (x*(c1+x2*c2)/(c3+x2));
}
```

`atan_66` computes the arctangent to about 6.6 decimal digits of accuracy using a simple rational polynomial. Its input range is 0 to $\pi$ /12; use the previous range reduction code (Figure 4.23).

```
// ********************************************************
// ***
// ***    Routines to compute arctangent to 13.7 digits
// ***   of accuracy.
// ***
// ********************************************************
//
//              atan_137s computes atan(x)
//
// Accurate to about 13.7 decimal digits over the range [0, pi/12].
```

```
//
// Algorithm:
//             atan(x)= x(c1+c2*x**2+c3*x**4)/(c4+c5*x**2+c6*x**4+x**6)
//
double atan_137s(double x)
{
const double c1= 48.70107004404898384;
const double c2= 49.5326263772254345;
const double c3= 9.40604244231624;
const double c4= 48.70107004404996166;
const double c5= 65.7663163908956299;
const double c6= 21.587934067020262;

double x2;                          // The input argument squared

x2=x * x;
return (x*(c1+x2*(c2+x2*c3))/(c4+x2*(c5+x2*(c6+x2))));
}
```

atan_137 computes the arctangent to about 13.7 decimal digits of accuracy using a simple rational polynomial. Its input range is 0 to π /12; use the previous range reduction code (Figure 4.24).

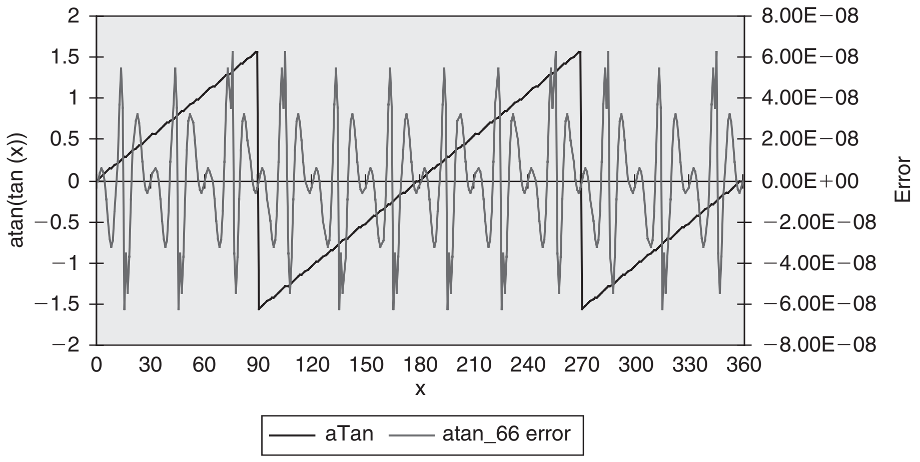


**Figure 4.23: atan_66 error**

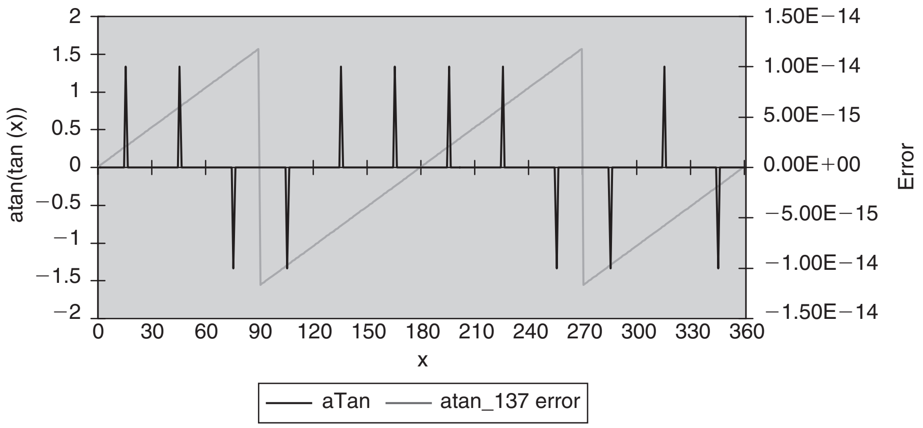


**Figure 4.24: atan_137 error**

## 4.4.11 Precision

So far I've talked about the approximations' precision imprecisely. What exactly does precision mean? Hart chooses to list *relative* precision for roots and exponentials:

$$P = abs\left(\frac{approximated\_result - correct\_answer}{correct\_answer}\right)$$

Logarithms are a different game altogether; he uses absolute precision, defined by:

$P=abs(approximated\_result\_correct\_answer)$

To figure decimal digits of precision use:

$Digits=2 - \log_{10}(P)$

This page intentionally left blank

CHAPTER 5

# The Real World

## 5.1 Electromagnetics for Firmware People

Programming classes and circuit theory were a breeze for me as an EE student at the University of Maryland many years ago. Math was a bit more of a struggle, especially the more advanced classes like abstract algebra. But the two required electromagnetics classes were killers.

I had no idea what the professors were babbling about. Orthogonal E and B fields sounded pretty simple until instantiated into the baffling form of Maxwell's Laws.

Maxwell said a lot of profound things, like:

$$\nabla \times E = \mu_0 \varepsilon_0 \frac{\partial \mathbf{B}}{\partial t}$$

I'm told that this is a thing of beauty, a concise expression that encapsulates complex concepts into a single tidy formula.

Despite two semesters wrestling with this and similar equations, to this day I have no idea what it means. We learned to do curls and circular integrals until our eyes bugged out, but beyond the mechanical cranking of numbers never quite got the intent hidden in those enigmatic chunks of calculus.

All aspiring RF engineers missed many parties in their attempts not only to pass the classes, but to actually understand the material. Secure in the knowledge that I planned a career in the digital domain I figured a bare passing grade would be fine. Digital ICs of the time switched in dozens of nanoseconds. Even the school's $10 million Univac ran at a not-so-blistering 1.25 MHz. At those speeds electrons behave.

Not long after college IC fabrication techniques started to improve at a dramatic rate. Speeds crept to the tens of MHz, and then to the billions of Hertz. Gate switching times plummeted to intervals now measured in hundreds of picoseconds.

Ironically, now electromagnetics is the foundational science of all digital engineering. Even relatively slow 8-bit systems have high-speed signals propagating all around the printed circuit board, signals whose behavior is far more complex than the pristine nirvana of an idealized zero or one.

I should have studied harder.

## 5.1.1 Speed Kills

Most of us equate a fast clock with a fast system. That's fine for figuring out how responsive your World of Warcraft game will be, or if Word will load in less than a week. In digital systems, though, fast systems are those where gates switch between a zero and a one very quickly.

In 1822 Frenchman Jean Baptiste Joseph Fourier showed that any function can be expressed as the sum of sine waves. Turns out he was wrong; his analysis applies (in general) only to *periodic* functions, those that repeat endlessly.

The Fourier series for a function $f(x)$ is:

$$f(x) = a_0 + \sum_{n=1}^{\infty}[a_n\cos(nx) + b_n\sin(nx)]$$

The coefficients $a$ and $b$ are given by mathematical wizardry not relevant to this discussion. It's clear that when $n$ gets huge, if $a$ and/or $b$ is non-zero then $f(x)$ has frequency components in the gigahertz, terahertz, or higher.

What does this mean to us?

Digital circuits shoot pulse streams of all sorts around the PCB. Most, especially the system clock, look a lot like square waves, albeit waves that are somewhat distorted due to the imperfection of all electronic components. The Fourier series for a square wave—a representation of that signal using sines and cosines—has frequency components out to infinity. In other words, if we could make a perfect clock, an idealized square wave, even systems with slow clock rates would have frequency components up to infinite Hertz racing around the board.

In the real world we can't make a flawless square wave. It takes time for a gate to transition from a zero to a one. The Fourier series for a signal that slowly drools between zero and one has low frequency components. As the transition speeds up, frequencies escalate. Fast edges create high frequency signals much faster than high clock rates.

How bad is this? An approximate rule of thumb tells us most of the energy from logic transitions stays below frequency $F$ given the "rise time" (or time to switch from a zero to a one) $T_r$:

$$F = \frac{0.5}{T_r}$$

Older logic like TI's CD74HC240 sport a rise time of 18 ns or so. Little of the energy from Fourier effects will exceed 28 MHz. But don't be surprised to find measurable frequency components in the tens of MHz … even for a system with a 1 Hz clock!

The same component in more modern CY74FCT240T guise switches in a nanosecond or less. The energy cutoff is around 500 MHz, regardless of clock speed.

### 5.1.2 Bouncing

Why do we care about these high frequencies that race around our computer board that sputters around at a mere handful of MHz?

At DC a wire or PCB track has only one parameter of interest: resistance. Over short runs that's pretty close to zero, so for logic circuits a wire is nearly a perfect conductor.

Crank up the frequency, though, and strange things start to happen. That simple wire or track now looks rather like a capacitor and an inductor. Capacitors and inductors exhibit a form of resistance to alternating current called *reactance*. The reactance of a capacitor is:

$$X_c = \frac{1}{2\pi FC}$$

where $F$ is the frequency and $C$ is the capacitance. So as the frequency goes up, the reactance goes down. Caps block DC (the reactance is infinite) and pass AC. An inductor has exactly the opposite characteristic.

The wire still has its (small) DC resistance and at high frequencies some amount of reactance. Impedance is the vector sum of resistance and reactance, essentially the total resistance of a device at a particular frequency, and is:

$$Z = \sqrt{X^2 + R^2}$$

So here's the weird part: shoot a signal with a sharp edge down a wire or PCB track. If the impedance of the gate driving the wire isn't exactly the same as the one receiving the signal, some of the pulse quite literally bounces back to the driver. Since there's still an impedance mismatch, the signal bounces back to the receiver. The bouncing continues until the echoes damp out. Bouncing gets worse as the frequencies increase. As we've seen, shorter rise times generate those high frequencies.

Figure 5.1 is an oscilloscope trace of a not very fast pulse with about a 5 ns rise time. The overshoot and subsequent ringing visible on top of the "one" state is this bouncing effect.

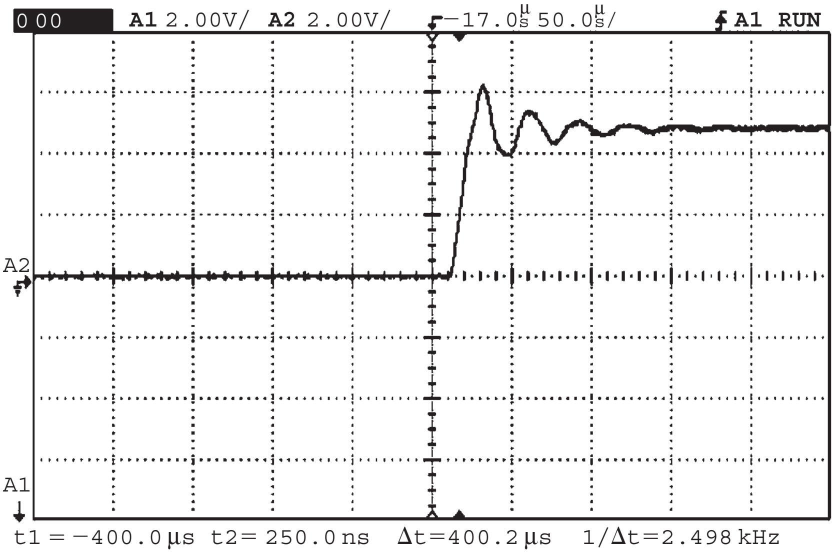


Figure 5.1: Ringing from mismatched impedances

Clearly, if the signal overshoots or undershoots by enough then wrong data can get latched. The system crashes, people die, and you've barricaded yourself behind the entrance as a reporter from *60 Minutes* pounds on the door.

It gets worse. Most of the logic we use today is CMOS, which offers decent performance for minimal power consumption. When the input to a CMOS gate goes somewhat more positive than the power supply (as that overshoot in Figure 5.1 does) the device goes into "SCR latchup." It tries to connect power to ground through its internal transistors. That works for a few milliseconds. Then the part self-destructs from heat.

Back when I was in the in-circuit emulator business we shipped the first of a new line of debuggers. This 8-bit emulator ran at a paltry 6 MHz, but we drove signals to the user's target board very fast, with quite short rise times. The impedance of the customer's board was very different from that of our emulator; the overshoot drove his chips into SCR latchup. Every IC on his board quite literally exploded, plastic debris scattering like politicians dodging Michael Moore.

As I recall we never got paid for that emulator.

### 5.1.3 Firmware Implications

Work with your board developers to insure that the design not only works but is debuggable. You'll hang all sorts of debug and test tools onto nodes on the board to get the code working properly. In-circuit emulators, for instance, connect to many dozens of very fast signals, and always create some sort of impedance mismatch.

Resistors are your friend. It's easy to add a couple of impedance-matching resistors, called *terminators*, at the receiving end of a driven line. Commonly used values are 220 ohms for the pull-up resistor and 270 ohms for the one to ground. Since the power supply has a low impedance (otherwise the voltage to the chips would swing wildly as the load changes) the two resistors are essentially in parallel. With the values indicated the line sees about 120 ohms, not a bad value for most circuits (Figure 5.2).

These will greatly increase the power consumption of the board. An alternative is to replace the bottom resistor with a small, couple-of-hundred picofarad, capacitor.

Make sure the designers put terminations on critical signals. Any edge-triggered input to the processor (or to the circuit board) is particularly susceptible to electromagnetic-induced

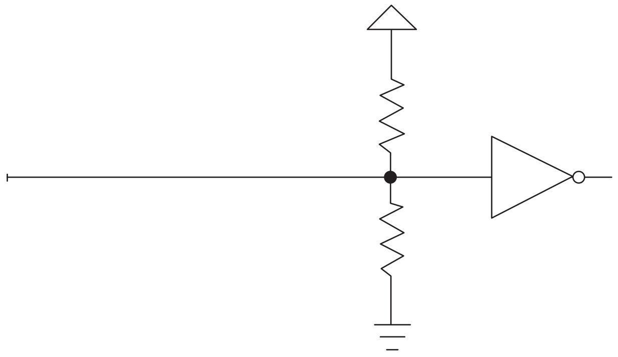


**Figure 5.2: Termination resistors at the end of a driven line match impedances**

noise and reflections. For instance non-maskable interrupt is often edge-triggered. Without termination the slight impedance change resulting from connecting a tool to the line may cause erratic and mysterious crashes.

Many processors reduce pin counts using a multiplexed bus. The CPU supplies the address first, while asserting a strobe named ALE (address latch enable), AS (address strobe), or something similar. External logic latches the value into a register. Then the processor transfers data over the same pins. Just a bit of corruption of ALE will—once in a while—causes the register to latch the wrong information. Without termination, connecting a tool may give you many joyous days of chasing ghostly problems that appear only sporadically.

Clocks are an ever-increasing source of trouble. Most designs use a single clock source that drives perhaps dozens of chips. There's little doubt that the resulting long clock wire will be rife with reflections, destroying its shape. Unfortunately, most CPUs are quite sensitive to the shape and level of clock.

But it's generally not possible to use a termination network on clock, as many CPUs need a clock whose "one" isn't a standard logic level. It's far better to use a single clock source that drives a number of buffers, each of which distributes it to different parts of the board. To avoid skew (where the phase relationship of the signal is a bit different between each of the resulting buffer outputs), use a single buffer chip to produce these signals.

One of my favorite tools for debugging firmware problems is the oscilloscope. A mixed-signal scope, which combines both analog and digital channels, is even better. But scopes lie. Connect the instrument's ground via a long clip lead and the displayed signal will be utterly corrupt since Fourier frequencies far in excess of what you're measuring turn long ground wires into complex transmission lines.

Use that short little ground lead at the end of the probe … and things might not get much better. As speeds increase that 3-inch wire starts to look like an intricate circuit. Cut it shorter. Or remove the ground lead entirely and wrap a circle of wire around the probe's metal ground sheath. The other end gets soldered to the PCB and is as short as is humanly possible.

#### 5.1.3.1 A reference

After staggering through those electromagnetics classes I quickly sold the textbooks. They were pretty nearly written in some language I didn't speak, one as obscure to me as Esperanto.

But there is a book that anyone with a bit of grounding in electronics can master, that requires math no more complex than algebra: *High-Speed Digital Design: A Handbook of Black Magic*, by Howard Johnson and Martin Graham, 1993, PTR Prentice Hall, Englewood Cliffs, NJ.

Reviewers on Amazon love the book or hate it. Me, I find it readable and immediately useful. The work is for the engineer who simply has to get a job done. It's not theoretical at all and presents equations without derivations. But it's practical.

And as for theoretical electromagnetics? My son goes to college to study physics in the fall. I've agreed to write the checks if he answers my physics questions. Maybe in a few years he'll be able to finally explain Maxwell's Laws.

Until then you can pry my termination resistors from my cold, dead hands.

## 5.2 Debouncing

The beer warms a bit as you pound the remote control. Again and again, temper fraying, you click the "channel up" key until the TV finally rewards your efforts. But it turns out channel 345 is playing *Jeopardy* so you again wave the remote in the general direction of the set and continue fiddling with the buttons.

Some remotes work astonishingly well, even when you bounce the beam off three walls before it impinges on the TV's IR detector. Others don't. One vendor told me reliability simply isn't important as users will subconsciously hit the button again and again till the channel changes.

When a single remote press causes the tube to jump two channels, we developers know lousy debounce code is at fault. The FM radio on my sailboat has a tuning button that advances too far when I hit it hard. The usual suspect: bounce.

When the contacts of any mechanical switch bang together, they rebound a bit before settling, causing bounce. Debouncing, of course, is the process of removing the bounces, of converting the brutish realities of the analog world into pristine ones and zeros. Both hardware and software solutions exist, though by far the most common are those done in a snippet of code.

Surf the net to sample various approaches to debouncing. Most are pretty lame. Few are based on experimental bounce parameters. A medley of anecdotal tales passed around the newsgroups substitutes for empirical evidence.

Ask most developers about the characteristics of a bounce and they'll toss out a guess at a max bounce time. But there's an awful lot going on during the bounce. How can we build an effective bounce filter, in hardware or software, unless we understand the entire event? During that time a long and complex string of binary bits is hitting our code. What are the characteristics of that data?

We're writing functions that process an utterly mysterious and unknown input string. That's hardly the right way to build reliable code.

### 5.2.1 The Data

So I ran some experiments.

I pulled some old switches out of my junk box. Twenty bucks at the ever-annoying local Radio Shack yielded more (have you noticed that Radio Shack has fewer and fewer components? It's getting hard to buy a lousy NPN transistor there.). Baynesville Electronics (http://www.baynesvilleelectronics.com), Baltimore's best electronics store, proved a switch treasure trove. Eventually I had 18 very different kinds of switches.

My desktop PC always has a little $49 MSP430 (TI's greatly underrated 16-bit microprocessor) development board attached, with IAR's toolchain installed. It's a matter of seconds to pop a little code into the board and run experiments. Initially I'd planned to connect each switch to an MSP430 input and have firmware read and report bounce parameters. A bit of playing around with the mixed signal scope (MSO) showed this to be an unwise approach.

Many of the switches exhibited quite wild and unexpected behavior. Bounces of under 100 ns were common (more on this later). No reasonable micro could reliably capture these sorts of transitions, so I abandoned that plan and instead used the scope, connecting both analog and digital channels to the switch. This let me see what was going on in the analog domain, and how a computer would interpret the data. A 5 volt supply and 1 k pull-up completed the test jig.

If a sub-100 ns transition won't be captured by a computer, why worry about it? Unfortunately, even a very short signal will toggle the logic once in a while. Tie it to an interrupt and the likelihood increases. Those transitions, though very short, will occasionally pervert the debounce routine. For the sake of the experiment we need to see them.

I gave up regular oscilloscopes long ago; now my Agilent 54645D MSO is a trusty assistant that peers deep into electronic systems. An MSO is both logic analyzer and o-scope, all in one. Trigger from either an analog channel or a digital pattern to start the trace. The MSO shows, like no other instrument, the relationship between the real world and our digital instantiation of it.

I tested the trigger switches from an old cheap game-playing joystick, the left mouse button from an ancient Compaq computer (on PCB in upper left corner), toggle switches, pushbuttons, and slide switches. Some were chassis mount, others were meant to be soldered directly onto circuit boards (Figure 5.3).

I pressed each switch 300 times, logging the min and max amount of bouncing for both closing and opening of the contacts. Talk about mind-numbingly boring! I logged every individual bounce time for each actuation into a spreadsheet for half the switches till my eyes glazed over and gentle wife wondered aloud if I was getting some sort of Pavlovian reward.

The results were interesting.

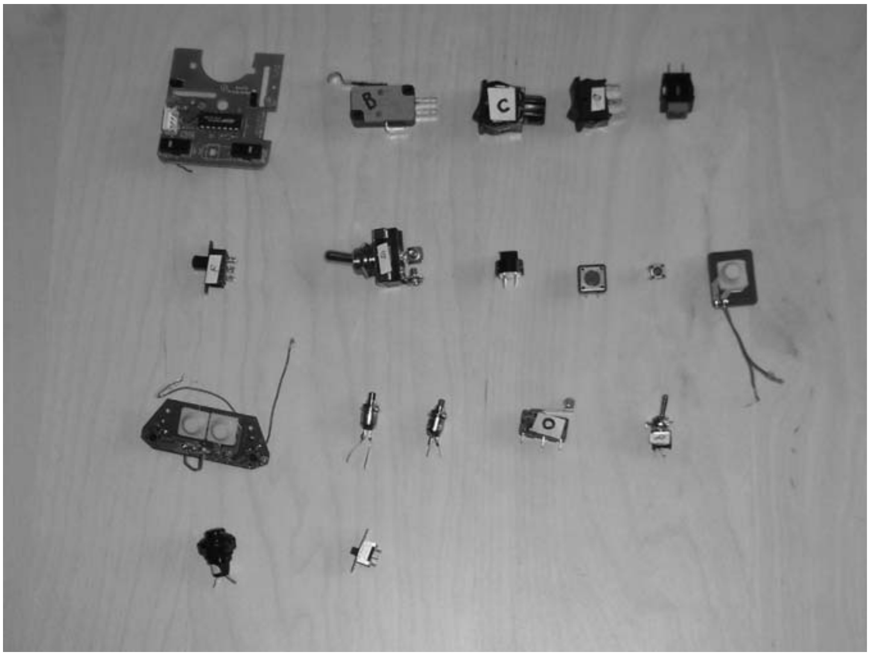


**Figure 5.3: Switches tested. The upper left is switch A, with B to its right, working to E (in red), and then F below A, etc.**

### *5.2.2 Bounce Stats*

So how long do switches bounce for? The short answer: sometimes a lot, sometimes not at all.

Only two switches exhibited bounces exceeding 6200 μsec. Switch E, what seemed like a nice red pushbutton, had a worst-case bounce when it opened of 157 msec—almost 1/6 of a second! Yuk. Yet it never exceeded a 20 μsec bounce when closed. Go figure.

Another switch took 11.3 μsec to completely close one time; other actuations were all under 10 μsec.

Toss out those two samples and the other 16 switches exhibited an average 1557 μsec of bouncing, with, as I said, a max of 6200 μsec. Not bad at all.

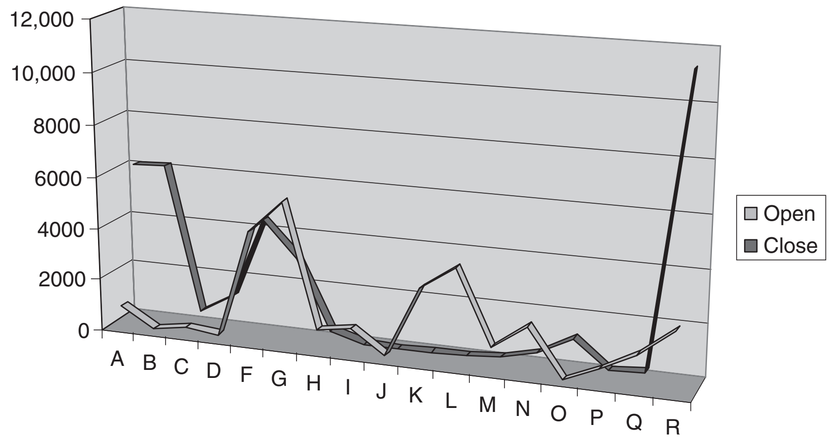


**Figure 5.4: Bounce times in microseconds, for opening and closing each switch (letter A to R). Switch E was left out, as its 157 msec bounces would horribly skew the graph**

Seven of the switches consistently bounced much longer when closed than when opened. I was amazed to find that for most of the switches many bounces on opening lasted for less than 1 μsec—that's right, less than a millionth of a second. Yet the very next experiment on the same switch could yield a reading in the hundreds of microseconds.

Years ago a pal and I installed a system for the Secret Service that had thousands of very expensive switches on panels in a control room. We battled with a unique set of bounce challenges because the uniformed officers were too lazy to stand up and press a button. They tossed rulers at the panels from across the room. Different impacts created (and sometimes destroyed, but hey, it's only taxpayer money after all) quite an array of bouncing. So in these experiments I tried to actuate each device with a variety of techniques: pushing hard or soft, fast or slow, releasing gently or with a snap, looking for different responses. F, a slide switch, was indeed very sensitive to the rate of actuation. Toggle switch G showed a 3 to 1 difference in bounce times depending on how fast I bonked its lever. A few others showed similar results, but there was little discernable pattern (Figure 5.4).

I was fascinated with the switches' analog behavior. A few operated as expected, yielding a solid zero or 5 volts. But most gave much more complicated responses.

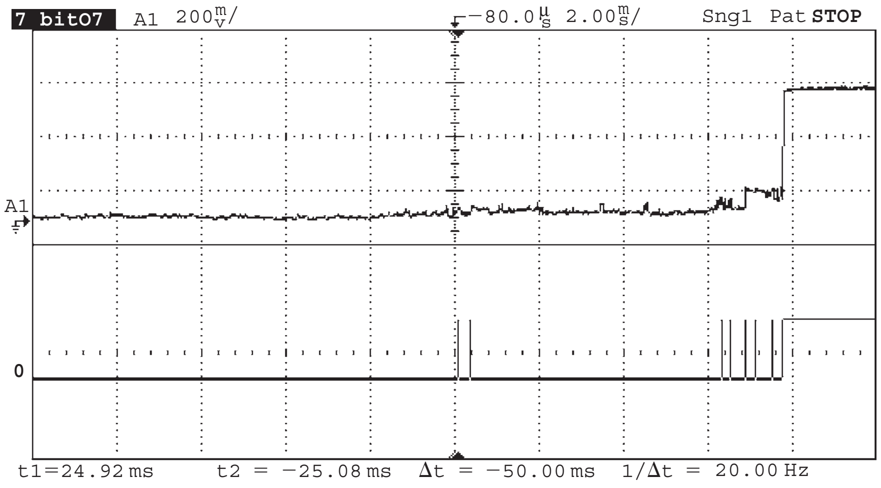


**Figure 5.5: Switch A at 2 msec/div. Note 8 msec of unsettled behavior before it finally decides to open**

The MSO responded to digital inputs assuming TTL signal levels. That means 0–0.8 volts is a zero, 0.8–2.0 volts is unknown, and above 2 is a one. The instrument displayed both digital and analog signals to see how a logic device would interpret the real-world's grittiness.

Switch A was typical. When opened the signal moved just a bit above ground and wandered in the hundreds of millivolts range for up to 8 msec. Then it suddenly snapped to a one. As the signal meandered up to near a volt the scope interpreted it as a one, but the analog's continued uneasy rambles took it in and out of "one" territory. The MSO showered the screen with hash as it tried to interpret the data.

It was as if the contacts didn't bounce so much as wiped, dragging across each other for a time, acting like a variable resistor (Figure 5.5).

Looking into this more deeply, I expanded the traces for switch C and, with the help of Ohm's Law, found the resistance when the device opened crawled pretty uniformly over 150 µsec from 0 to 6 ohms, before suddenly hitting infinity. There was no bouncing per se; just an uneasy ramp up from 0 to 300 mV before it suddenly zapped to a solid +5 (Figure 5.6).

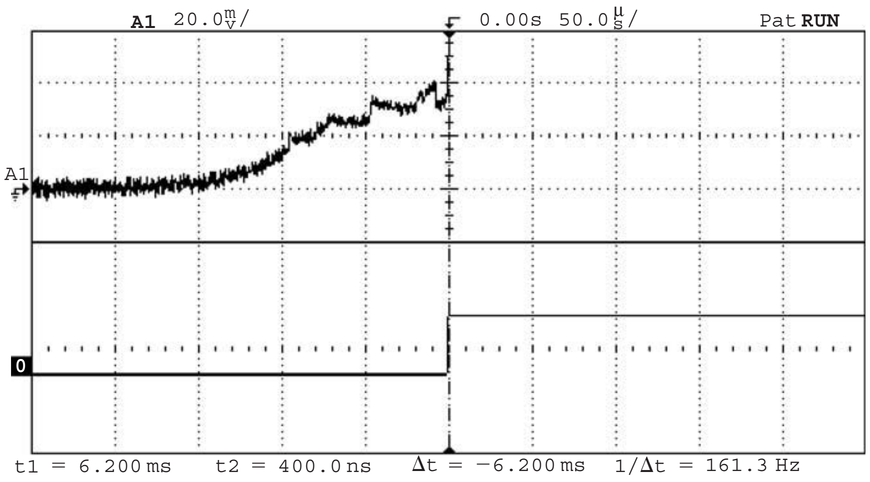


**Figure 5.6: Switch C—50 µsec/div and 200 mV/div**

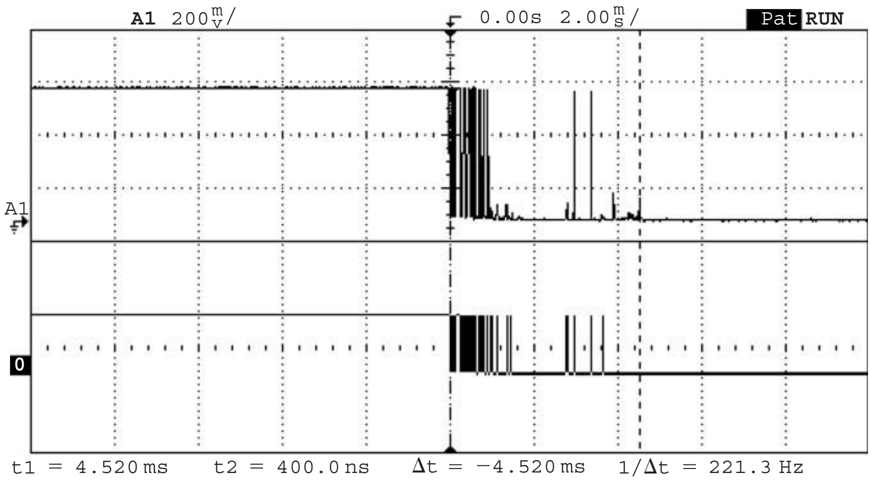


**Figure 5.7: Switch B—note how the analog peak to the right didn't quite trigger the logic channel**

Another artifact of this wiping action was erratic analog signals treading in the dreaded no-man's land of TTL uncertainty (0.8–2.0 volts), causing the MSO to dither, tossing out ones or zeros almost randomly, just as your microprocessor would if connected to the same switch (Figure 5.7).

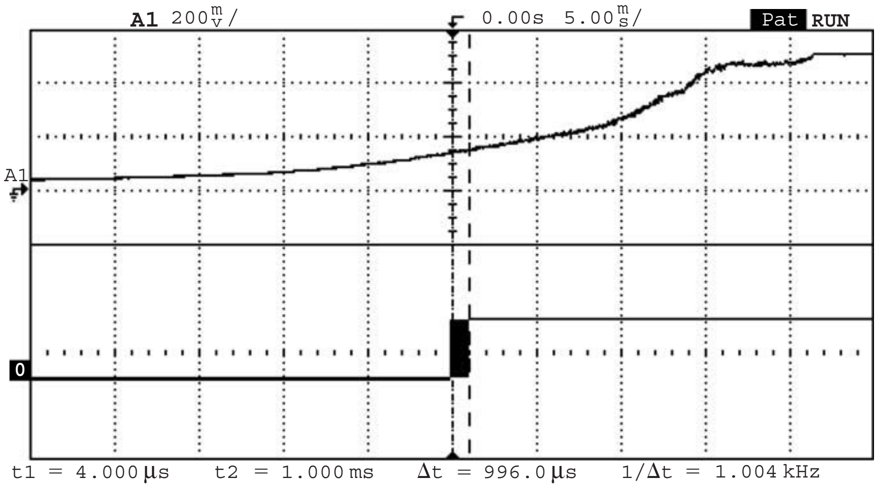


**Figure 5.8: Switch K at 5 msec/div—which slowly ramps up and down when actuated. Cool!**

The two from the el cheapo game joystick were nothing more than gold contacts plated onto a PCB; a rubber cover, when depressed, dropped some sort of conductive elastomer onto the board. Interestingly, the analog result was a slow ramp from 0 to 5 volts, with no noise, wiping, or other uncertainty. Not a trace of bounce. And yet … the logic channel showed a millisecond or so of wild oscillations! What's going on?

With TTL logic, signals in the range of 0.8–2.0 volts are illegal. Anything goes, and everything did. Tie this seemingly bounce-free input to your CPU and prepare to deal with tons of oscillation—virtual bounces (Figure 5.8).

My assessment, then, is that there's much less whacking of contacts going on than we realize. A lot of the apparent logic hash is from analog signals treading in illegal logic regions. Regardless, the effect on our system is the same and the treatment identical. But the erratic nature of the logic warns us to avoid simple sampling algorithms, like assuming two successive reads of a one means a one.

### *5.2.3 Anatomy of a Bounce*

So we know how long the contacts bounce and that lots of digital zaniness—ultra short pulses in particular—can appear.

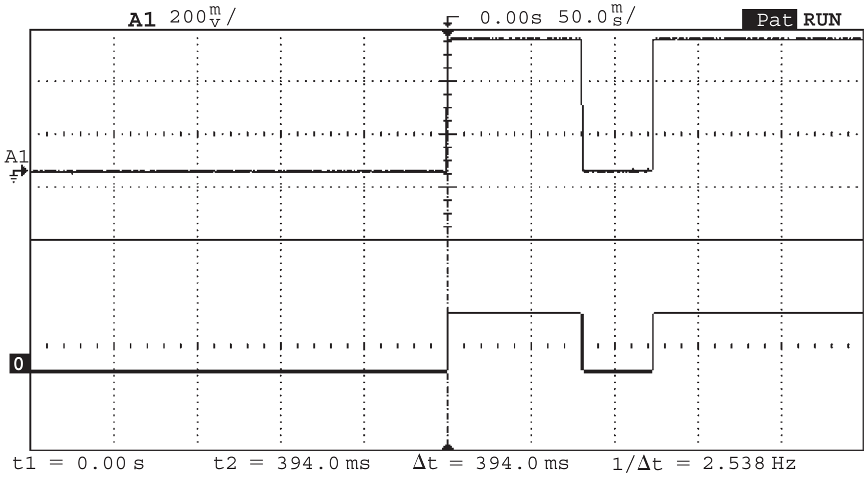


**Figure 5.9: Switch E again, at 50 msec/div. Do you have blood pressure problems? You will after writing code to debounce this!**

But what happens during the bounce? Quite a lot, and every bounce of every switch was different. Many produced only high speed hash till a solid one or zero appeared. Others generated a serious pulse train of discernable logic levels like one might expect. I was especially interested in results that would give typical debounce routines heartburn.

Consider switch E again, that one with the pretty face that hides a vicious 157 msec bouncing heart. One test showed the switch going to a solid one for 81 msec, after which it dropped to a perfect zero for 42 msec before finally assuming its correct high state. Think what that would do to pretty much any debounce code! (Figure 5.9).

Switch G was pretty well behaved, except that a couple of times it gave a few microsecond ones before falling to zero for over 2 msec. Then it assumed its correct final one. The initial narrow pulse might escape your polled I/O, but would surely fire off an interrupt, had you dared wire the system so. The poor ISR would be left puzzled as it contemplates 2 msec of nothingness. “Me? Why did it invoke me? Ain’t nuthin’ there!” (Figure 5.10).

O is a very nice, high quality microswitch which never showed more than 1.18 msec of bouncing. But digging deeper I found it usually generated a pulse train guaranteed to play havoc with simple filter code. There’s no high speed hash, just hard-to-eliminate solid

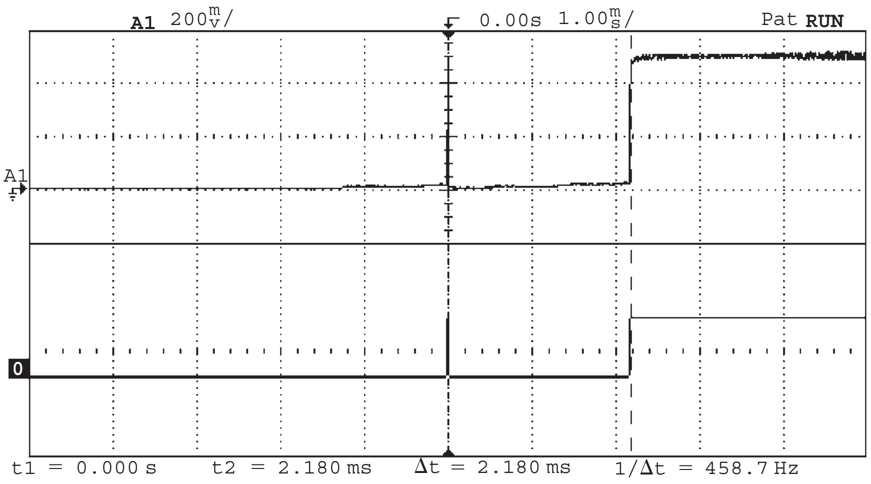


**Figure 5.10: Switch G. One super-narrow pulse followed by 2 msec of nothingness. A sure-fire ISR confuser**

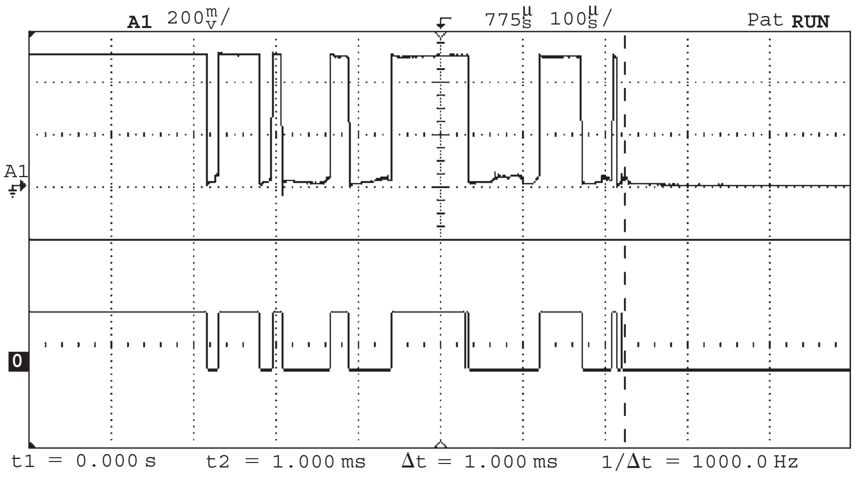


**Figure 5.11: Switch O, which zaps around enough to confuse dumb debouncers**

ones and zeros. One actuation yielded 7 clean zeros levels ranging in time from 12 to 86 µsec, and 7 logic ones varying from 6 to 95 µsec. Easy to filter? Sure. But not by code that just looks for a couple of identical reads (Figures 5.11 and 5.12).

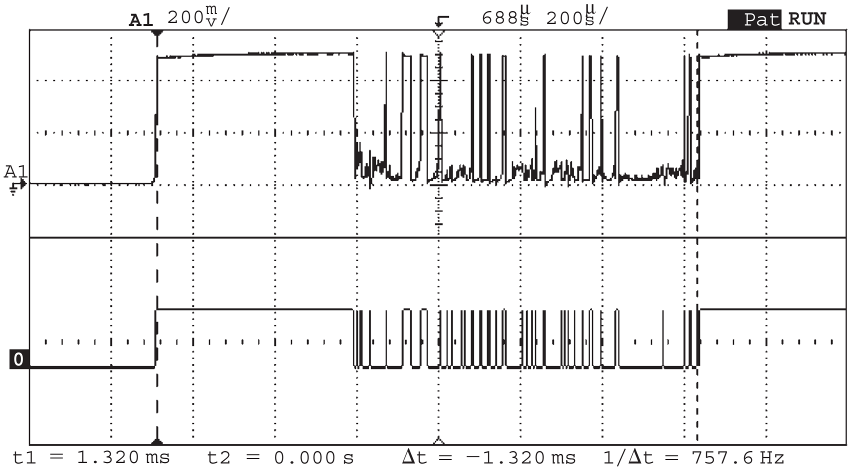


**Figure 5.12: Switch Q—when released, it goes high for 480 µsec before generating 840 µsec of hash, a sure way to blow an interrupt system mad if poorly designed**

What happens if we press the buttons really, really fast? Does that alter the bouncing in a significant way? It's awfully hard for these 50-year-old fingers to do anything particularly quickly, so I set up a modified experiment, connecting my MSP430 board to a sizeable 3 amp four pole relay. Downloading code into the CPU's flash let me toggle the relay at different rates.

Bounce times ranged from 410 to 2920 µsec, quite similar to those of the switches, presumably validating the experiment. The relay had no noticeable analog effects, banging cleanly between 0 and 5 volts.

The raucous clacking of contacts overwhelmed our usual classical fare for a few hours as the MSO accumulated bounce times in storage mode. When the relay opened, it always had a max bounce time of 2.3 to 2.9 msec, at speeds from 2.5 to 30 Hz. More variation appeared on contact closure: at 2.5 Hz bounces never exceeded 410 µsec, which climbed to 1080 µsec at 30 Hz. Why? I have no idea. But it's clear there is some correlation between fast actuations and more bounce. These numbers suggest a tractable factor of two increase, though—not a scary order of magnitude or more.

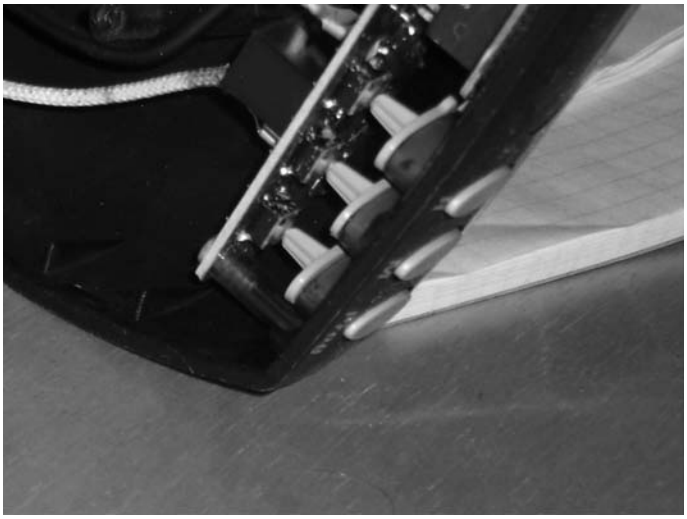


**Figure 5.13: PCB switches in a cheap coffeemaker**

In the bad old days we used a lot of leaf switches which typically bounced forever. Weeks, it seemed. Curious I disassembled a number of cheap consumer products expecting to find these sorts of inexpensive devices. None found! Now that everything is mounted on a PCB vendors use board-mounted switches, which are pretty darn good little devices (Figure 5.13).

I admit these experiments aren't terribly scientific. No doubt someone with a better education and more initials following his name could do a more reputable study for one of those journals no one reads. But as far as I know there's no data on the subject available anywhere, and we working engineers need some empirical information.

Use a grain of salt when playing with these numbers. Civil engineers don't really know the exact strength of a concrete beam poured by indolent laborers, so they beef things up a bit. They add margin. Do the same here. Assume things are worse than shown.

### *5.2.4 Hardware Debouncers*

The following schematic shows a classic debounce circuit. Two cross-coupled NAND gates form a very simple Set-Reset (SR) latch. The design requires a double-throw

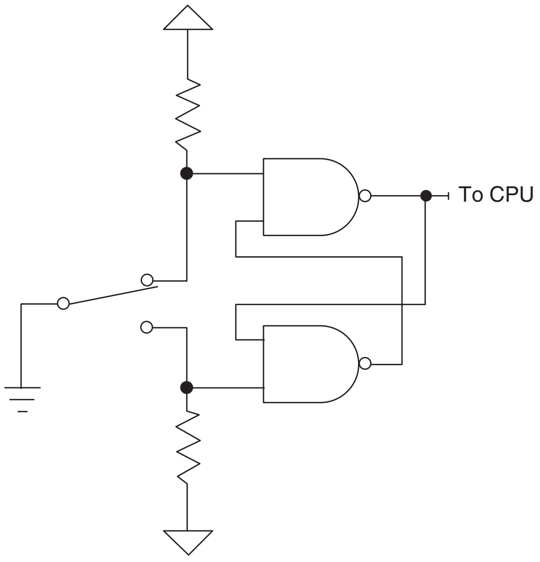


**Figure 5.14: The SR debouncer**

switch. Two pull-up resistors generate a logic one for the gates; the switch pulls one of the inputs to ground (Figure 5.14).

The SR latch is a rather funky beast, as confusing to non-EEs as recursion is to, well, just about everyone.

With the switch in the position shown the upper gate's output will be a one, regardless of the value of the other input. That and the one created by the bottom pull-up resistor drives the lower NAND to a zero … which races around back into the other gate. If the switch moves between contacts and is for a while suspended in the nether region between terminals, the latch maintains its state because of the looped back zero from the bottom gate.

The switch moves a rather long way between contacts. It may bounce around a bit but will never bang all the way back to the other contact. Thus, the latch's output is guaranteed bounce-free.

The circuit suggests an alternative approach, a software version of the same idea. Why not skip the NAND pair and run the two contracts, with pull-ups, directly to input pins on the CPU? Sure, the computer will see plenty of bounciness, but write a trivial bit of code that detects *any* assertion of either contact … which means the switch is in that position, as follows:

```
if(switch_hi())state=ON;
if(switch_lo())state=OFF;
```

`switch_hi` and `switch_lo` each reads one of the two throws. Other functions in the program examine variable `state` to determine the switch's position.

This saves two gates but costs one extra input pin on the processor. It's the simplest—and most reliable—debounce code possible.

The MC14043/14044 chips consist of four SR flip-flops, so might be an attractive solution for debouncing multiple switches. A datasheet can be found at http://www.radanpro.com/el/dslpro.php?MC14043.pdf.

### 5.2.5 An RC Debouncer

The SR circuit is the most effective of all debouncing approaches … but it's rarely used. Double-throw switches are bulkier and more expensive than the simpler single-throw versions. An awful lot of us use switches that are plated onto the circuit board, and it's impossible to make DP versions of these. So EEs prefer alternative designs that work with cheap single-throw switches.

Though complex circuits using counters and smart logic satisfy our longing for pure digital solutions to all problems, from signal processing to divorce, it's easier and cheaper to exploit the peculiar nature of a resistor–capacitor (RC) network.

Charge or discharge a capacitor through a resistor and you'll find the voltage across the cap rises slowly; it doesn't snap to a new value like a sweet little logic circuit. Increase the value of either component and the time lag ("time constant" in EE lingo) increases (Figure 5.15).

This circuit is a typical RC debouncer. A simple circuit, surely, yet one that hides a surprising amount of complexity.

Suppose our fearless flipper opens the switch. The voltage across the cap is zero, but it starts to climb at a rate determined by the values of $R_1$, $R_2$, and C. Bouncing contacts pull the voltage down and slow the cap's charge accumulation. If we're very clever in selecting the values of the components, the voltage stays below a gate's logic one level

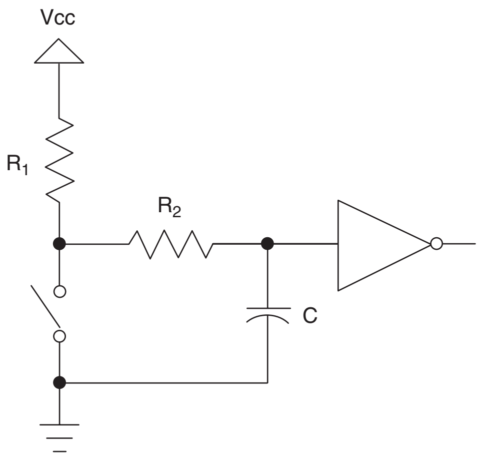


**Figure 5.15: An RC debouncer**

until all of the whacking and thudding ceases. (If the time constant is too long, of course, the system won't be responsive to fast switch actuations.)

The gate's output is thus a pristine bounce-free logic level.

Now suppose the switch has been open for a while. The cap is fully charged. Snap! The user closes the switch, which discharges the cap through $R_2$. Slowly, again, the voltage drools down and the gate continues to see a logic one at its input for a time. Perhaps the contacts open and close a bit during the bouncing. While open, even if only for short periods, the two resistors start to recharge the cap, reinforcing the logic one to the gate. Again, the clever designer selects component values that guarantee the gate sees a one until the clacking contacts settle.

Squalid taverns are filled with grizzled veterans of the bounce wars recounting their circuits and tales of battles in the analog trenches. Most will puzzle over $R_2$, and that's not entirely due to the effects of the cheap booze. The classic RC debouncer doesn't use this resistor, yet it's critically important to getting a thwack-free output from the gate.

$R_2$ serves no useful purpose when the switch opens. $R_1$ and C effectively remove those bounces. But strange things can happen when suddenly discharging a capacitor. The early bouncing might be short, lasting microseconds or less. Though a dead short should instantly discharge the cap, there are no pristine conditions in the analog world. The switch has some resistance, as do the wires and PCB tracks that interconnect everything.

Every wire is actually a complex circuit at high speeds. You wouldn't think a dull-headed customer flipping the switch a few times a second would be generating high-speed signals, but sub-microsecond bounces, which may have very sharp rise times, have frequency components in the tens of MHz or more. Inductance and stray capacitance raise the impedance (AC resistance) of the closed switch. The cap won't instantly discharge.

Worse, depending on the physical arrangement of the components, the input to the gate might go to a logic zero while the voltage across the cap is still one-ish. When the contacts bounce open, the gate now sees a one. The output is a train of ones and zeros—bounces.

$R_2$ insures the cap discharges slowly, giving a clean logic level regardless of the storm of bounces. The resistor also limits current flowing through the switch's contacts, so they aren't burned up by a momentary major surge of electrons from the capacitor.

Another trick lurks in the design. The inverter cannot be a standard logic gate. TTL, for instance, defines a zero as an input between 0.0 and 0.8 volts. A one starts at 2.0. In between is a DMZ which we're required to avoid. Feed 1.2 volts to such a gate, and the output is unpredictable. But this is exactly what will happen as the cap charges and discharges.

Instead use a device with "Schmitt Trigger" inputs. These devices have hysteresis; the inputs can dither yet the output remains in a stable, known state.

Never run the cap directly to the input on a microprocessor, or to pretty much any I/O device. Few of these have any input hysteresis.

### 5.2.6 Doing the Math

The equation for discharging a cap is:

$$V_{\text{cap}} = V_{\text{initial}}(e^{\frac{-t}{RC}})$$

where
$V_{\text{cap}}$ is the voltage across the capacitor at time $t$,
$V_{\text{initial}}$ is the voltage initially on the cap,

$t$ is the time in seconds,
$R$ and $C$ are the values of the resistor and capacitor in ohms and farads, respectively.

The trick is to select values that insure the cap's voltage stays above $V_{th}$, the threshold at which the gate switches, till the switch stops bouncing. It's surprising how many of those derelicts hanging out at the waterfront bars pick an almost random time constant. "The boys 'n me, we jest figger sumpin like 5 msec." Shortchanging a real analysis starts even a clean-cut engineer down the slippery slope to the wastrel vagabond's life.

Most of the switches I examined had bounce times well under 10 msec. Use 20 to be conservative.

Rearranging the time constant formula to solve for $R$ (the cost and size of caps vary widely so it's best to select a value for $C$ and then compute $R$) yields:

$$R = \frac{-t}{C \ln\left(\frac{V_{th}}{V_{initial}}\right)}$$

Though it's an ancient part, the 7414 hex inverter is a Schmitt Trigger with great input hysteresis. The AHCT version has a worst case $V_{th}$ for a signal going low of 1.7 volts. (Note that these parameters vary widely depending on the part's manufacturer.) Let's try 0.1 µF for the capacitor since those are small and cheap, and solve for the condition where the switch just closes. The cap discharges through $R_2$. If the power supply is 5 volts (so $V_{initial}$ is 5), then $R_2$ is 185 K. Of course, you can't actually *buy* that kind of resistor, so use 180 K.

But, the analysis ignores the gate's input leakage current. A CMOS device like the 74AHCT14 dribbles about a microamp from the inputs. That 180 K resistor will bias the input up to 0.18 volts, uncomfortably close to the gate's best-case switching point of 0.5 volt. Change $C$ to 1 µF and $R_2$ is now 18 K.

$R_1 + R_2$ controls the cap's charge time, and so sets the debounce period for the condition where the switch opens. The equation for charging is:

$$V_{th} = V_{final}(1 - e^{-t/RC})$$

Solving for $R$:

$$R = \frac{-t}{C \ln\left(1 - \frac{V_{\text{th}}}{V_{\text{final}}}\right)}$$

$V_{\text{final}}$ is the final charged value—the 5 volt power supply. $V_{\text{th}}$ is now the worst-case transition point for a high-going signal, which for our 74AHCT14 is a peachy 0.9 volts. $R_1 + R_2$ works out to 101 K. Figure on 82 K (a standard part) for $R_1$ (Figure 5.16).

The diode is an optional part needed only when the math goes haywire. It's possible, with the wrong sort of gate where the hysteresis voltages assume other values, for the formulas to pop out a value for $R_1 + R_2$ which is less than that of $R_2$. In this case the diode forms a short cut that removes $R_2$ from the charging circuit. All of the charge flows through $R_1$. The previous equation still applies, except we have to account for drop across the diode. Change $V_{\text{final}}$ to 4.3 volts (5 minus the 0.7 diode drop), turn the crank, and $R_1$ pops out.

Be wary of the components' tolerances. Standard resistors are usually ±5%. Capacitors vary wildly—+80/−20% is a common rating for electrolytics. Even small ceramics might vary ±30%.

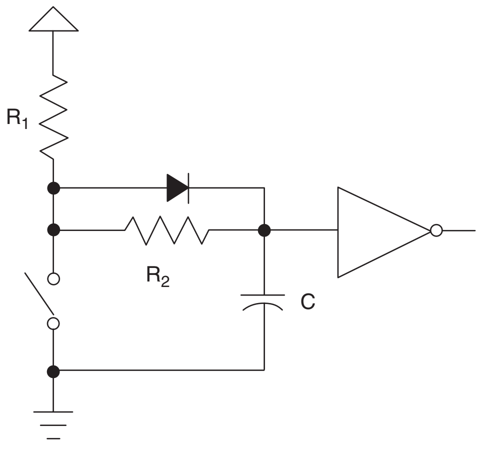


**Figure 5.16: An RC debouncer that actually works in all cases**

### 5.2.7 Other Thoughts

Don't neglect to account for the closed resistance of oddball switches. Some conductive elastomer devices exceed 200 ohms.

Two of the elastomer switches I examined didn't bounce at all; their output smoothly ramped from 0 to +5 volts. The SR and RC debounce circuits are neither necessary nor effective. Better: run the switch directly into a Schmitt Trigger's input.

Never connect an undebounced switch to the clock of a flip-flop. The random bounce hash is sure to confuse the device. A 74HCT74 has a max rise and fall time spec of 6 nsec—easily exceeded by some of the data I acquired from the 18 switches tested.

The 74HC109 requires a minimum clock width of 100 nsec. I found pulses shorter than this in my experiments. Its higher-tech brother, the 74HFC109, actually has a Schmitt Trigger clock input—it's a much safer part to use when connected to real-world events.

Similarly, don't tie undebounced switches, even if Schmitt Triggered, to interrupt inputs on the CPU. Usually the interrupt pin goes to the clock input of an internal flip-flop. As processors become more complex, their datasheets give less useful electrical information; they're awash in programming data but leave designers adrift without complete timing specs. Generally we have no idea what the CPU expects as a max rise time or the min pulse width. Those internal flops aren't perfect, so don't flirt with danger by feeding them garbage.

The MC14490 is a cool chip that consists of 6 debouncers. A datasheet is at http://engineering.dartmouth.edu/~engs031/databook/mc14490.pdf. But in August of 2004 Digikey wants $5.12 each for these parts; it's cheaper to implement a software debounce algorithm in a PIC or similar sub-$1 microcontroller.

Always remember to tie unused inputs of any logic circuit to Vcc or ground.

### 5.2.8 Software Debouncers

Software debounce routines range from the utterly simple to sophisticated algorithms that handle multiple switches in parallel. But many developers create solutions without completely understanding the problem. Sure, contacts rebound against each other. But the environment itself can induce all sorts of short transients that mask themselves as switch transitions. Called EMI (electromagnetic interference), these bits of nastiness come from energy coupled into our circuits from wires running to the external world, or even from static electricity zaps induced by shuffling feet across a dry carpet. Happily EMI and contact whacking can be cured by a decent debounce routine … but both factors do affect the design of the code.

Consider the simplest of all debouncing strategies: read the switch once every 500 msec or so, and set a flag indicating the input's state. No reasonable switch will bounce that long. A read during the initial bounce period returns a zero or a one indicating the switch's indeterminate state. No matter how we interpret the data (i.e., switch on or off) the result is meaningful. The slow read rate keeps the routine from deducing that bounces are multiple switch closures. One downside, though, is slow response. If your user won't hit buttons at a high rate, this is probably fine. A fast typist, though, can generate 100 words per minute or almost 10 characters per second. A rotating mechanical encoder could generate even faster transitions.

But there's no EMI protection inherent in such a simple approach. An application handling contacts plated onto the PCB is probably safe from rogue noise spikes, but one that reads from signals cabled onto the board needs more sophisticated software, since a single glitch might look like a contact transition.

It's tempting to read the input a couple of times each pass through the 500 msec loop and look for a stable signal. That'll reject much or maybe all of the EMI. But some environments are notoriously noisy. Many years ago I put a system using several Z80 s and a PDP-11 in a steel mill. A motor the size of a house drawing thousands of amps drove the production line. It reversed direction every few seconds. The noise generated by that changeover coupled *everywhere*, and destroyed everything electronic unless carefully protected. We optocoupled all cabling simply to keep the smoke inside the ICs, where it belongs. All digital inputs still looked like hash and needed an astonishing amount of debounce and signal conditioning.

### 5.2.9 Debounce Policy

Seems to me there are some basic constraints to place on our anti-contact-clacking routines. Minimize CPU overhead. Burning execution time while resolving a bounce is a dumb way to use processor cycles. Debounce is a small problem and deserves a small part of the computer's attention.

The undebounced switch must connect to a programmed I/O pin, never to an interrupt. Few microprocessor datasheets give much configuration or timing information about the interrupt inputs. Consider Microchip's PIC12F629 (datasheet at http://ww1.microchip.com/downloads/en/DeviceDoc/41190c.pdf). A beautiful schematic shows an interrupt pin run through a Schmitt Trigger device to the data input of a pair of flops. Look closer and it's clear that's used only for one special "interrupt on change" mode. When the pin is used as a conventional interrupt the signal disappears into the bowels of the CPU, sans hysteresis and documentation. However, you can count on the interrupt driving the clock or data pin on an internal flip flop. The bouncing zaniness is sure to confuse any flop, violating minimum clock width or the data setup and hold times.

Try to avoid sampling the switch input at a rate synchronous to events in the outside world that might create periodic EMI. For instance, 50 and 60 Hz are bad frequencies. Mechanical vibration can create periodic interference. I'm told some automotive vendors have to avoid sampling at a rate synchronous to the vibration of the steering column.

Finally, in most cases it's important to identify the switch's closure quickly. Users get frustrated when they take an action and there's no immediate response. You press the button on the gas pump or the ATM and the machine continues to stare at you, dumbly, with the previous screen still showing, till the brain-dead code finally gets around to grumpily acknowledging that, yes, there IS a user out there and the person actually DID press a button.

Respond *instantly* to user input. In this fast-paced world delays aggravate and annoy. But how fast is fast enough?

I didn't know so I wired a switch up to an SBC and wrote a bit of simple code to read a button and, after a programmable delay, turn on an LED. Turns out a 100 msec delay is quite noticeable, even to these tired old 20/1000 eyes. 50 msec, though, seemed instantaneous. Even the kids concurred, astonishing since it's so hard to get them to agree on anything.

So let's look at a couple of debouncing strategies.

## 5.2.10 A Counting Algorithm

Most people use a fairly simple approach that looks for *n* sequential stable readings of the switch, where *n* is a number ranging from 1 (no debouncing at all) to, seemingly, infinity.

Generally the code detects a transition and then starts incrementing or decrementing a counter, each time rereading the input, till *n* reaches some presumably safe, bounce-free, count. If the state isn't stable, the counter resets to its initial value.

Simple, right? Maybe not. Too many implementations need some serious brain surgery. For instance, use a delay so the repetitive reads aren't back to back, merely microseconds apart. Unless your application is so minimal there are simply no free resources, don't code the delay using the classic construct: `for(i=0;i<big_number;++i);`. Does this idle for a millisecond… or a second? Port the code to a new compiler or CPU, change wait states or the clock rate and suddenly the routine breaks, requiring manual tweaking. Instead use a timer that interrupts the CPU at a regular rate—maybe every millisecond or so—to sequence these activities.

The following code shows a sweet little debouncer that is called every `CHECK_MSEC` by the timer interrupt, a timer-initiated task, or some similar entity.

## A simple yet effective debounce algorithm

```
#define CHECK_MSEC   5   // Read hardware every 5 msec
#define PRESS_MSEC   10  // Stable time before registering pressed
#define RELEASE_MSEC100// Stable time before registering released

// This function reads the key state from the hardware.
extern bool_t RawKeyPressed();

// This holds the debounced state of the key.
bool_t DebouncedKeyPress = false;
//   Service routine called every CHECK_MSEC to
// debounce both edges
void DebounceSwitch1(bool_t *Key_changed, bool_t *Key_pressed)
{
      static uint8_t Count = RELEASE_MSEC / CHECK_MSEC;
      bool_t RawState;
      *Key_changed = false;
      *Key_pressed = DebouncedKeyPress;
      RawState = RawKeyPressed();
      if (RawState == DebouncedKeyPress) {
```

```
      // Set the timer which allows a change from current state.
      if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
      else                   Count = PRESS_MSEC / CHECK_MSEC;
      } else{
        // Key has changed - wait for new state to become stable.
        if (--Count == 0) {
              // Timer expired - accept the change.
              DebouncedKeyPress = RawState;
              *Key_changed=true;
              *Key_pressed=DebouncedKeyPress;
              // And reset the timer.
              if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
      else                           Count = PRESS_MSEC / CHECK_MSEC;
      }
   }
}
```

You'll notice there are no arbitrary count values; the code doesn't wait for *n* stable states before declaring the debounce over. Instead it's all based on time and is therefore eminently portable and maintainable.

`DebounceSwitch1()` returns two parameters. `Key_Pressed` is the current debounced state of the switch. `Key_Changed` signals the switch has changed from open to closed, or the reverse.

Two different intervals allow you to specify different debounce periods for the switch's closure and its release. To minimize user delays why not set `PRESS_MSEC` to a relatively small value, and `RELEASE_MSEC` to something higher? You'll get great responsiveness yet some level of EMI protection.

### 5.2.11 An Alternative

An even simpler routine, shown below, returns TRUE once when the debounced leading edge of the switch closure is encountered. It offers protection from both bounce and EMI.

## An even simpler debounce routine

```
// Service routine called by a timer interrupt
bool_t DebounceSwitch2()
{
static uint16_t State = 0; // Current debounce status
State=(State<<1) | !RawKeyPressed() | pipe; 0xe000;
if(State==0xf000)return TRUE;
return FALSE;
}
```

Like the routine in listing 1, `DebounceSwitch2()` gets called regularly by a timer tick or similar scheduling mechanism. It shifts the current raw value of the switch into variable `State`. Assuming the contacts return zero for a closed condition, the routine returns `FALSE` till a dozen sequential closures are detected.

One bit of cleverness lurks in the algorithm. As long as the switch isn't closed, ones shift through `State`. When the user pushes on the button the stream changes to a bouncy pattern of ones and zeros, but at some point there's the last bounce (a one) followed by a stream of zeros. We OR in `0xe000` to create a "don't care" condition in the upper bits. But as the button remains depressed, `State` continues to propagate zeros. There's just the one time, when the last bouncy "one" was in the upper bit position, that the code returns a `TRUE`. That bit of wizardry eliminates bounces and detects the edge, the transition from open to closed.

Change the two hex constants to accommodate different bounce times and timer rates.

Though quite similar to a counting algorithm this variant translates much more cleanly into assembly code. One reader implemented this algorithm in a mere 11 lines of 8051 assembly language.

Want to implement a debouncer in your FPGA or ASIC? This algorithm is ideal. It's loopless and boasts but a single decision, one that's easy to build into a single wide gate.

### *5.2.12 Handling Multiple Inputs*

Sometimes we're presented with a bank of switches on a single input port. Why debounce these individually when there's a well-known (though little used) algorithm to handle the entire port in parallel?

Here's one approach. `DebounceSwitch()`, which is called regularly by a timer tick, reads an entire byte-wide port that contains up to 8 individual switches. On each call it stuffs the port's data into an entry in circular queue `State`. Though shown as an array with but a single dimension, a second loiters hidden in the width of the byte. `State` consists of columns (array entries) and rows (each defined by bit position in an individual entry, and corresponding to a particular switch).

**Code that debounces many switches at the same time**

```
#define MAX_CHECKS 10           // # checks before a switch is debounced
uint8_t Debounced_State;        // Debounced state of the switches
uint8_t State[MAX_CHECKS];      // Array that maintains bounce status
uint8_t Index;                  // Pointer into State

// Service routine called by a timer interrupt
void DebounceSwitch3()
{
       uint8_t i,j;
       State[Index]=RawKeyPressed();
       ++Index;
       j=0xff;
       for(i=0; i<MAX_CHECKS;i++)j=j & State[i];
       Debounced_State= j;
       if(Index>=MAX_CHECKS)Index=0;
}
```

A short loop ANDs all column entries of the array. The resulting byte has a one in each bit position where that particular switch was on for every entry in `State`. After the loop completes, variable `j` contains 8 debounced switch values.

One could exclusive OR this with the last `Debounced_State` to get a one in each bit where the corresponding switch has changed from a zero to a one, in a nice debounced fashion.

Don't forget to initialize `State` and `Index` to zero.

I prefer a less computationally intensive alternative that splits `DebounceSwitch()` into two routines; one, driven by the timer tick, merely accumulates data into array `State`. Another function, `Whats_Da_Switches_Now(), ANDs` and `XORs` as described, but only when the system needs to know the switches' status.

## 5.2.13 Summing up

All of these algorithms assume a timer or other periodic call that invokes the debouncer. For quick response and relatively low computational overhead I prefer a tick rate of a handful of milliseconds. One to five milliseconds is ideal. Most switches seem to exhibit under 10 msec bounce rates. Coupled with my observation that a 50 msec response seems instantaneous, it seems reasonable to pick a debounce period in the 20–50 msec range.

Hundreds of other debouncing algorithms exist. These are just a few of my favorite, offering great response, simple implementation, and no reliance on magic numbers or other sorts of high-tech incantations.

CHAPTER 6

# Disciplined Development

## 6.1 Disciplined Development

The seduction of the keyboard has been the downfall of all too many embedded projects.

Writing code is fun. It's satisfying. We feel we're making progress on the project. Our bosses, all too often unskilled in the nuances of building firmware, look on approvingly, smiling because we're clearly accomplishing something worthwhile.

As a young developer working on assembly language-based systems I learned to expect long debugging sessions. Crank some code, and figure on months making it work. Debugging is hard work so I learned to budget 50% of the project time to chasing down problems.

Years later, while making and selling emulators, I saw this pattern repeated, constantly, in virtually every company I worked with. In fact, this very approach to building firmware is a godsend to the tool companies who all thrive on developers' poor practices and resulting sea of bugs. Without bugs debugger vendors would be peddling pencils.

A quarter century after my own first dysfunctional development projects, in my travels lecturing to embedded designers, I find the pattern remains unbroken. The rush to write code overwhelms all common sense.

The overused word "process" (note only the word is overused; the concept itself is sadly neglected in the firmware world) has garnered enough attention that some developers claim to have institutionalized a reasonable way to create software. Under close questioning, though, the majority of these admits to applying their rules in a haphazard manner. When the pressure heats up—the very time when sticking to a system that *works* is most needed—most succumb to the temptation to drop the systems and just crank out code.

As you're boarding a plane you overhear the pilot tell his right-seater "We're a bit late today; let's skip the take-off checklist." Absurd? Sure. Yet this is precisely the tack we take as soon as deadlines loom; we abandon all discipline in a misguided attempt to beat our code into submission.

### 6.1.1 Any Idiot Can Write Code

In their studies of programmer productivity, Tom DeMarco and Tim Lister found that all things being equal, programmers with a mere 6 months of experience typically perform as well as those with a year, a decade, or more.

As we developers age we get more experience—but usually the same experience, repeated time after time. As our careers progress we justify our escalating salaries by our perceived increasing wisdom and effectiveness. Yet the data suggests that the *value of experience is a myth*.

Unless we're prepared to find new and better ways to create firmware, and until we implement these improved methods, we're no more than a step above the wild-eyed teen-age guru who lives on Jolt and Twinkies while churning out astonishing amounts of code.

*Any idiot can create code; professionals find ways to consistently create high quality software on time and on budget.*

### 6.1.2 Firmware Is the Most Expensive Thing in the Universe

Norman Augustine, former CEO of Lockheed Martin, tells a revealing story about a problem encountered by the defense community. A high performance fighter aircraft is a delicate balance of conflicting needs: fuel range versus performance, speed versus weight. It seems that by the late 1970s fighters were about as heavy as they'd ever be. Contractors, always pursuing larger profits, looked in vain for something they could add that cost a lot, but which weighed nothing.

The answer: firmware. Infinite cost, zero mass. Avionics now accounts for more than 40% of a fighter's cost.

Two decades later nothing has changed … except that firmware is even more expensive.

### *6.1.3 What Does Firmware Cost?*

Bell Labs found that to achieve 1–2 defects per 1000 lines of code they produce 150–300 lines per month. Depending on salaries and overhead, this equates to a cost of around $25–50 per line of code.

Despite a lot of unfair bad press, IBM's space shuttle control software is remarkably error free, and may represent the best firmware ever written. The cost? $1000 per statement, for no more than one defect per 10,000 lines.

Little research exists on embedded systems. After asking for a per-line cost of firmware I'm usually met with a blank stare followed by an absurdly low number. "$2 a line, I guess" is common. Yet, a few more questions (How many people? How long from inception to shipping?) reveal numbers an order of magnitude higher.

Anecdotal evidence, crudely adjusted for reality, suggests that if you figure your code costs $5 a line you're lying—or the code is junk. At $100/line you're writing software documented almost to DOD standards. Most embedded projects wind up somewhere in-between, in the $20–40/line range. There are a few gurus out there who consistently do produce quality code much cheaper than this, but they're on the 1% asymptote of the bell curve. If you feel you're in that select group—we all do—take data for a year or two. Measure time spent on a project from inception to completion (with all bugs fixed) and divide by the program's size. Apply your loaded salary numbers (usually around twice the number on your paycheck stub). You'll be surprised.

### *6.1.4 Quality Is Nice ... As Long As It's Free*

The cost data just described is correlated to a quality level. Since few embedded folks measure bug rates, it's all but impossible to add the quality measure into the anecdotal costs. But quality does indeed have a cost.

We can't talk about quality without defining it. Our intuitive feel that a bug-free program is a high quality program is simply wrong. Unless you're using the Netscape "give it away for free and make it up in volume" model, we write firmware for one reason only: profits. Without profits the engineering budget gets trimmed. Without profits the business eventually fails and we're out looking for work.

Happy customers make for successful products and businesses. The customer's delight with our product is the ultimate and only important measure of quality.

Thus: *the quality of a product is exactly what the customer says it is*.

Obvious software bugs surely mean poor quality. A lousy user interface equates to poor quality. If the product doesn't quite serve the buyer's needs, the product is defective.

It matters little whether our code is flaky or marketing overpromised or the product's spec missed the mark. The company is at risk due to a quality problem so we've all got to take action to cure the problem.

No-fault divorce and no-fault insurance acknowledge the harsh realities of trans-millennium life. We need a no-fault approach to quality as well, to recognize that no matter where the problem came from, we've all got to take action to cure the defects and delight the customer.

This means when marketing comes in a week before delivery with new requirements, a mature response from engineering is not a stream of obscenities. Maybe … just maybe … marketing has a point. We make mistakes (and spend heavily on debugging tools to fix them). So do marketing and sales.

Substitute an assessment of the proposed change for curses. Quality is not free. If the product will not satisfy the customer as designed, if it's not till a week before shipment that these truths become evident, then let marketing and other departments know the impact on the cost and the schedule.

Funny as the Dilbert comic strip is, it does a horrible disservice to the engineering community by reinforcing the hostility between engineers and the rest of the company. The last thing we need is more confrontation, cynicism, and lack of cooperation between departments. We're on a mission: *make the customer happy*! That's the *only* way to consistently drive up our stock options, bonuses, and job security.

Unhappily, Dilbert does portray too many companies all too accurately. If your outfit requires heroics all the time, if there's no (polite) communication between departments, then something is broken. Fix it or leave.

### 6.1.5 The CMMI

Few would deny that firmware is a disaster area, with poor quality products getting to market late and over budget. Don't become resigned to the status quo. As engineers we're paid to solve problems. No problem is greater, no problem is more important, than finding or inventing faster, better ways to create code.

The Software Engineering Institute's (www.sei.cmu.edu) Capability Maturity Model Integration (CMMI) defines five levels of software maturity and outlines a plan to move up the scale to higher, more effective levels:

1. *Initial*: Ad hoc and Chaotic. Few processes are defined, and success depends more on individual heroic efforts than on following a process and using a synergistic team effort.

2. *Repeatable*: Intuitive. Basic project management processes are established to track cost, schedule, and functionality. Planning and managing new products are based on experience with similar projects.

3. *Defined*: Standard and Consistent. Processes for management and engineering are documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing software.

4. *Managed*: Predictable. Detailed software process and product quality metrics establish the quantitative evaluation foundation. Meaningful variations in process performance can be distinguished from random noise, and trends in process and product qualities can be predicted.

5. *Optimizing*: Characterized by Continuous Improvement. The organization has quantitative feedback systems in place to identify process weaknesses and strengthen them pro-actively. Project teams analyze defects to determine their causes; software processes are evaluated and updated to prevent known types of defects from recurring.

Captain Tom Schorsch of the US Air Force realized that the CMMI is just an optimistic subset of the true universe of development models. He discovered the CIMM—Capability Integration Immaturity Model—which adds four levels from 0 to −3:

0. *Negligent*: Indifference. Failure to allow successful development process to succeed. All problems are perceived to be technical problems. Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process.

−1. *Obstructive*: Counter Productive. Counter productive processes are imposed. Processes are rigidly defined and adherence to the form is stressed. Ritualistic ceremonies abound. Collective management precludes assigning responsibility.

−2. *Contemptuous*: Arrogance. Disregard for good software engineering institutionalized. Complete schism between software development activities and software process improvement activities. Complete lack of a training program.

−3. *Undermining*: Sabotage. Total neglect of own charter, conscious discrediting of organization's software process improvement efforts. Rewarding failure and poor performance.

If you've been in this business for a while, this extension to the CMMI may be a little too accurate to be funny…

The idea behind the CMMI is to find a defined way to predictably make good software. The words "predictable" and "consistently" are the keynotes of the CMMI. Even the most dysfunctional teams have occasional successes—generally surprising everyone. The key is to change the way we build embedded systems so we are *consistently* successful, and so we can reliably *predict* the code's characteristics (deadlines, bug rates, cost, etc.).

Figure 6.1 shows the result of using the tenants of the CMMI in achieving schedule and cost goals. In fact, Level 5 organizations don't always deliver on time. The probability of being on time, though, is high and the typical error bands low.

Compare this to the performance of a Level 1 (Initial) team. The odds of success are about the same as at the craps tables in Las Vegas. A 1997 survey in *EE Times* confirms this data in its report that 80% of embedded systems are delivered late.

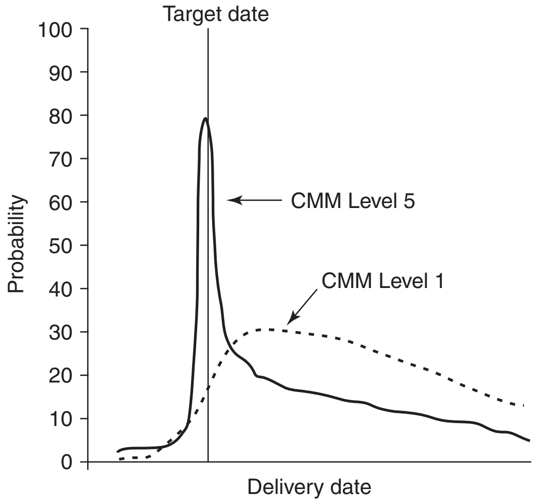


**Figure 6.1: Improving the process improves the odds of meeting goals and narrows the error bands**

One study of companies progressing along the rungs of the CMMI found *per year* results of:

- 37% gain in productivity
- 18% more defects found pre-test
- 19% reduction in time to market
- 45% reduction in customer-found defects

It's pretty hard to argue with results like these. Yet the vast majority of organizations are at Level 1. In my discussions with embedded folks I've found most are only vaguely aware of the CMMI. An obvious moral is to study constantly. Keep up with the state of the art of software development.

At the risk of being proclaimed a heretic and being burned at the stake of political incorrectness, I advise most companies to be wary of the CMMI. Despite its obvious benefits, the pursuit of CMMI is a difficult road all too many companies just cannot navigate. Problems include:

1. Without deep management commitment CMMI is doomed to failure. Since management rarely understands—or even cares about—the issues in creating

high quality software, their tepid buy-in all too often collapses when under fire from looming deadlines.

2. The path from level to level is long and torturous. Without a passionate technical visionary guiding the way and rallying the troops, individual engineers may lose hope and fall back on their old, dysfunctional software habits.

CMMI is a tool. Nothing more. Study it. Pull good ideas from it. Proselytize its virtues to your management. But have a backup plan you can realistically implement *now* to start building better code immediately. Postponing improvement while you "analyze options" or "study the field" always leads back to the status quo. Act now!

Solving problems is a high-visibility process; preventing problems is low-visibility. This is illustrated by an old parable:

In ancient China there was a family of healers, one of whom was known throughout the land and employed as a physician to a great lord. The physician was asked which of his family was the most skillful healer. He replied, "I tend to the sick and dying with drastic and dramatic treatments, and on occasion someone is cured and my name gets out among the lords."

"My elder brother cures sickness when it just begins to take root, and his skills are known among the local peasants and neighbors."

"My eldest brother is able to sense the spirit of sickness and eradicate it before it takes form. His name is unknown outside our home."

## 6.2 The Seven Step Plan

Arm yourself with one tool—one tool only—and you can make huge improvements in both the quality and delivery time of your next embedded project.

That tool is: *an absolute commitment to make some small but basic changes to the way you develop code*.

Given the will to change, here's what you should do *today*:

1. Buy and use a Version Control System.

2. Institute a Firmware Standards Manual.

3. Start a program of Code Inspections.
4. Create a quiet environment conducive to thinking.

More on each of these in a few pages. Any attempt to institute just one or two of these four ingredients will fail. All couple synergistically to transform crappy code to something you'll be proud of.

Once you're up to speed on steps 1–4, add the following:

5. Measure your bug rates.
6. Measure code production rates.
7. Constantly study software engineering.

Does this prescription sound too difficult? I've worked with companies that have implemented steps 1–4 in a single day. Of course they tuned the process over a course of months. That, though, is the very meaning of the word "process"—something that constantly evolves over time.

But the benefits accrue as soon as you start the process. Let's look at each step in a bit more detail.

### 6.2.1 Step 1: Buy and Use a VCS

Even a one-person shop needs a formal VCS (Version Control System). It is truly magical to be able to rebuild any version of a set of firmware, even one many years old. The VCS provides a sure way to answer those questions that pepper every bug discussion, like "when did this bug pop up?"

The VCS is a database hosted on a server. It's the repository of all of the company's code, make files, and the other bits and pieces that make up a project. There's no reason not to include hardware files as well—schematics, artwork, and the like.

A VCS insulates your code from the developers. It keeps people from fiddling with the source; it gives you a way to track each and every change. It controls the number of people working on modules and provides mechanisms to create a single correct module from one that has been (in error) simultaneously modified by two or more people.

Sure, you can sneak around the VCS, but like cheating on your taxes there's eventually a day of reckoning. Maybe you'll get a few minutes of time savings up front… inevitably followed by hours or days of extra time paying for the shortcut.

Never bypass the VCS. Check modules in and out as needed. Don't hoard checked-out modules "in case you need them." Use the system as intended, daily, so there's no VCS clean up needed at the project's end.

The VCS is also a key part of the file backup plan. In my experience it's foolish to rely on the good intentions of people to back-up religiously. Some are passionately devoted; others are concerned but inconsistent. All too often the data is worth more than all of the equipment in a building, even more than the building itself. Sloppy backups spell eventual disaster.

I admit to being anal-retentive about backups. A fire that destroys all of the equipment would be an incredible headache, but a guaranteed business-buster is the one that smokes the data.

Yet, preaching about data duplication and implementing draconian rules is singularly ineffective.

A VCS saves all project files on a single server, in the VCS database. Develop a backup plan that saves the VCS files each and every night. With the VCS there's but one machine whose data is life and death for the company, so the backup problem is localized and tractable. Automate the process as much as possible.

One Saturday morning I came into the office with two small kids in tow. Something seemed odd, but my disbelief masked the nightmare. Awakening from the fog of confusion I realized all of engineering's computers were missing! The entry point was a smashed window in the back. Fearful there was some chance the bandits were still in the facility, I rushed the kids next door and called the cops.

The thieves had made off with an expensive haul of brand-new computers, including the server that hosted the VCS and other critical files. The most recent backup tape, which had been plugged into the drive on the server, was also missing.

Our backup strategy, though, included daily tape rotation into a fireproof safe. After delighting the folks at Dell with a large emergency computer order, we installed the 1-day-old tape and came back up with virtually no loss of data.

If you have never had an awful, data-destroying event occur, just wait. It will surely happen. Be prepared.

#### 6.2.1.1 Checkpoint your tools

An often overlooked characteristic of embedded systems is their astonishing lifetime. It's not unusual to ship a product for a decade or more. This implies that you've got to be prepared to support old versions of every product.

As time goes on, though, the tool vendors obsolete their compilers, linkers, debuggers, and the like. When you suddenly have to change a product originally built with version 2.0 of the compiler—and now only version 5.3 is available—what are you going to do? The new version brings new risks and dangers. At the very least it will inflict your product with a host of unknowns. Are there new bugs? A new code generator means the real-time performance of the product will surely differ. Perhaps the compiled code is bigger, and no longer fits in ROM.

It's better to simply use the original compiler and linker throughout the product's entire lifecycle, so *preserve the tools*. At the end of a project check all of the tools into the VCS. It's cheap insurance.

When I suggested this to a group of engineers at a disk drive company, they cheered! Now that big drives cost virtually nothing there's no reason not to go heavy on the mass storage and save everything.

A lot of vendors provide VCS. But today the most popular is the open source Subversion (http://subversion.tigris.org). Another open source product, Trac (http://trac.edgewall .org), gives Subversion a Wiki front end with a better user interface.

The frenetic march of technology creates yet another problem we've largely ignored: today's media will be unreadable tomorrow. Save your tools on their distribution CDs or DVDs and surely in the not too distant future that media will be supplanted by some other, better technology. With time you'll be unable to find a CD reader.

The VCS lives on your servers, so migrates with the advance of technology. If you've been in this field for a while you've tossed out each generation of unreadable media: can you find a drive that will read an 8-inch floppy any more? How about a 160k 5-inch disk?

### 6.2.2 Step 2: Institute a Firmware Standards Manual

You can't write good software without a consistent set of code guidelines. Yet, the vast majority of companies has no standards, no written and enforced baseline rules.

A commonly cited reason is the lack of such standards in the public domain. So, I've removed this excuse by including a firmware standard in Appendix A.

See Chapter 3 for more discussion about standards.

### 6.2.3 Step 3: Use Code Inspections

Testing is important, but used alone will lead to products infested with bugs. Testing usually exercises about half the code.

The solution is a disciplined program of code inspections (see Chapter 3).

Everyone loves open source software, mostly because of the low bug rate. Remember the open source mantra: "with enough eyes all bugs are shallow."

That's what inspections are all about.

### 6.2.4 Step 4: Create a Quiet Work Environment

For my money the most important work on software productivity in the last 20 years is DeMarco and Lister's *Peopleware* (1987, Dorset House Publishing, NY). Read this slender volume, then read it again, and then get your boss to read it.

For a decade the authors conducted coding wars at a number of different companies, pitting teams against each other on a standard set of software problems. The results showed that, using any measure of performance (speed, defects, etc.), the average of those in the 1st quartile outperformed the average in the 4th quartile by a factor of 2.6. Surprisingly, none of the factors you'd expect to matter correlated to the best and worst performers. Even experience mattered little, as long as the programmers had been working for at least 6 months.

They did find a very strong correlation between the office environment and team performance. Needless interruptions yielded poor performance. The best teams had private (read "quiet") offices and phones with "off" switches. Their study suggests that quiet time saves vast amounts of money.

Think about this. The almost minor tweak of getting some quiet time can, according to their data, multiply your productivity by 260%! That's an astonishing result. For the same salary your boss pays you now, he'd get almost three of you.

The winners—those performing almost three times as well as the losers—had the following environmental factors:

| | 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 |

Too many of us work in a sea of cubicles, despite the clear data showing how ineffective they are. It's bad enough that there's no door and no privacy. Worse is when we're subjected to the phone calls of all of our neighbors. We hear the whispered agony as the poor sod in the cube next door wrestles with divorce. We try to focus on our work … but being human, the pathos of the drama grabs our attention till we're straining to hear the latest development. Is this an efficient use of an expensive person's time?

One correspondent told of how, when working for a Fortune 500 company, heavy hiring led to a shortage of cubicles for incoming programmers. One was assigned a manager's office, complete with window. Everyone congratulated him on his luck. Shortly a maintenance worker appeared and boarded up the window. The office police considered a window to be a luxury reserved for management, not engineers.

Dysfunctional? You bet.

Various studies show that after an interruption it takes, on average, around 15 minutes to resume a "state of flow"—where you're once again deeply immersed in the problem at hand. Thus, if you are interrupted by colleagues or the phone three or four times an hour, *you cannot get any creative work done!* This implies that it's impossible to do support and development concurrently.

Yet the cube police will rarely listen to data and reason. They've invested in the cubes, and they've made a decision, by God! The cubicles are here to stay!

This is a case where we can only wage a defensive action. Educate your boss but resign yourself to failure. In the meantime, take some action to minimize the downside of the environment. Here are a few ideas:

- Wear headphones and listen to music to drown out the divorce saga next door.
- Turn the phone off. If it has no "off" switch, unplug the damn thing. In desperate situations attack the wire with a pair of wire cutters. Remember that a phone is a bell that anyone in the world can ring to bring you running. Conquer this madness for your most productive hours.
- Know your most productive hours. I work best before lunch; that's when I schedule all of my creative work, all of the hard stuff. I leave the afternoons free for low-IQ activities like meetings, phone calls, and paperwork.
- Disable the email. It's worse than the phone. Your 200 closest friends who send the joke of the day are surely a delight, but if you respond to the email reader's "bing" you're little more than one of NASA's monkeys pressing a button to get a banana.
- Put a curtain across the opening to simulate a poor man's door. Since the height of most cubes is rather low, use a Velcro fastener or a clip to secure the curtain across the opening. Be sure others understand that when it's closed you are not willing to hear from anyone unless it's an emergency.

An old farmer and a young farmer are standing at the fence talking about farm-lore, and the old farmer's phone starts to ring. The old farmer just keeps talking about herbicides and hybrids, until the young farmer interrupts "Aren't you going to answer that?"

"What fer?" Says the old farmer.

"Why, 'cause it's ringing. Aren't you going to get it?" says the younger.

The older farmer sighs and knowingly shakes his head. "Nope," he says. Then he looks the younger in the eye to make sure he understands, "Ya see, I bought that phone for MY convenience."

Never forget that the phone is a bell that anyone in the world can ring to make you jump. Take charge of your time!

It stands to reason we need to focus to think, and that we need to think to create decent embedded products. Find a way to get some privacy, and protect that privacy above all.

When I use the Peopleware argument with managers they always complain that private offices cost too much. Let's look at the numbers.

DeMarco and Lister found that the best performers had an average of 78 square feet of private office space. Let's be generous and use 100. In the Washington DC area in 1998 nice—very nice—full service office space runs around $30/square foot/year.

Cost: 100 square feet: $3000/year = 100 sq ft * $30/ft/year

One engineer costs: $120,000 = $60,000 * 2 (overhead)

The office represents: 2.5% of cost of the worker = $3000/$120,000

Thus, if the cost of the cubicle is zero, then only a 2.5% increase in productivity pays for the office! Yet DeMarco and Lister claim a 260% improvement. Disagree with their numbers? Even if they are off by an order of magnitude a private office is 10 times cheaper than a cubicle.

You don't have to be a rocket scientist to see understand the true cost/benefit of private offices versus cubicles.

### 6.2.5 Step 5: Measure Your Bug Rates

Code inspections are an important step in bug reduction. But bugs—some bugs—will still be there. We'll never entirely eliminate them from firmware engineering.

Understand, though, that bugs are a natural part of software development. He who makes no mistakes surely writes no code. Bugs—or defects in the parlance of the software engineering community—are to be expected. It's OK to make mistakes, as long as we're prepared to catch and correct these errors.

Though I'm not big on measuring things, bugs are such a source of trouble in embedded systems that we simply have to log data about them. There are three big reasons for bug measurements:

1. We find and fix them too quickly. We need to slow down and think more before implementing a fix. Logging the bug slows us down a trifle.

2. A small percentage of the code will be junk. Measuring bugs helps us identify these functions so we can take appropriate action.

3. Defects are a sure measure of customer-perceived quality. Once a product ships we've got to log defects to understand how well our firmware processes satisfy the customer—the ultimate measure of success.

But first a few words about "measurements."

It's easy to take data. With computer assistance we can measure just about anything and attempt to correlate that data to forces as random as the wind.

Demming noted that using measurements as motivators is doomed to failure. He realized that there are two general classes of motivating factors: the first he called "intrinsic." This includes things like professionalism, feeling like part of a team, and wanting to do a good job. "Extrinsic" motivators are those applied to a person or team, such as arbitrary measurements, capricious decisions, and threats. Extrinsic motivators drive out intrinsic factors, turning workers into uncaring automatons. This may or may not work in a factory environment, but is deadly for knowledge workers.

So measurements are an ineffective tool for motivation.

Good measures promote *understanding*: to transcend the details and reveal hidden but profound truths. These are the sorts of measures we should pursue relentlessly.

But we're all very busy and must be wary of getting diverted by the measurement process. Successful measures have the following three characteristics:

1. They're easy to do.

2. Each gives insight into the product and/or processes.

3. The measure supports effective change-making. If we take data and do nothing with it, we're wasting our time.

For every measure think in terms of first *collecting* the data, then *interpreting* it to make sense of the raw numbers. Then figure on *presenting* the data to yourself, your boss, or your colleagues. Finally, be prepared to *act* on the new understanding.

#### *6.2.5.1 Stop, look, listen*

In the bad old days of mainframes, computers were enshrined in technical tabernacles, serviced by a priesthood of specially vetted operators. Average users never saw much beyond the punch card readers.

In those days of yore an edit-execute cycle started with punching perhaps thousands of cards, hauling them to the computer center (being careful not to drop the card boxes; on more than one occasion I saw grad students break down and weep as they tried to figure out how to order the cards splashed across the floor), and then waiting a day or more to see how the run went. Obviously, with a cycle this long no one could afford to use the machine to catch stupid mistakes. We learned to "play computer" (sadly, a lost art) to deeply examine the code before the machine ever had a go at it.

How things have changed! Found a bug in your code? No sweat—a quick edit, compile, and re-download take no more than a few seconds. Developers now look like hummingbirds doing a frenzied edit-compile-download dance.

It's wonderful that advancing technology has freed us from the dreary days of waiting for our jobs to run. Watching developers work, though, I see we've created an insidious invitation to bypass *thinking*.

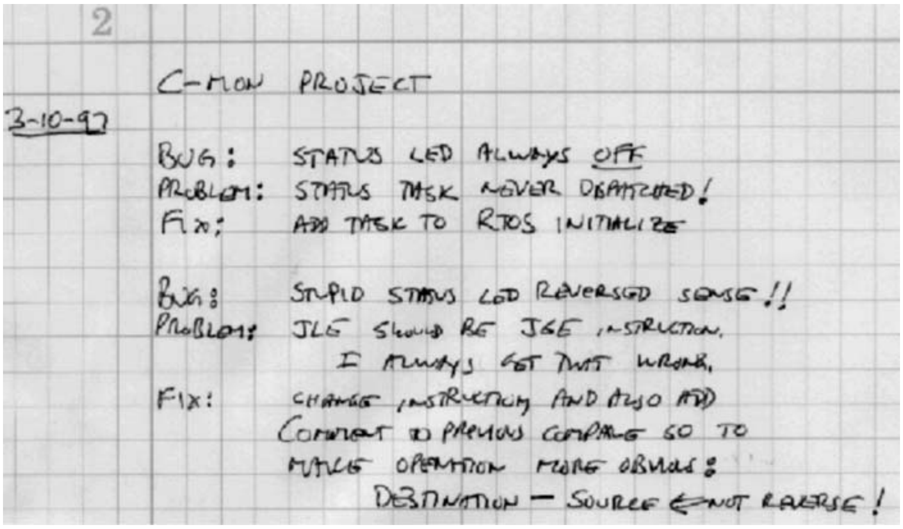
How often have you found a problem in the code, and thought "uh, if I change this maybe the bug will go away"? To me that's a sure sign of disaster. If the change fails to fix the problem, you're in good shape. The peril is when a poorly thought out modification does indeed "cure" the defect. Is it really cured? Did you just mask it?

Unless you've thought things through, *any* change to the code is an invitation to disaster.

Our fabulous tools enable this dysfunctional pattern of behavior. To break the cycle we have to slow down a bit.

The EEs traditionally keep engineering notebooks, bound volumes of numbered pages, ostensibly for patent protection reasons but more often useful for logging notes, ideas, and fixes. Firmware folks should do no less.

When you run into a problem, stop for a few seconds. Write it down. Examine your options and list those as well. Log your proposed solution (see Figure 6.2).



2

C-MON PROJECT

3-10-97

BUG: STATUS LED ALWAYS OFF
PROBLEM: STATUS TASK NEVER DISPATCHED!
FIX: ADD TASK TO RTOS INITIALIZE

BUG: STUPID STATUS LED REVERSED SENSE!!
PROBLEM: JLE SHOULD BE JGE INSTRUCTION.
I ALWAYS GET THAT WRONG.
FIX: CHANGE INSTRUCTION AND ALSO ADD
COMMENT TO PREVIOUS COMPARE SO TO
MAKE OPERATION MORE OBVIOUS:
DESTINATION – SOURCE ⟸ NOT REVERSE!

**Figure 6.2: A personal bug log**

Keeping such a journal helps force us to think things through more clearly. It's also a chance to reflect for a moment, and, if possible, come up with a way to avoid that sort of problem in the future.

One colleague recently fought a tough problem with a wild pointer. While logging the symptoms and ideas for fixing the code, he realized that this particular flavor of bug could appear in all sorts of places in the code. Instead of just plodding on, he set up a logic analyzer to trigger on the wild writes ... and found seven other areas with the same problem, all of which had not as yet exhibited a symptom. Now that's what I call a great debug strategy—using experience to predict problems!

### *6.2.5.2 Identify bad code*

Barry Boehm found that typically 80% of the defects in a program are in 20% of the modules. IBM's numbers showed 57% of the bugs are in 7% of modules. Weinberg's numbers are even more compelling: 80% of the defects are in 2% of the modules.

In other words, *most of the bugs will be in a few modules or functions*. These academic studies confirm our common sense. How many times have you tried to beat a function into submission, fixing bug after bug after bug, convinced that this one is (hopefully) the last?

We've all also had that awful function that just simply stinks. It's ugly. The one that makes you slightly nauseous every time you open it. A decent code inspection will detect most of these poorly crafted beasts, but if one slips through we have to take some action.

Make identifying bad code a priority. Then trash those modules and start over.

It sure would be nice to have the chance to write every program twice: the first time to gain a deep understanding of the problem; the second to do it right. Reality's ugly hand means that's not an option. But, the bad code, the code where we spend far too much time debugging, needs to be excised and redone. The data suggests we're talking about recoding only around 5% of the functions—not a bad price to pay in the pursuit of quality.

Boehm's studies show that these problem modules cost, on average, *four times* as much as any other module. So, if we identify these modules (by tracking bug rates) we can rewrite them *twice* and still come out ahead.

Bugzilla (http://www.bugzilla.org) is a free, open source, extremely popular bug tracking package. Use it with Scmbug (http://www.mkgnu.net/?q=scmbug) to integrate the bug tracking software with Subversion or any other VCS.

### 6.2.6 Step 6: Measure Your Code Production Rates

Schedules collapse for a lot of reasons. In the 50 years people have been programming electronic computers we've learned one fact above all: without a clear project specification any schedule estimate is nothing more than a stab in the dark. Yet every day dozens of projects start with little more definition than "well, build a new instrument kind of like the last one, with more features, cheaper, and smaller." Any estimate made to a vague spec is totally without value.

The corollary is that given the clear spec, we need time—sometimes lots of time—to develop an accurate schedule. It isn't easy to translate a spec into a design, and then to realistically size the project. You simply cannot do justice to an estimate in 2 days, yet that's often all we get.

Further, managers must accept schedule estimates made by their people. Sure, there's plenty of room for negotiation: reduce features, add resources, or permit more bugs (gasp). Yet most developers tell me their schedule estimates are capriciously changed by management to reflect a desired end date, with no corresponding adjustments made to the project's scope.

The result is almost comical to watch, in a perverse way. Developers drown themselves in project management software, mousing milestone triangles back and forth to meet an arbitrary date cast in stone by management. The final printout may look encouraging but generally gets the total lack of respect it deserves from the people doing the actual work. The schedule is then nothing more than dishonesty codified as policy.

There's an insidious sort of dishonest estimation too many of us engage in. It's easy to blame the boss for schedule debacles, yet often we bear plenty of responsibility. We get lazy, and don't invest the same amount of thought, time, and energy into scheduling that we give to debugging. "Yeah, that section's kind of like something I did once before" is, at best, just a start of estimation. You cannot derive time, cost, or size from such a vague statement … yet too many of us do. "Gee, that looks pretty easy—say a week" is a variant on this theme.

Doing less than a thoughtful, thorough job of estimation is a form of self-deceit, that rapidly turns into an institutionalized lie. "We'll ship December 1" we chant, while the estimators know just how flimsy the framework of that belief is. Marketing prepares glossy brochures, technical pubs writes the manual, production orders parts. December 1 rolls around, and, surprise! January, February, and March go by in a blur. Eventually the product goes out the door, leaving everyone exhausted and angry. Too much of this stems from a lousy job done in the first week of the project when we didn't carefully estimate its complexity.

It's time to stop the madness! (see Chapter 2 about scheduling).

Few developers seem to understand that knowing code size—even if it were 100% accurate—is only half of the data absolutely required to produce any kind of schedule. It's amazing that somehow we manage to solve the following equation:

$$\text{development time} = (\text{program size in line of code}) \times (\text{time per line of code})$$

when time-per-line-of-code is totally unknown.

If you estimate modules in terms of lines of code (LOC), then you must know—exactly—the cost per LOC. Ditto for function points or any other unit of measure. Guesses are not useful.

When I sing this song to developers the response is always "yeah, sure, but I don't have LOC data … what do I do about the project I'm on today?" There's only one answer: sorry, pal—you're outta luck. IBM's LOC/month number is useless to you, as is one from the FAA, DOD, or any other organization. In the commercial world we all hold our code to different standards, which greatly skews productivity in any particular measure.

You simply must measure how fast you generate embedded code. Every single day, for the rest of your life. It's like being on a diet—even when everything's perfect, you've shed those 20 extra pounds, you'll forever be monitoring your weight to stay in the desired range. Start collecting the data today, do it forever, and over time you'll find a model of your productivity that will greatly improve your estimation accuracy. Don't do it, and every estimate you make will be, in effect, a lie, a wild, meaningless guess.

### 6.2.7 Step 7: Constantly Study Software Engineering

The last step is the most important. Study constantly. In the 50 years since ENIAC we've learned a lot about the right and wrong ways to build software; almost all of the lessons are directly applicable to firmware development.

How does an elderly, near-retirement doctor practice medicine? In the same way he did before World War II, before penicillin? Hardly. Doctors spend a lifetime learning. They understand that lunch time is always spent with a stack of journals.

Like doctors, we too practice in a dynamic, changing environment. Unless we master better ways of producing code we'll be the metaphorical equivalent of the 16th century medicine man, trepanning instead of practicing modern brain surgery.

Learn new techniques. Experiment with them. Any idiot can write code; the geniuses are those who find better ways of writing code.

One of the more intriguing approaches to creating a discipline of software engineering is the Personal Software Process, a method created by Watts Humphrey. An original architect of the CMMI, Humphrey realized that developers need a method they can use *now*, without waiting for the CMMI revolution to take hold at their company. His vision is not easy, but the benefits are profound. Check out his *A Discipline for Software Engineering,* Watts S. Humphrey, 1995, Addison-Wesley.

### *6.2.8 Summary*

With a bit of age it's interesting to look back, and to see how most of us form personalities very early in life, personalities with strengths and weaknesses that largely stay intact over the course of decades.

The embedded community is composed of mostly smart, well-educated people, many of whom believe in some sort of personal improvement. But, are we successful? How many of us live up to our New Year's resolutions?

Browse any bookstore. The shelves groan under self-help books. How many people actually get helped, or at least helped to the point of being done with a particular problem? Go to the diet section—I think there are more diets being sold than the sum total of national excess pounds. People buy these books with the best of intentions, yet every year America gets a little heavier.

Our desires and plans for self-improvement—at home or at the office—are among the more noble human characteristics. The reality is that we fail—a lot. It seems the most common way to compensate is a promise made to ourselves to "try harder" or to "do better." It's rarely effective.

Change works best when we change the way we do things. Forget the vague promises—invent a new way of accomplishing your goal. Planning on reducing your drinking? Getting regular exercise? Develop a process that insures you're meeting your goal.

The same goes for improving your abilities as a developer. Forget the vague promises to "read more books" or whatever. Invent a solution that has a better chance of succeeding. Even better—steal a solution that works from someone else.

Cynicism abounds in this field. We're all self-professed experts of development, despite the obvious evidence of too many failed projects.

I talk to a lot of companies who are convinced that change is impossible, that the methods I espouse are not effective (despite the data that shows the contrary), or that management will never let them take the steps needed to effect change.

That's the idea behind the "Seven Steps." Do it covertly, if need be; keep management in the dark if you're convinced of their unwillingness to use a defined software process to create better embedded projects faster.

If management is enlightened enough to understand that the firmware crisis requires change—and lots of it!—then educate them as you educate yourself.

Perhaps an analogy is in order. The industrial revolution was spawned by a lot of forces, but one of the most important was the concentration of capital. The industrialists spent vast sums on foundries, steel mills, and other means of production. Though it was possible to handcraft cars, dumping megabucks into assembly lines and equipment yielded lower prices, and eventually paid off the investment in spades.

The same holds true for intellectual capital. Invest in the systems and processes that will create massive dividends over time. If we're unwilling to do so, we'll be left behind while others, more adaptable, put a few bucks up front and win the software wars.

A final thought:

If you're a process cynic, if you disbelieve all I've said in this chapter, ask yourself one question: do I consistently deliver products on time and on budget?

If the answer is no, then *what are you doing about it*?

## 6.3 The Postmortem

The TV camera pans across miles of woodland, showing ghastly images of wreckage. Some is identifiable: the remnants of an engine, a child's doll, scattered papers from a businessperson's briefcase; much is not. The reporter, on a mission to turn tragedy into a career, breathlessly pours facts and speculation into the microphone. Shocked viewers swear off air travel till time diminishes their sense of horror.

Yet the disaster, a calamity of ineffable proportions to those left waiting for loved ones who never come home, is in fact a success of sorts. The NTSB searches for and finds the black boxes that record the flight's final moments, and over the course of months or years reconstructs the accident. We've all seen the stunning computer-generated final moments of a plane's crash on the Discovery Channel. Experts find the root cause of the incident … and then change something. Maybe there's a mechanical flaw in the plane's structure, perhaps an electrical fire initiated the accident. The FAA issues instructions to the aircraft's builders and users to implement an engineering change.

Perhaps the pilots were confused by their instrumentation, or they handled the wind shear incorrectly. Maybe maintenance people serviced a control surface incorrectly. Or perhaps it was found that Americans are getting fat so old loading guidelines no longer apply (as was recently the case in one incident). Changes are made to training or procedures. This sort of accident never happens again.

A jet cruises in the sparse air at 40,000 feet where it's 60 below zero. Four hundred thousand pounds of aluminum traveling at 600 knots relies on a complex web of wiring, electronics, mechanics, and plumbing to keep the passengers safe. It's astonishing a modern plane works at all, yet air travel is the safest form of transportation ever invented. The reason is the feedback loop that turns accidents into learning experiences.

Contrast the airplane accident with the carnage on our roads—over 40,000 people are killed in the United States of America each year in car crashes; another 2 million are injured. The accident ends with the car crash (plus enduring litigation); we learn nothing from either, we take no important lessons away, we make no changes in the way we drive. Traffic slows around the emergency crews cutting a twisted body from the smashed car, but then we're soon standing hard on the accelerator again, weaving in and out of traffic inches from the bumper ahead, in a manic search to save time that may shave, at best, a few seconds from the commute.

Carmakers do improve the safety of their vehicles by adding crumple zones and air bags, but the essential fact is that the danger sprouts from poor driving. The car and driver represent a system without feedback, running wildly out of control.

Feedback stabilizes systems. Every EE knows this. Amplifiers all use negative feedback to control their output. An oscillator has positive feedback, and so, well, oscillates.

Feedback stabilizes human systems as well. The IRS's pursuit of tax cheats keeps most 1040s relatively honest. A recent awful crash on my street led to a week or two of radar enforcement. Speeds dropped to the mandated 30 mph, but the police soon moved on to other neighborhoods.

Feedback does—or should—stabilize embedded development efforts. Most of the teams I see work madly on a project, delivering late and buggy. The boss is angry and customers are screaming. Yet as soon as the thing gets out the door we immediately start developing another project. There's neither feedback nor introspection.

Resumes abound with "experience"; often that engineer with two-dozen projects and 20 years behind him actually has had the same experience time after time. The same old heroics and the same bad decisions form the fabric of his career.

Is it any wonder so few systems go out on time?

### *6.3.1 Engineering Managers*

In most organizations the engineering managers are held accountable for getting the products out in the scheduled time, at a budgeted cost, with a minimal number of bugs. These are noble, important goals.

How often, though, are the managers encouraged—no, *required*—to improve the *process* of designing products?

The Total Quality movement in many companies seems to have bypassed engineering altogether. Every other department is held to the cold light of scrutiny, and the processes tuned to minimize wasted effort. Engineering has a mystique of dealing with unpredictable technologies and workers immune to normal management controls. Why can't R&D be improved just like production and accounting?

Now, new technologies are a constant in this business. These technologies bring risks, risks that are tough to identify, let alone quantify. We'll always be victims of unpredictable problems.

Worse, software is very difficult to estimate. Few of us have the luxury to completely and clearly specify a project before starting. Even fewer don't suffer from creeping featurism as the project crawls toward completion.

Unfortunately, most engineering departments use these problems as excuses for continually missing goals and deadlines. The mantra "engineering is an art, not a science" weaves a spell that the process of development doesn't lend itself to improvement.

Phooey.

Engineering management is about removing obstacles to success. Mentoring the developers. Acquiring needed resources.

It's also about closing feedback loops. Finding and removing dysfunctional patterns of operation. Discovering new, better ways to get the work done.

Doing things the same old way is a prescription for getting the same old results.

### *6.3.2 Postmortems*

How do developers go about learning more about their craft? Buy a pile of books, perhaps read some of them, peruse the magazines, go to conferences, bring in outside gurus. These are all great and necessary steps. But it's astonishing that most refuse to learn from their own actions.

A company may spend hundreds of thousands to millions developing a project. Many things will go right and too many wrong during the work. Wise developers understand that their engineering group does indeed make products, but is also a laboratory where experiments are always in progress. Each success is a Eureka moment, and each failure a chance to gain insight into how not to do development. Edison commented that, though he had had 1000 failures in his pursuit of some new invention, he had also learned 1000 things that do not work.

We can fool ourselves into thinking that each of these success/failure moments is a powerful learning tool. Sure, we take away some insight. But this is a casual way to learn, one that's personal and so of no benefit to other team members.

I prefer to acquire experience scientifically. Firmware development is too expensive to take any other approach. We must use the development environment as a laboratory to discover solutions to our problems. This means all projects should end with a postmortem, a process designed to suck the educational content of a particular development effort dry.

The postmortem is a formal process that starts during the project itself. Collect data artifacts as they are generated—for instance, the estimated schedule, the bug logs, and change requests. Include technical information as well, such as the estimated size (in lines of code and in object file bytes) versus actuals, real-time performance results, tool issues, etc.

After the product is released schedule the postmortem. Do it immediately upon project completion while memories are still fresh and before the team disbands (especially in matrix organizations). My rule of thumb is to do the postmortem no more than 3 days after project completion.

Management must support the process and must make it clear this work is important. Dysfunctional organizations that view firmware as a necessary evil will try to subvert anything that's not directly linked to writing code. In this case run a stealth postmortem, staying under the screens of the top dogs. If even the team lead doesn't buy into this sort of process-improvement endeavor, I guess you're doomed and might as well start looking for a better job.

A facilitator runs the postmortem. In many activities I advocate rotating all team members through the moderator/leader role, even those soft-spoken individuals afraid to participate in verbal exchanges. It's a great way to teach folks better social and leadership skills. But postmortems tend to fail without a strong leader running the show. Use the team lead, or perhaps a developer well-respected by the entire group, one who is able to run a meeting.

All of the developers participate in the postmortem. We're trying to maximize the benefits, so everyone is involved and everyone learns the resulting lessons. In some cases it might make sense to bring in folks involved in the project in other ways, such as the angry customer or QA people.

The facilitator first makes it clear there are but two ways to get into trouble. First, it is the end of the project, probably late, we're all tired and hate each other. Despite this everyone must put in a few more hours of hard work, as the postmortem is so important. Slack off and you'll get zinged. Second, obstruct or trash the process and expect to be fired. "Yeah, this is just another stupid process thing that is a waste of time" is a clear indication you're not interested in improving. We don't want developers who insist on remaining in a stasis field.

He or she also insures the postmortem isn't used to beat up on a particular developer who might have been a real problem on the project. The fundamental rule of management must apply: praise publicly, discipline privately. Deal with problem people off-line.

Hold a sort of history-day meeting. Run by the facilitator, it's where we look at the problems encountered during the project. The data that was acquired during the effort is a good source of quantitative insight into the issues.

This is not a complaint session. The facilitator must be strong enough to quash criticisms and grumbling. It's also not an attempt to solve any problem. Rather, identify problems that appear solvable. Pick a few that promise the maximum return on investment.

Resist the temptation to solve all of the ills suffered during the project. I'm a child of the 1960s. At the time we thought we could save the world—we couldn't. But it was possible to implement small changes, to make some things better. Don't expect any one postmortem to lead you to firmware nirvana. Postmortems are baby steps we take to move to a higher plane. Try to do too much and the effort will collapse.

Pick a few problems—depending on the size of the group—maybe 2, 3, or 4. Break the team into groups, with each group tasked to crack a single issue.

The groups must focus on creating solutions that are implementable, and that are comprised of action items. If the project suffered from ending streams of changes from the 23-year-old marketing droid, a solution like "stop accepting changes," or "institute a change control process" is useless. These are nice sentiments but will never bear fruition.

Create plans specifying particular actions. "Joe evaluates change control tools by April 1. Selects one. Trains entire team on the process by April 15. No uncontrolled changes accepted after that date."

If each group comes back and presents its solutions to the entire team, the postmortem process will absolutely fail. We engineers have huge egos. Each of us knows we can solve any problem better than almost anyone else. If team A comes in and tells me how to fix myself I'll immediately toss out a dozen alternate approaches. The meeting will descend into chaos and nothing will result.

Instead, before making any presentations, team A solicits input on its idea from each developer. This is low-key, done via water-cooler meetings. The team is looking for both ideas and buy-in. Use Congress as a model: nothing happens on the House floor. All negotiations take place in back rooms, so when the vote occurs on the floor it's all but a fait accompli.

A final meeting is held, at which time the solutions are presented and recorded. In writing.

End with the post-project party. What! You don't do those? The party is an essential part of maintaining a healthy engineering group. All ones and zeros makes Joe a dull boy. The party eases tensions created by the intense work environment. But it happens only after the project is completely finished, including the postmortem.

The postmortem is done, the team disbands. Now the most important part of the postmortem begins. That's the closing of the loop, the employment of feedback to improve future projects. When the next development effort starts, the leader and all team members should—must!—read through all of the prior postmortems. This is the chance to avoid mistakes and to learn from the past. A report that's filed away in a dusty cabinet never to surface is a waste of time.

A 1999 study ("Techniques and Recommendations for Implementing Valuable Postmortems in Software Development Projects" by Gloria H. Congdon, Masters Thesis at the University of Minnesota, May 1999) showed that of 56 postmortems the developers found 89% of them very worthwhile. The 11% failed ones are the most interesting—developers rated them bad to awful because there was no follow-through. The postmortem took place but the results were ignored.

Enlightened management or those companies lucky enough to have a healthy process group will use the accumulated postmortems outside of project planning to synthesize risk templates. If a pattern like "every time we pick a new CPU we have massive tool problems" emerges, then it's reasonable to suggest never changing CPUs, or taking some other action to mitigate this problem.

Plane crashes, though tragic, are used in a healthy way to prevent future accidents, to save future lives. Shouldn't we employ a similar feedback mechanism to save future projects, to learn new ways to be more effective developers?

APPENDIX A

# A Firmware Standard

## A.1 Scope

This document defines the standard way all programmers will create embedded firmware. Every programmer is expected to be intimately familiar with the Standard, and to understand and accept these requirements. All consultants and contractors will also adhere to this Standard.

The reason for the Standard is to insure all Company-developed firmware meets minimum levels of readability and maintainability. Source code has two equally important functions: it must *work*, and it must clearly *communicate how it works* to a future programmer or the future version of yourself. Just as a standard English grammar and spelling make prose readable, standardized coding conventions ease readability of one's firmware.

Part of every code review is to insure the reviewed modules and functions meet the requirements of the Standard. Code that does not meet this Standard will be rejected.

No Standard can cover every eventuality. There may be times where it makes sense to take exception to one or more of the requirements incorporated in this document. Every exception must meet the following requirements:

- *Clear Reasons*: Before making an exception to the Standard, the programmer(s) will clearly spell out and understand the reasons involved and will communicate these reasons to the project manager. The reasons must involve clear benefit to the project and/or Company; stylistic motivations or programmer preferences and idiosyncrasies are not adequate reasons for making an exception.

- *Approval*: The project manager will approve all exceptions made.
- *Documentation*: Document the exception in the comments, so during code reviews and later maintenance the technical staff understands the reasons and nature of the exception.

## A.2 Projects

### A.2.1 Directory Structure

To simplify use of a version control system, and to deal with unexpected programmer departures and sicknesses, every programmer involved with each project will maintain identical directory structures for the source code associated with the project.

The general "root" directory for a project takes the form:

```
/projects/project-name/rom_name
```

where

- `/projects` is the root of all firmware developed by the Company. By keeping all projects under one general directory, version control and backup are simplified; it also reduces the size of the computer's root directory.
- `/project-name` is the formal name of the project under development.
- `/rom_name` is the name of the ROM the code pertains to. One project may involve several microprocessors, each of which has its own set of ROMs and code. Or, a single project may have multiple binary images, each of which goes into its own set of ROMs.

Required directories:

`/projects/project-name/tools`—compilers, linkers, assemblers used by this project. All tools will be checked into the VCS (version control system) so in 5 to 10 years, when a change is required, the (now obsolete and unobtainable) tools will still be around. It's impossible to recompile and retest the project code every time a new version of the compiler or assembler comes out; the only alternative is to preserve old versions, forever, in the VCS.

`/projects/project-name/rom_name/headers`—all header files, such as .h or assemble include files, go here.

`/projects/project-name/rom_name/source`—source code. This may be further broken down into header, C, and assembly directories. The MAKE files are also stored here.

`/projects/project-name/rom_name/object`—object code, including compiler/assembler objects and the linked and located binaries.

`/projects/project-name/rom_name/test`—This directory is the one, and only one, that is not checked into the VCS and whose subdirectory layout is entirely up to the individual programmer. It contains work-in-progress, which is generally restricted to a single module. When the module is released to the VCS or the rest of the development team, the developer must clean out the directory and eliminate any file that is duplicated in the VCS.

### *A.2.2 Version File*

Each project will have a special module that provides firmware version name, version date, and part number (typically the part number on the ROM chips). This module will list, in order (with the newest changes at the top of the file), all changes made from version to version of the released code.

Remember that the production or repair departments may have to support these products for years or decades. Documentation gets lost and ROM labels may come adrift. To make it possible to correlate problems to ROM versions, even after the version label is long gone, the Version file should generate only one bit of "code"—a string that indicates, in ASCII, the current ROM version. Some day in the future a technician—or you yourself!—may then be able to identify the ROM by dumping the ROM's contents. An example definition is:

```
# undef VERSION
# define VERSION "Version 1.30"
```

The Version file also contains the Abbreviations Table. See Section A.4 for more detail.

Example:

```
/***************************************************
Version Module-Project SAMPLE

Copyright 2007 Company
All Rights Reserved

The information contained herein is confidential
property of Company. The use, copying, transfer or
disclosure of such information is prohibited except
by express written agreement with Company.

12/18/07-Version 1.3-ROM ID 78-130
           Modified module AD_TO_D to fix scaling
           algorithm; instead of y=mx, it now
           computes y=mx+b.
10/29/07-Version 1.2-ROM ID 78-120
           Changed modules DISPLAY_LED and READ_DIP
           to incorporate marketing's request for a
           diagnostics mode.
09/03/07-Version 1.1-ROM ID 78-110
           Changed module ISR to properly handle
           non-reentrant math problem.
07/12/07-Version 1.0-ROM ID 78-100
           Initial release
***************************************************/
# undef VERSION
# define VERSION "Version 1.30"
```

### A.2.3 Make and Project Files

Every executable will be generated via a MAKE file, or the equivalent supported by the tool chain selected. The MAKE file includes all of the information needed to automatically build the entire ROM image. This includes compiling and assembling source files, linking, locating (if needed), and whatever else must be done to produce a final ROM image.

An alternative version of the MAKE file may be provided to generate debug versions of the code. Debug versions may include special diagnostic code, or might have a somewhat different format of the binary image for use with debugging tools.

In integrated development environments (like Visual C++) specify a PROJECT file that is saved with the source code to configure all MAKE-like dependencies.

In no case is any tool *ever* to be invoked by typing in a command, as invariably command line arguments "accumulate" over the course of a project … only to be quickly forgotten once version 1.0 ships.

### A.2.4 Startup Code

Most ROM code, especially when a C compiler is used, requires an initial startup module that sets up the compiler's run-time package and initializes certain hardware on the processor itself, including chip selects, wait states, etc.

Startup code generally comes from the compiler or locator vendor, and is then modified by the project team to meet specific needs of the project. It is invariably compiler- and locator-specific. Therefore, the first modification made to the startup code is an initial comment that describes the version numbers of all tools (compiler, assembler, linker, and locator) used.

Vendor-supplied startup code is notoriously poorly documented. To avoid creating difficult-to-track problems, *never* delete a line of code from the startup module. Simply comment-out unneeded lines, being careful to put a note in that you were responsible for disabling the specific lines. This will ease re-enabling the code in the future (for example, if you disable the floating point package initialization, one day it may need to be brought back in).

Many of the peripherals may be initialized in the startup module. Be careful when using automatic code generation tools provided by the processor vendor (tools that automate chip select setup, for example). Since many processors boot with RAM chip selects disabled, always include the chip select and wait state code in-line (not as a subroutine). Be careful to initialize these selects at the very top of the module, to allow future subroutine calls to operate, since some debugging tools will not operate reliably until these are set up.

### A.2.5 Stack and Heap Issues

Always initialize the stack on an *even* address. Resist the temptation to set it to an odd value like `0Xffff`, since on a word machine an odd stack will cripple system performance.

Since few programmers have a reasonable way to determine maximum stack requirements, always assume your estimates will be incorrect. For each stack in the system, make sure the initialization code fills the entire amount of memory allocated to the stack with the value 0×55. Later, when debugging, you can view the stack and detect stack overflows by seeing no blocks of 0×55 in that region. Be sure, though, that the code that fills the stack with 0×55 automatically detects the stack's size, so a late night stack size change will not destroy this useful tool.

Embedded systems are often intolerant of heap problems. Dynamically allocating ad freeing memory may, over time, fragment the heap to the point that the program crashes due to an inability to allocate more RAM. (Desktop programs are much less susceptible to this as they typically run for much shorter periods of time.)

So, be wary of the use of the `malloc()` function. When using a new tool chain examine the malloc function, if possible, to see if it implements garbage collection to release fragmented blocks (note that this may bring in another problem, as during garbage collection the system may not be responsive to interrupts). *Never* blindly assume that allocating and freeing memory is cost- or problem-free.

If you chose to use `malloc()`, always check the return value and safely crash (with diagnostic information) if it fails.

Consider using a replacement `malloc()`, such as the ones available from http://members.chello.nl/h.robbers and http://www.fourmilab.ch/bget/

When using C, if possible (depending on resource issues and processor limitations), always include Walter Bright's MEM package (http://c.snippets.org/browser.php) with the code, at least for the debugging.

MEM provides:

- ISO/ANSI verification of allocation/reallocation functions
- Logging of all allocations and frees
- Verifications of frees
- Detection of pointer over- and under-runs

- Memory leak detection
- Pointer checking
- Out of memory handling

## A.3 Modules

### A.3.1 General

A *Module* is a single file of source code that contains one or more functions or routines, as well as the variables needed to support the functions.

Each module contains a number of *related* functions. For instance, an A/D converter module may include all A/D drivers in a single file. Grouping functions in this manner makes it easier to find relevant sections of code and allows more effective encapsulation.

Encapsulation—hiding the details of a function's operation, and keeping the variables used by the function local—is absolutely essential. Though C and assembly language don't explicitly support encapsulation, with careful coding you can get all of the benefits of this powerful idea as do people using OOP languages.

In C and assembly language you can define all variables and RAM inside the modules that use those values. Encapsulate the data by defining each variable for the scope of the functions that use these variables only. Keep them private within the function, or within the module, that uses them.

Modules tend to grow large enough that they are unmanageable. Keep module sizes under 1000 lines to insure tools (source debuggers, compilers, etc.) are not stressed to the point they become slow or unreliable, and to enhance clarity.

### A.3.2 Templates

To encourage a uniform module look and feel, create module templates named `module_template.c` and `module_template.asm`, stored in the source directory, that becomes part of the code base maintained by the VCS. Use one of these files as the base for all new modules. The module template includes a standardized form for the header (the comment block preceding all code), a standard spot for file includes and module-wide declarations, function prototypes, and macros. The templates also include the standard format for functions.

Here's the template for C code:

```
/***************************************************
Module name:

Copyright 2007 Company as an unpublished work.
All Rights Reserved.

The information contained herein is confidential
property of Company. The user, copying, transfer or
disclosure of such information is prohibited except
by express written agreement with Company.

First written on xxxxx by xxxx.

Module Description:
(fill in a detailed description of the module's
function here).

***************************************************/
/* Include section
Add all #includes here

***************************************************/
/* Defines section
Add all #defines here

***************************************************/
/* Function Prototype Section
Add prototypes for all *** called by this
module, with the exception of runtime routines.

***************************************************/
```

The template includes a section defining the general layout of functions, as follows:

```
/**************************************************
Function name   :TYPE foo(TYPE arg1, TYPE arg2…)
  returns       :return value description
  arg1                :description
  arg2                :description
Created by      :author's name
Date created          :date
Description     :detailed description
Notes           :restrictions, odd modes
**************************************************/
```

The template for assembly modules is:

```
;***************************************************
; Module name:
;
; Copyright 2007 Company as an unpublished work.
; All Rights Reserved.
;
; The information contained herein is confidential
; property of Company. The user, copying, transfer or
; disclosure of such information is prohibited except
; by express written agreement with Company.
;
; First written on xxxxx by xxxx.
;
; Module Description:
; (fill in a detailed description of the module
; here).
;
;****************************************************
; Include section
; Add all "includes" here
;****************************************************
```

```
The template includes a section defining the general layout of
functions, as follows:
```

```
;****************************************************
; Routine name          :foobar
;  returns       :return value(s) description
;  arg1                 :description of arguments
;  arg2                 :description
; Created by     :author's name
; Date created          :date
; Description    :detailed description
; Notes                 :restrictions, odd modes
;***************************************************
```

### A.3.3 Module Names

Never include the project's name or acronym as part of each module name. It's much better to use separate directories for each project.

Big projects may require many dozens of modules; scrolling through a directory listing looking for the one containing function `main()` can be frustrating and confusing. Therefore store function `main()` in a module named main.c or main.asm.

Filenames will be all lowercase to enhance portability between Windows and Linux/UNIX systems.

File extensions will be:

| | |
|---|---|
| C Source Code | filename.c |
| C Header File | filename.h |
| Assembler files | filename.asm |
| Assembler include files | filename.inc |
| Object Code | filename.obj |
| Libraries | filename.lib |
| Shell Scripts | filename.bat |
| Directory Contents | README |
| Build rules for make | project.mak |

## A.4 Variables

### *A.4.1 Names*

Regardless of language, use long names to clearly specify the variable's meaning. If your tools do not support long names, get new tools.

Separate words within the variables by underscores. Do not use capital letters as separators. Consider how much harder `IcantReadThis` is on the eyes versus `I_can_read_this`.

Variable and function names are defined with the first words being descriptive of broad ideas, and later words narrowing down to specifics. For instance: `Universe_Galaxy_System_Planet`. Consider the following names: `Timer_0_Data`, `Timer_0_Overflow`, and `Timer_0_Capture`. This convention quickly narrows variables to particular segments of the program. Never assume that a verb must be first, as often seen when naming functions. `Open_Serial_Port` and `Close_Serial_Port` do a much poorer job of grouping than the better alternative of `Serial_Port_Open` and `Serial_Port_Close`.

Acronyms and abbreviations are not allowed as parts of variable names unless:

1. defined in a special Abbreviations Table which is stored in the Version file
2. an accepted industry convention like LCD, LED, and DSP

Clarity is our goal! An example Abbreviation Table is:

```
/* Abbreviation Table
Dsply  ==  Display (the verb)
Disp   ==  Display (our LCD display)
Tot    ==  Total
Calc   ==  Calculation
Val    ==  Value
Pos    ==  Position
*/
```

The ANSI C specification restricts the use of names that begin with an underscore and either an uppercase letter or another underscore (_[A-Z_][0-9A-Za-z_]). Much compiler run-time code also starts with leading underscores. To avoid confusion, never name a variable or function with a leading underscore.

These names are also reserved by ANSI for its future expansion:

| | |
|---|---|
| E[0–9A–Z][0–9A–Za–z]* | Errno values |
| is[a–z][0–9A–Za–z]* | Character classification |
| to[a–z][0–9A–Za–z]* | Character manipulation |
| LC_[0–9A–Za–z_]* | Locale |
| SIG[_A–Z][0–9A–Za–z_]* | Signals |
| str[a–z][0–9A–Za–z_]* | String manipulation |
| mem[a–z][0–9A–Za–z_]* | Memory manipulation |
| wcs[a–z][0–9A–Za–z_]* | Wide character manipulation |

## A.4.2 Global Variables

All too often C and especially assembly programs have one huge module with all of the variable definitions. Though it may seem nice to organize variables in a common spot, the peril is these are all then global in scope. Global variables are responsible for much undebuggable code, reentrancy problems, global warming, and male pattern baldness. Avoid them!

Real time code may occasionally require a few—and only a few—global variables to insure reasonable response to external events. *Every global variable must be approved by the project manager*.

When globals are used, put all of them into a single module. They are so problematic that it's best to clearly identify the sin via the name `globals.c` or `globals.asm`.

### A.4.3 Portability

Avoid the use of "int" and "long", as these declarations vary depending on the machine. Create typedefs as follows:

| | signed | Unsigned |
|---|---|---|
| 8 bit: | int8_t | uint8_t |
| 16 bit: | int16_t | uint16_t |
| 32 bit: | int32_t | uint32_t |
| 64 bit: | int64_t | uint64_t |

Don't assume that the address of an int object is also the address of its least significant byte. This is not true on big-endian machines.

## A.5 Functions

Regardless of language, *keep functions small!* The ideal size is less than a page; in no case should a function ever exceed two pages. Break large functions into several smaller ones.

The only exception to this rule is the very rare case where real-time constraints (or sometimes stack limitations) mandate long sequences of in-line code. The project manager must approve all such code … but first look hard for a more structured alternative!

Explicitly declare every parameter passed to each function. Clearly document the meaning of the parameter in the comments.

Define a prototype for every called function, with the exception of those in the compiler's run-time library. Prototypes let the compiler catch the all-too-common errors of incorrect argument types and improper numbers of arguments. They are cheap insurance.

In general, function names should follow the variable naming protocol.

## A.6 Interrupt Service Routines

ISRs, though usually a small percentage of the code, are often the hardest bits of firmware to design and debug. Crummy ISRs will destroy the project schedule!

Decent interrupt routines, though, require properly designed hardware. Sometimes it's tempting to save a few gates by letting the external device just toggle the interrupt line for a few microseconds. This is unacceptable. Every interrupt must be latched until acknowledged, either by the processor's interrupt-acknowledge cycle (be sure the hardware acks the proper interrupt source), or via a handshake between the code and the hardware.

Use the non-maskable interrupt (NMI) only for catastrophic events, like the apocalypse or imminent power failure. Many tools cannot properly debug NMI code. Worse, NMI is guaranteed to break non-reentrant code.

If at all possible, design a few spare I/O bits in the system. These are tremendously useful for measuring ISR performance.

Keep ISRs short! Long (too many lines of code) and slow are the twins of ISR disaster. Remember that *long* and *slow* may be disjoint; a five-line ISR with a loop can be as much of a problem as a loop-free 500-line routine. When an ISR grows too large or too slow, spawn another task and exit. Large ISRs are a sure sign of a need to include an RTOS.

Budget time for each ISR. Before writing the routine, understand just how much time is available to service the interrupt. Base all of your coding on this, and then *measure* the resulting ISR performance to see if you met the system's need. Since every interrupt competes for CPU resources, that slow ISR that works is just as buggy as one with totally corrupt code.

Never allocate or free memory in an ISR unless you have a clear understanding of the behavior of the memory allocation routines. Garbage collection or the ill-behaved behavior of many run-time packages may make the ISR time non-deterministic.

On processors with interrupt vector tables, fill every entry of the table. Point those entries not used by the system to an error handler, so you've got a prayer of finding problems due to incorrectly programmed vectors in peripherals.

Though non-reentrant code is always dangerous in a real time system, it's often unavoidable in ISRs. Hardware interfaces, for example, are often non-reentrant. Put all such code as close to the beginning of the ISR as possible, so you can then re-enable interrupts. Remember that as long as interrupts are off, the system is not responding to external requests.

## A.7 Comments

Code *implements* an algorithm; the comments *communicate* the code's operation to your and others. Adequate comments allow you to understand the system's operation without having to read the code itself.

Write comments in *clear English*. Use the sentence structure Miss Grandel tried to pound into your head in grade school. Avoid writing the Great American Novel; be concise yet explicit … but be complete.

Avoid long paragraphs. Use simple sentences: noun, verb, object. Use active voice: "Motor_Start actuates the induction relay after a 4 second pause." Be complete. Good comments capture everything important about the problem at hand.

Use proper case. Using all caps or all lowercase simply makes the comments harder to read and makes the author look like an illiterate moron.

Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines which work together to perform a macro function. However, never assume that long variable names create "self-documenting code." Self-documenting code is an oxymoron, so add comments where needed to make the firmware's operation crystal clear. It should be possible to get a sense of the system's operation by reading only the comments.

Explain the meaning and function of every variable declaration. Every single one. Explain the return value, if any. Long variable names are merely an *aid* to understanding; accompany the descriptive name with a deep, meaningful, prose description.

Explain the parameters during the function definition, as follows:

```
type function_name(type parameter1 /* comment */
      type parameter2 /* comment */)
```

Comment assembly language blocks, and any line that is not crystal clear. The worst comments are those that say "move AX to BX" on a MOV instruction! Reasonable commenting practices will yield about one comment on every other line of assembly code.

In general, do not postfix a comment to a line of C code. That is, the following construct is incorrect:

```
c_code_of_some_sort; // comment
```

Instead, prefix the line with comment line or lines, as follows:

```
/*
comments
*/
```

The reason: Postfixed comments must line up to enhance readability. To do so we generally start the comment on some column very far to the right, reducing the amount of space available for the comment … so we spend a lot of time removing words from the comment to make it fit, and reducing the information content. Making changes is harder, as change in the source results in the need to realign the comment.

Exceptions to this rule are things like commenting the definition of a structure element or the meaning of a case in a switch statement.

This rule does not apply to assembly language, where each statement does so little a short comment is appropriate.

Acronyms and abbreviations are not allowed unless defined in the Abbreviation Table stored in the Version file (see more about this in Section A.4.1). While "DSP" might mean "Display" to you, it means "Digital Signal Processor" to me. Clarity is our goal!

Though it's useful to highlight sections of comments with strings of asterisks, never have characters on the left or right side of a block of comments. It's too much trouble to maintain proper spacing as the comments later change. In other words, this is not allowed:

```
/**********************************************
* This comment incorrectly uses right-hand*
* asterisks *
**********************************************/
```

The correct form is:

```
/**********************************************
This comment does not use right-hand
asterisks
***********************************************/
```

## A.8 Coding Conventions

### *A.8.1 General*

No line may ever be more than 80 characters.

Don't use absolute path names when including header files. Use the form `#include <module/name>` to get public header files from a standard place.

Never, ever use "magic numbers." Instead, first understand where the number comes from, then define it in a constant, and then document your understanding of the number in the constant's declaration.

### *A.8.2 Spacing and Indentation*

Put a space after every keyword, unless a semicolon is the next character, but never between function names and the argument list.

Put a space after each comma in argument lists and after the semicolons separating expressions in a `for` statement.

Put a space before and after every binary operator (like +, −, etc.). Never put a space between a unary operator and its operand (e.g., unary minus).

Put a space before and after pointer variants (star, ampersand) in declarations. Precede pointer variants with a space, but have no following space, in ***.

Indent C code in increments of two spaces. That is, every indent level is two, four, six, etc. spaces.

Always place the # in a preprocessor directive in column 1.

### A.8.3 C Formatting

Never nest `if` statements more than three deep; deep nesting quickly becomes incomprehensible. It's better to call a function, or even better to replace complex `ifs` with a SWITCH statement.

Place braces so the opening brace is the last thing on the line, and place the closing brace first, like:

```
if (result > ; a_to_d) {
   do a bunch of stuff
}
```

Note that the closing brace is on a line of its own, except when it is followed by a continuation of the same statement, such as:

```
do {
 body of the loop
} while (condition);
```

When an `if-else` statement is nested in another `if` statement, always put braces around the `if-else` to make the scope of the first `if` clear.

When splitting a line of code, indent the second line like this:

```
function (float arg1, int arg2, long arg3,
      int arg4)
```

or,

```
if (long_variable_name && constant_of_some_sort == 2
    && another_condition)
```

Use too many parentheses. Never let the compiler resolve precedence; explicitly declare precedence via parentheses.

Never make assignments inside `if` statements. For example don't write:

```
if ((foo= (char *) malloc (sizeof *foo)) == 0)
  fatal ("virtual memory exhausted");
```

instead, write:

```
foo= (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal ("virtual memory exhausted")
```

If you use `#ifdef` to select among a set of configuration options, add a final `#else` clause containing a `#error` directive so that the compiler will generate an error message if none of the options has been defined:

```
#ifdef sun
#define USE_MOTIF
#elif hpux
#define USE_OPENLOOK
#else
#error unknown machine type
#endif
```

### A.8.4 Assembly Formatting

Tab stops in assembly language are as follows:

- Tab 1: column 8
- Tab 2: column 16
- Tab 3: column 32

Note that these are all in increments of 8, for editors that don't support explicit tab settings. A large gap—16 columns—is between the operands and the comments.

Place labels on lines by themselves, like this:

```
label:
      mov   r1, r2      ;r1=pointer to I/O
```

Precede and follow comment blocks with semicolon lines:

```
;
; Comment block that shows how comments stand
; out from the code when preceded and followed by
; "blank" lines.
;
```

Never run a comment between lines of code. For example, do not write like this:

```
mov   r1, r2       ;Now we set r1 to the value
add   r3, [data]   ;we read back in read_ad
```

Instead, use either a comment block, or a line without an instruction, like this:

```
mov   r1, r2       ;Now we set r1 to the value
                   ;we read back in read_ad
add   r3, [data]
```

Be wary of macros. Though useful, macros can quickly obfuscate meaning. Do pick very meaningful names for macros.

APPENDIX B

# A Simple Drawing System

Just as firmware standards give a consistent framework for creating and managing code, a drawing system organizes hardware documentation. Most middle- to large-sized firms have some sort of drawing system in place; smaller companies, though, need the same sort of management tool.

Use the following standard intact or modified to suit your requirements. Feel free to download the machine readable version from www.ganssle.com/ades/dwg.html.

## B.1 Scope

This document describes a system that:

- Guarantees everyone has, and uses, accurate engineering documents
- Manages storage of such documents and computer files so to make their backup easy and regular
- Manages the current configuration of each product

The system outlined is primarily a method to describe exactly what goes into each product through a system of drawings. A top level Configuration Drawing points to lower level drawings, each of which points to specific parts and/or even lower level drawings. After following the "pointer chain" all the way down to the lowest level, one will have access to:

- Complete assembly drawings including mod lists
- A complete parts list
- By reference, other engineering documents like schematics and source files

The system works through a network of Bills of Materials (BOMs), each of which includes the pointers to other drawings or the part numbers of bit pieces to buy and build.

Our primary goal is to build and sell products, so the drawing system is tailored to give production all of the information needed to manufacture the latest version of a product. However, keeping in mind that we must maintain an auditable trail of engineering support information, the system always contains a way to access the latest such information.

## B.2 Drawings and Drawing Storage

### *B.2.1 Definitions*

The term "drawing" includes any sort of documentation required to assemble and maintain the products. Drawings can include schematics, BOMs, assembly drawings, PAL and code source files, etc.

A "Part" is anything used to build a product. Parts include bit pieces like PC boards and chips, and may even include programmed PALs and ROMs. A part may be described on a drawing by a part number (like 74HCT74), or by a drawing number (in the case of something we build or contract to build).

### *B.2.2 Drawing Notes*

Every drawing has a drawing number associated with it. This number is organized by product series, as follows:

| | |
|---|---|
| Company Documentation | #0001 to #0499 |
| Configuration drawings: | #0500 to #0999 |
| Product line "A": | #1000 to #1999 |
| Product line "B": | #2000 to #2999 |
| Product line "C": | #3000 to #3999 |

Every drawing has a revision letter associated with it, and marked clearly upon it. Revision letters start with the letter "A" and proceed to "Z." If there are more than 26 revisions, after "Z" comes "AA," then "AB," etc.

The first release of any drawing is to be marked revision "A." There are to be no drawings with no revision letters.

Every drawing will have the date of the revision clearly marked upon it, with the engineer's initials or name.

Every drawing will have a master printed out and stored in the MASTERs file. The engineer releasing the drawing or the revision will stamp the Master with a red MASTER stamp and will fill in a date field on that stamp.

Though in many cases both electronic and paper copies of drawings (such as for a schematic) exist, the paper copy is always considered the MASTER.

Drawing numbers are always four-digit numerics, prefixed by the "#" character.

### B.2.3 Storage

All Master drawings and related documentation will be stored in the central repository. Master computer files will be stored on network drive in a directory (described later).

Everyone will have access to Master drawings and files. These are to be used as a reference only; no one may take a Master drawing from the central repository for any purpose except for the following:

Drawings may be removed to be photocopied. They must be returned immediately (within 30 min) to the central repository.

Drawings may be removed by an engineer for the sole reason of updating them, to incorporate engineering change orders (ECOs) or otherwise improve their accuracy. However, drawings may be removed only if they will be immediately updated; you may not pull a Master and "forget" about it for a few days. It is anticipated that, since most of our drawings are generated electronically, a Master will usually just be removed and replaced by a new version. See "Obsolete drawings" for rules regarding the disposition of obsoleted drawings.

Artwork may be removed to be sent out for manufacturing. However, all POs sent to PC vendors must require "return of artwork and all films." He who pulls the artwork or film is responsible to see that the PO has this information. Returned art must be immediately refiled.

All drawings will be stored in file folders in a “Master Drawing” file cabinet. Those that are too big to store (like D size drawings) will be folded. Drawings will be filed numerically by drawing number.

Artwork will be stored in a flat file, stored within protective paper envelopes. Every piece of artwork and film will have a drawing number and revision marked both on the art/film and on the envelope. If it is not convenient to make the art marking electronically, then use a magic marker.

### *B.2.4 Storage: Obsoleted Drawings*

Every Master drawing that is obsoleted will be removed from the current Master file and moved to an Obsolete file. Obsoleted drawings will be filed numerically by drawing number. Where a drawing has been obsoleted more than once, each old version will be substored by version letter.

The Master will be stamped with a red OBSOLETE stamp. Enter the date the drawing is cancelled next to the stamp. Thus, every Obsolete drawing will have two red stamps: MASTER (with the original release date) and OBSOLETE (with the cancellation date).

If old ECOs are associated with the Obsoleted drawing, be sure they remain attached to it when it is moved to the Obsolete file.

Obsoleted Artwork and films will be immediately destroyed.

Sometimes one makes a small modification to a Master drawing to incorporate an ECO—say, if a hand-drawn PC board assembly drawing changes slightly. In this case duplicate the master before making the change, stamp the duplicate OBSOLETE, and file the duplicate.

The reason for saving old drawings is to preserve historical information that might be needed to update/fix an old unit.

## B.3 Master Drawing Book

Whenever a drawing is released or updated, the Master Drawing Book will be modified by the releasing engineer to reflect the new information.

The Master Drawing Book is a looseleaf binder stored and kept with the Master drawing file. The Master Drawing Book lists every drawing we have by number and its current revision level. In addition, if one or more ECOs are current against a drawing, it will be listed along with a brief one-line description of what the ECO is for.

Just as important, the Master Drawing Book lists the name of the electronic version of a drawing. This name is always the name of the file(s) on the network drive, with the associated directory path listed.

Note that the “Dash Number” (described later in the Section B.5) is not included in the list, since one drawing might have many dash numbers.

Thus, the drawing list looks like:

| Dwg # | Revision | Rev date | Title | Filename |
|---|---|---|---|---|
| #1000 | A | 8-1-97 | Prod A BOM | PRODA-ASSY |
|  | ECO: PRODA.A.3 |  | Stabilize clock | PRODA\ECO.A |
|  | ECO: PRODA.A.1 |  | Secure cables | PRODA\ECO.A |
| #1001 |  | 8-2-97 | Prod A Baseplate | PRODA-BASE |

As drawings are updated the ECOs will no longer apply and should then be removed from the book.

Note that after each BOM drawing number there is a list of dash numbers that describe what each configuration of the drawing is.

A section at the rear of the book will contain descriptions of “Specials”—units we do something weird to make a customer happy. If we give someone a special PAL, document it with the source code and notes about the unit’s serial number, date, etc. A copy of this goes in the unit’s folder. It is the responsibility of the technician to insure that the folder and Master Drawing Book is updated with “special” information.

The Master Drawing Book master copy will be stored as file name ENGINEER\DOCS\MDB.DOC, and is maintained in Word.

## B.4 Configuration Drawings

Every product will have a Configuration Drawing associated with it. These Drawings essentially identify what goes into the shipping box.

Currently, the following Configuration Drawings should be supported:

| Dwg # | | Description |
|---|---|---|
| #0501 | | Product A |
| | -1 | 256 k RAM option |
| | -2 | 1 Mb RAM option |
| | -3 | 50 MHz option |
| #0502 | | Product B |
| #0503 | | Product C |
| | -1 | 256 k RAM option |
| | -2 | 1 Mb RAM option |
| | -3 | 50 MHz option |

The "dash numbers" are callouts to BOMs for variations on a standard theme.

The Configuration Drawing is a BOM (see Section B.5). As such, it calls out everything shipped to the customer. Items to be included in the Configuration Drawing include:

- The unit itself (perhaps with dash numbers as above)
- Manual (with version number)
- Software disk
- Paper warranty notice
- FCC Notice

Thus, starting with the Configuration Drawing, anyone can follow the "pointer trail" of BOMs and parts/drawings to figure out how to buy everything we need to make a unit, and then how to put it together.

## B.5 Bills of Materials

A BOM lists *every* part needed for a subassembly.

The Drawing System really has only three sorts of drawings: BOMs, drawings for piece parts, and other engineering documentation. A piece part drawing is just like a part: it is something we build or buy and incorporate into a subassembly. As such, every piece part drawing is called out on a BOM, as is every piece part we purchase (like a 74HCT74). The part number of a piece part made from a drawing is just the drawing number itself. So, if drawing number #1122 shows how to mill the product's baseplate, calling out part number #1122 refers to this part.

"Other engineering documentation" refers to schematics, test procedures, modification drawings, ROM/PAL drawings, and assembly drawings (pictorial representations of how to put a unit together). None of these calls out parts to buy, and therefore is always referenced on any BOM with a quantity of 0.

A piece part drawing can never refer to other parts; it is just one "thingy." A BOM always refers to other parts, and is therefore a collection of parts.

One BOM might call out another BOM. For example, the product A top level BOM might call out parts (like the unit's box), drawings (like the baseplate), and a number of other BOMs (one per circuit board). In other words, one BOM can call out another as a part (i.e., a subassembly).

Though all BOMs have conventional four-digit drawing numbers, everything that refers to a BOM does so by appending a "dash number." That is, BOM number #1234 is never called out on some higher level drawing as "#1234"; rather, it would be either of "#1234-1," or "#1234-2," etc.

The dash number has two functions. First, it identifies the called-out item as yet another subassembly. Any time you see a number with the dash number like this, you know that item is a subassembly.

The second reason is more important. The dash numbers let one drawing refer to several variations on a design. For example, if the BOM for the "Option A Memory Board" is drawing number #1000, then #1000-1 might refer to 128 K RAM and #1000-2 to 1 MB RAM. The design is the same, so we might as well use the same drawings. The configuration is just a little different; one drawing can easily call out both configurations.

A good way to view the drawing system is as a matrix of pointers. The Top Level Configuration Drawing (which is really a BOM) calls out subassemblies by referring to each with a drawing number with a dash suffix—a sort of pointer. Each subassembly contains pointers to parts or more levels of indirection to further BOMs. This makes it easy to share drawings between projects; you just have to monkey with the pointers. The dash numbers insure that every configuration of a project is documented, not just the overall PC layout.

### B.5.1 BOM Format

BOMs are never “pictures” of anything—they are always just BOMs (i.e., parts lists). The parts list includes every part needed to build that subassembly. Some of the parts might refer to further subassemblies.

The parts list of the BOM has the following fields:

- Item number (starting at 1 and working up)
- Quantity used, by dash number
- Part (or drawing) number
- Description
- Reference (i.e., U number or whatever)

Here is an example of a BOM numbered #1000, with three dash number options. This is a portion of a memory option board BOM with several different memory configurations:

| Item | Qty | | | Part # | Description | Ref |
|---|---|---|---|---|---|---|
| | -1 | -2 | -3 | | | |
| 1 | #1000-1 | | | | OPTION board 256 k | |
| 2 | | #1000-2 | | | OPTION board 1 Mb | |
| 3 | | | #1000-3 | | OPTION board 4 Mb | |
| 4 | | | | #1892 | OPTION ass’y | |
| 5 | | | | #1234 | OPTION schematic | |
| 6 | | | | #1111 | Test Procedure | |
| 7 | 1 | 1 | 1 | #1221 | OPTION PCB | |
| 8 | 8 | 8 | 8 | Ap1123 | 32 pin socket | U1-8 |
| 9 | 1 | 1 | 1 | 74F373 | IC | U10 |
| 10 | 8 | | | 62256 | Static RAM | U1-8 |
| 11 | | 8 | | 621128 | Static RAM | U1-8 |
| 12 | | | 2 | 624000 | Static RAM | U1-2 |
| 13 | 2 | | | APC3322 | Jumper | J1,2 |
| 14 | | 2 | 2 | APC3322 | Jumper | J3,4 |

First, note that each of the three BOM types (i.e., dash numbers) is listed at the beginning of the parts list. A column is assigned to each dash number; the quantities needed for a particular dash number are in this column. That is, there is a "quantity" column for each BOM type.

The first three entries, one per dash number, simply itemize what each dash number is. The quantity must be zero.

Each dash number column contains all quantity information to make that particular variation of the BOM.

Next, notice that drawing "#1892" is called out with a quantity of 0. Drawing #1892 shows how the parts are stuffed into the board and is essential to production. However, it cannot call parts that must be bought, so always has a quantity of 0.

The schematic and test procedure are listed, even though these are not really needed to build the unit. This is how all non-production engineering documents are linked into the system. All schematics, test procedures, and other engineering documentation that we want to preserve should be listed, but the quantity column should show 0. Notice also that a drawing number is assigned even to the test procedure. This insures that the test procedure is linked into the system and maintained properly.

The first column is the "item number." One number is assigned to each part, starting from 1 and working up. This is used where a mechanical drawing points out an item; in this case the item number would be in a circle, with an arrow pointing to the part on the drawing. It forms a cross reference between the pictorial stuffing drawing and the parts list. In most cases most item numbers will not have a corresponding circle on the drawing.

All jumpers that are inserted in the board are listed along with how they should be inserted (by the reference designator). This is the only documentation about board jumpering we need to generate.

Note that no modifications to the PCBs are listed. PC board modifications are to be listed on a separate "Mod" drawing, which is also referenced with a quantity of zero on the BOM.

## B.6 ROMs and PALs

Every ROM and PAL used in a unit will be called out by two entries in the parts list columns of the PC board BOM. The first entry calls out the device part number (like GAL22V10) and associated data so purchasing can buy the part. The second entry, which must follow right after the first, calls out a ROM or PAL BOM.

The ROM or PAL BOM will be called out with quantity of 0. This procedure really violates the definition of the drawing system, but it drastically reduces the number of drawings needed by production to build a unit.

On the PC board BOM, the callout for a ROM or PAL will look like:

| Item | Qty | Part # | Description | Ref |
|---|---|---|---|---|
| 1 | 1 | GAL22V10 | PAL | U19 |
| 2 | 0 | #1234-1 | (MASTERS\PRODA\M-U19. PDS) | B9 |

Thus, the first entry tells us what to buy and where to put it; the second refers to engineering documentation and the current checksum. For a ROM, list the version number instead of the checksum. The description field for the part must also include the ROM or PAL's file name in parentheses, with directory on the Lab computer.

ROMs, PALs, and SLD will be defined via BOMs, since these elements are really composed of potentially numerous sets of documentation. The ROM/PAL/SLD drawing will form the basic linkage to all source code files used in their creation.

The primary component of a PAL/ROM drawing is of course the device itself. Other rows will list the files needed to build the ROM or PAL.

Where two ROMs are derived from one set of code (like EVEN and ODD ROMs), these will both be on the same drawing.

An example ROM follows:

| Item | Qty | Part # | Description | Ref |
|---|---|---|---|---|
| | -1 | | | |
| | 1 | 1234-1 | 64180 P-bd ROM | U9 |
| 1 | | 27256-10 | EPROM, 100 ns | |
| 2 | | | PRODA.MAK—make file | proda\code |

Note that in this part list the EPROM itself is called out by conventional part number, but the quantity is 0 (since a quantity was called out on the PC board BOM that referenced this drawing).

A ROM, PAL, or SLD drawing calls out the ingredients of the device. In this case, the software's MAKE is listed so there's a reference from the hardware design to the firmware configuration.

If other engineering documentation exists, it should be referred to as well. This could include code descriptions, etc.

The last column contains the directory where these things are stored on the network drive.

The goal of including all of this information is to form one repository which includes pointers to all important parts of the component.

## B.7 ROM and PAL File Names

All PALs and ROMs will have file names defined by the conventions outlined here.

PALs are named: <board>-U<U number>.J<checksum>
ROMs are named: <board>-U<U number>.V<version>

Thus, you can tell a ROM from a PAL from the extension, whose first character is a V for a ROM or a J for a PAL.

Legal <board> names are (limited to one character):

M—main board
P—option A board
T—option B board

Examples:

M-U10.JAB—main board, U10, checksum=AB
M-U1.J12—main board, U1, checksum=12

## B.8 Engineering Change Orders

ECOs will be issued as required, in a timely fashion to insure all manufacturing and engineering needs are satisfied.

Every ECO is assigned against a drawing, not against a problem. You may have to issue several ECOs for one problem, if the change affects more than one drawing.

The reason for issuing perhaps several ECOs (one per drawing) is twofold: first, production builds units from drawings. They should not have to cross reference to find how to handle drawings. Secondly, engineering modifies drawings one at a time. All of the information needed to fix a drawing must be associated with the drawing in one place.

Each ECO will be attached to the affected drawing with a paper clip. The ECO stays attached only as long as the drawing remains incorrect. Thus, if you immediately fix the Master (say, change the PAL checksum on the drawing), then the ECO will be attached to the newly Obsoleted Master, and filed in the Obsolete file.

If the ECO is not immediately incorporated into, say, a schematic, then the person issuing the ECO will pencil the change onto the MASTER drawing, so the schematic always reflects the way the unit is currently built.

In addition, if the ECO is not immediately incorporated into the drawing, the engineer issuing the ECO will mark the Master Drawing Book with the ECO and a brief description of the reason for the ECO, as follows:

| Dwg # | Title | Revision | Rev Date | Filename |
|---|---|---|---|---|
| #3000 | Prod A BOM | A | 8-1-97 | PRODA-ASSY |
| | ECO: PRODA.A.3 | | Stabilize clk | PRODA.A.3 |
| | ECO: PRODA.A.1 | | Secure cables | PRODA.A.1 |

Note that the filename of the ECO is included in the Master Drawing Book.

When the ECO is incorporated into the drawing, remove the ECO annotation from the Master Drawing Book, as it is no longer applicable.

NEVER change a drawing without looking in the Master repository to see if other ECOs are outstanding against the drawing.

Every change gets an ECO, even if the change is immediately incorporated into a drawing. In this case, follow the procedure for obsoleting a drawing. This provides a paper audit trail of changes, so we can see why a change was made and what the change was.

Every ECO will result in incrementing the version numbers of all affected drawings. This includes the Configuration Drawing as well. To keep things simple, you do not have to issue an ECO to increment the Configuration version number. We do want this incremented, though, so we can track revision levels of the products. Add a line to the Master Drawing Book listing the reason for the change and the new revision level of the Configuration, as well as a list of affected drawings. This forms back pointers to old drawings and versions. Though we remove old ECO history from our drawings, never remove it from the Configuration Drawing's Master Drawing Book entry, as this will show the product's history.

The Master Drawing Book entry for an ECO'd Configuration Drawing will look like:

| Dwg # | Revision | Rev date | Title | Filename |
|---|---|---|---|---|
| #0600 | A | 8-1-97 | Prod A Configuration | PRODA-ASSY |
| | B | 8-2-97 | Mod clock circuit to be more stable (1000-1, 1234 modified) | |
| | C | 8-3-97 | Secure cables better | |

Sometimes a proposed ECO may not be acceptable to production. For example, a proposed mod may be better routed to different chip pins. Therefore, the engineer making an ECO must consult with production before releasing the ECO. (This avoids a formal (and slow) system of controlled ECO circulation.)

A decision must be made as to how critical the ECO is to production. The engineer issuing the ECO is authorized to shut down production, if necessary, to have the ECO incorporated in units currently being built.

Thus, to issue an ECO:

- Fill out the ECO form, one per drawing, and distribute it to production and all affected engineers.
- If you don't immediately fix the drawing, clip it to the affected drawing and mark the Master Drawing Book as described.
- If necessary, pencil the changes onto the Master drawing.
- Increment the Configuration Drawing version number immediately. Add a line to the Master Drawing Book after the Configuration Drawing entry describing the reason for the change and listing the affected drawings.
- If the change is a mod, consult with production on the proposed routing of the mod.
- If the change is critical, instruct production to incorporate it into current work-in-progress.
- Remember that most likely several drawings will be affected: a new mod will affect the schematic and the BOM that shows the mod list.

To incorporate an ECO into a drawing:

- Make whatever changes are needed to incorporate ALL ECOs clipped to that drawing.
- Revise the version letter upwards.
- Generate a new Master drawing, and Obsolete the old Master.
- Delete the ECO file from the network drive.
- Revise the Version letter on the Configuration Drawing.

## B.9 Responsibilities

The engineer making a change is responsible to insure that change is propagated into the drawing system and that the information is disseminated to all parties. He/she is responsible for filing the drawings, removing and refiling obsoleted drawings, stamping MASTER or OBSOLETE, etc.

The engineer making the change must update production's master ROM/PAL computer with current programming files and the drawings with checksums and versions as appropriate. The engineer must immediately also update the network drive and pass out ECOs.

Nothing in this precludes the use of clerical staff to help. However, final responsibility for correctness lies with the engineer making changes.

The Master Drawing Book does contains information about "Specials" we've produced. The manufacturing technician is responsible to insure that all appropriate information is saved both in this Book and in the unit's folder.

The production lab MUST maintain an accurate, neat book of CURRENT BOMs, to insure the units are built properly. Every change will result in an ECO; the lab must file that promptly.

APPENDIX C

# A Boss's Guide to Process Improvement

I hear from plenty of readers that their bosses just don't "get" software. Efforts to institute even limited methods to produce better code are thwarted by well-meaning but uninformed managers chanting the "can't you just write more code?" mantra.

Yet when I talk to the bosses many admit they simply don't know the rules of the game. Software engineering isn't like building widgets or designing circuit boards. The disciplines are quite different, techniques and tools vary, and the people themselves all too often quirky and resistant to standard management ploys. Most haven't the time or patience to study dry tomes or keep up with the standard journals. So here's my short intro to the subject. Give it to your boss.

Dear boss: The first message is one you already know. Firmware is the most expensive thing in the universe. Building embedded code will burn through your engineering budget at a rate matched only by a young gold digger enjoying her barely sentient ancient billionaire's fortune.

Most commercial firmware costs around $20–40 per line, measured from the start of a project till it's shipped. When developers tell you they can "code that puppy over the weekend" be very afraid. When they estimate $5/line, they're on drugs or not thinking clearly. Defense work with its attendant reams of documentation might run upwards of $100 per line or more; the space shuttle code is closer to $1000 per line, but is without a doubt the best code ever written.

For even a tiny 5 K line application, $20–40 per line translates into a six-figure budget. The moral: embarking on any development endeavor without a clear strategy is a sure path to squandering vast sums.

Like the company that asked me to evaluate a project that was 5 years late and looked more hopeless every day. I recommended they trash the $40 million effort and start over, which they did. Or the startup which, despite my best efforts to convince them otherwise, believed the consultants' insanely optimistic schedule. They're now out of business—the startup, that is. The consultants are thriving.

## C.1 Version Control

First, before even thinking about building any sort of software, install and have your people use a version control system (VCS). Building even the smallest project without a VCS is a waste of time and an exercise in futility.

The NEAR spacecraft dumped a great deal of its fuel and was nearly lost when an accelerometer transient caused the on-board firmware to execute abort code … incorrect abort code, software that had never really been tested. Two versions of the 1.11 flight software existed; unhappily, the wrong set flew. The code was maintained on uncontrolled servers. Anyone could, and did, change the software. Without adequate version control, it wasn't clear what made up correct shipping software.

A properly deployed VCS insures these sorts of dumb mistakes just don't happen. The VCS is a sort of database for software, releasing the code to users but tracking who changed what when. Why did the latest set of changes break working code? The VCS will report what changed, who did it, and when, giving the team a chance to efficiently troubleshoot things.

Maybe you're shipping release 2.34, but one user desperately requires the old 2.1 software. Perhaps a bug snuck in sometime in the last 10 versions and you need to know which code is safe. A VCS reconstructs any version at any time.

Have you ever misplaced code? In October of 1999 the FAA announced they had lost the source code to all of the software that controlled air traffic between Chicago and the regional airports. The code all lived on one developer's machine, one angry person who quit and deleted it all. He did, however, install it on his home computer, encrypted. The FBI spent 6 months reverse engineering the encryption key to get their code back. Sound like disciplined software development? Maybe not.

Without a VCS, a failure of any engineer's computer will mean you lose code, since it's all inevitably scattered around amongst the development team. Theft or a fire—unhappily everyday occurrences in the real world—might bankrupt you. The computers have little value, but that source code is worth millions.

The version control database—the central repository of all of your valuable software—lives on a single server. Daily backups of that machine, stored offsite, insure your business's survival despite almost any calamity.

Some developers complain that the VCS won't protect them from lazy programmers who cheat the system. You or your team lead should audit the VCS's logs occasionally to be sure developers aren't checking out modules and leaving them on their own computers. A report that takes just seconds to produce will tell you who hasn't checked in code, and how long it has been out on their own computers.

VCS range in price from free (like the GNU products) to expensive, but even the expensive ones are cheap. See http://better-scm.berlios.de/comparison/comparison.html for a comprehensive list of products.

## C.2 Firmware Standards

What language is spoken in America? English, of course, but try talking to random strangers on a street corner in any city today. The dialects range from educated middle-American to incomprehensible near-gibberish. It's all English, of a sort, but it sounds more like the fallout from the Tower of Babel.

In the firmware world we speak a common language: C, C++, or assembly, usually. Yet there's no common dialect; developers exploit different aspects of the lingos, or construct their programs using legal but confusing constructs.

The purpose of software is to work, of course, but also to clearly communicate the programmer's intentions to maintenance people. Clear communication means we must all use similar dialects. Someone—that's you, boss—must specify the dialect.

The C and C++ languages are so conducive to abuse that there's a yearly obfuscated C contest whose goal is to produce utterly obscure but working code. Figure C.1 is an excerpt from one winning entry; this is real, working, but utterly incomprehensible code. Everyone wants the URL to see other bizarre entries, but forget it! Vow that your group will produce world-class software that's cheap to maintain.

```
                       O p                                          ,B,
                    D,A=6,Z                                       ,S=0,v=
                 0,n=0,W=400                                      ,H=300,a[7]
               ={ 33,99, 165,                                    231,297,363} ;
             XGCValues G={ 6,0                                  ,~0L,0,1} ; short
            T[]={ 0,300,-20,0,4                                ,-20,4,10,4,-5,4,5,
          4,-20,4,20,4,-5,4,5,4,                              -10,4,20},b[]={ 0,0,4,
        0,-4,4,-4,-4,4,-4,-4,4} ;                            C L[222],I[222];dc(O x){
       M(T,a[x],H,12); } Ne(C 1,o                           s) { l.f=l.a=1; l.b=l.u=s;
       1.t=16; 1.e=0; U;} nL(O t,O                         a,O b,O x,O y,O s,O p){ C l;
      1.d=0; 1.f=s; 1.t=t; y-=1.c=b;                      l.e=t==2?x;p; x-=l.s=a;s=(x|1)
     %2*x; t=(y|1)%2*y; 1.u=(a=s>t?s:                    t)>>9;l.a=(x<<9)/a;l.b=(y<<9)/a;
    U; } di(C I) {O p,q,r,s,i=222;C 1;                  B=D=0; R i--){ l=L[i]; Y>7){ p=I.s
   -1.s>>9; q=I.c-1.c>>9; r=1.t==8?1.b:                l.a; s=p*p+q*q; if(s<r*r||I.t==2&&s<
   26) F S+=10; s=(20<<9)/(s|1); B+=p*s;              D+=q*s; }} F 0; } hi(O x,O d){ O i=A;
  R i--&&(x<a[i]-d||x>a[i]+d)); F i;}      dL(){ O      c,r=0, i=222,h; C l; R i--){ l=L[i];
 Y){ r++;c=1.f; Y==3){c=1.u; 1.t=0;     E; }R c--){--     l.u;h=l.c>>9; Y>7){XDrawArc(d,w,g,
 (1.s>>9)-++1.a,h-1.a,1.a*2,1.a*2,0  ,90<<8); if(!1.u){    I[i].t-=8; l=I[i]; } } else Y==2)M
 (b,1.s>>9,h,6); else XDrawPoint(d   ,w,g,(1.s+=1.a)>>9,    h=(l.c+=l.b)>>9); Y==4&&!l.u){ Ne
 (1,20); K; } Y&&1.t<3&&(di(1)||h>   H)){ if(h>H&&(c=hi(    l.s>>9,25))>=0){ dC(c); a[c]=a[--
A]; }Ne(1,30); Y==1){ E;K; } else    c=1.t=0;} Y==1&&h<H    -75&&!N(p*77)){ do{ nL(1,l.s,l.c,
                                      N(W<<9),H<<9,1,i+
                                        1); I[i].d++;
                                           }R N(3)

                                        );         K;
                                       l.u=c; c=0; } Y
                                      ==2){ l.s+=l.a+B;
                                     l.a= (l.e-l.s)/((H+
                                    20-h)|1); l.c+=l.b+D;
                                   M(b,l.s>>9,l.c>>9,6); }
                                  } L[i]=1; } } F r; } J(){
                                 R A) { XFlush(d); v&&sleep(
                                3); Z=++v*10; p=50-v; v%2&&hi
                               ((a[A]=N(W-50)+25),50)<0 &&A++;
                              XClearWindow (d,w); for(B=0; B<A;
                             dC(B++)); R Z|dL()){ Z&&!N(p)&&(Z--
                            ,nL(1+!N(p),N(W<<9), 0,N(W<<9),H<<9,1
                           ,0)); usleep(p*200); XCheckMaskEvent(d,
                          4,&e)&&A&&--S&&nL(4,a[N(A)]<<9,H-10<<9,e.
                         xbutton.x<<9,e.xbutton.y<<9,5,0);}S+=A*100;
                             B=sprintf(m,Q,v,S); XDrawString(d,w
                                     ,g,W/3,H/2,m,B); } }
```

**Figure C.1: Real C code ... but what dialect? Who can understand this? (reprinted by permission of the IOCCC)**

The code won't be readable unless we use constructs that don't cause our eyes to trip and stumble over unusual indentation, brace placement, and the like. That means setting rules, a standard, used to guide the creation of all new code.

The standard defines far more than stylistic issues. Deeply nested conditionals, for instance, lead to far more testing permutations than any normal person can manage. So the standard limits nesting. It specifies naming conventions for variables, promoting identifiers that have real meaning. Tired of seeing i, ii, or iii for loop variable names? The standard outlaws such lazy practices. Rules define how to construct useful comments. Comments are an integral and essential part of the source code, every bit as important as for and while loops. Replace or retrain any team member who claims to write "self-commenting code."

Some developers use the excuse that it's too time consuming to produce a standard. Plenty exist on the net, and one is in this book. It contains the brace placement rule that infuriates the most people … so you'll change it and make it your own.

So write or get a firmware standard. And work with your folks to make sure *all* new code follows the standard.

## C.3 Code Inspections

What's the cheapest way to get rid of bugs? Why, just don't put any in!

Trite, perhaps, yet there's more than a grain of wisdom there. Too many developers crank lots of code fast, and then spend ages fixing their mistakes. The average project eats 50% of the schedule in debugging and test! Reduce debugging, by inserting fewer bugs, and accelerate the schedule.

Inspect all new code. That is, use a formal process that puts every function in front of a group of developers before they spend any time debugging. The best inspections use a team of about four people who examine every line of C in detail. They'll find most of the bugs before testing.

Study after study shows inspections are 20 times cheaper at eliminating bugs than debugging. Maybe you're suspicious of the numbers—fine, divide by an order of magnitude. Inspections still shine, cutting debugging in half.

More compellingly it turns out that most debugging strategies never check half the code. Things like deeply nested IF statements and exception handlers are tough to test. My collection of embedded disasters shows a similar disturbing pattern: most stem from poorly executed, pretty much untested error handlers.

Inspections and firmware standards go hand in hand. Neither works without the other. The inspections insure programmers code to the standard, and the standard eliminates inspection-time arguments over stylistic issues. If the code meets the standard, then no debates about software styles are permitted.

Most developers hate inspections. Tough. You'll hear complaints that they take too long. Wrong. Well-paced inspection meetings examine 150 lines of code per hour, a rate that's hardly difficult to maintain (that's 2.5 lines of C per minute), yet that costs the company only a buck or so per line. Assuming, of course, that the inspection has no value at all, which we know is simply not true.

Your role is to grease the skids so the team efficiently cranks out fabulous software. Inspections are a vital part of that process. They won't replace debugging, but will find most of the bugs very cheaply.

I wrote that code inspections are 20 times cheaper than debugging. That's quite a claim! Figure C.2 shows how that at 20×, given that debugging typically consumes half the schedule, using inspections effectively lets you divide the schedule by 1.9.

Don't believe the 20× factor? Divide it by an order of magnitude. Figure C.2 shows even at that pessimistic figure you can divide the schedule by 1.3.

Have your people look into inspections closely. The classic reference is *Software Inspection* by Gilb and Graham (1993, Addison-Wesley, New York), but Karl Wiegers'

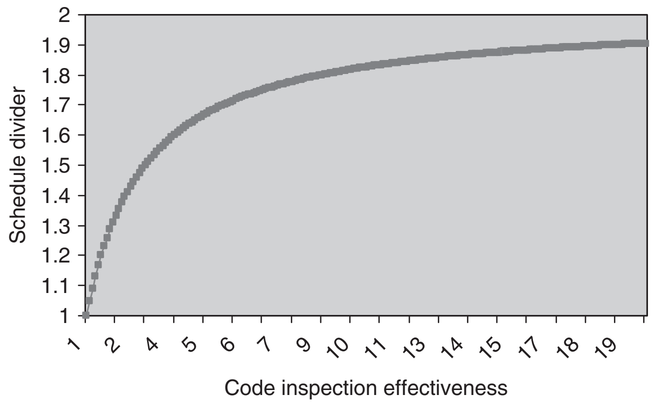


**Figure C.2: Shaving the schedule with code inspections**

newer and much more readable book *Peer Reviews in Software* (2001, Addison-Wesley, New York) targets teams of all sizes (including solo programmers).

## C.4 Chuck Bad Code

Toss out bad code.

A little bit of the software is responsible for most of the debugging headaches. When your developers are afraid to make the smallest change to a module, that's a sure sign it's time to rewrite the offending code.

Developers tend to accept their mistakes, to attempt to beat lousy code into submission. It's a waste of time and energy. Barry Boehm showed in *Software Engineering Economics* that the crummy modules consume four times the development effort of any other module.

Identify bad sections early, before wasting too much time on them, and then recode. Count bug rates using bug tracking software. Histogram the numbers occasionally to find those functions whose error rates scream "fix me!" … and have the team recode.

Figure on tossing out about 5% of the system. Remember that Boehm showed this is much cheaper than trying to fix it.

Don't beat your folks up for the occasional function that's a bloody mess. They may have screwed up, but have learned a lot about what should have been done. Use the experience as a chance to create a killer implementation of the function, now that the issues are clearly understood. Healthy teams use mistakes as learning experiences.

Use bug tracking software, such as the free bugzilla (http://www.bugzilla.org/), or any of dozens of commercial products (nice list at http://www.aptest.com/resources.html).

Even the most disciplined developers sometimes do horrible things in the last few weeks to get the device out the door. Though no one condones these actions, fact is that quick hacks happen in the mad rush to ship. That's life. It's also death for software.

Quick hacks tend to accumulate. Version 1.0 is pretty clean, but the evil inflicted in the last few weeks of the project add to problems induced in 1.1, multiplied by an ever-increasing series of hacks added to every release. Pretty soon the programming team says things like "we can't maintain this junk anymore." Then it's too late to take corrective action.

Acknowledge that some horrible things happened in the shipping mania. But before adding features or fixing bugs in the next release, give the developers time to clean up the mess. Pay back the technical debt they incurred in the previous version's end game. Otherwise these hacks will haunt the system forever, reduce overall productivity as the team struggles with the lousy code in each maintenance cycle, and eventually cause the code to rot to the point of uselessness.

## C.5 Tools

A poll on embedded.com (http://embedded.com/pollArchive/?surveyno = 12900001) suggests 85% of companies won't spend more than $1 K on any but the most essential tools. Considering the $150 K+ loaded cost of a single engineer, it's nuts not to spend a few grand on a tool that offers even a small productivity boost.

Like what? Lint, for one. Lint is a program that examines the source code and identifies suspicious areas. It's like the compiler's syntax checker, but one on steroids. Only Lint is smart enough to watch variable and function usage across multiple files. Compilers can't do that. Aggressive Lint usage picks out many problems before debugging starts, for a fraction of the cost. Lint all source files before doing code inspections.

Gimpel (www.gimpel.com) sells one for $239. Buy it, and insure your engineers use it on all new code. Lint is annoying at first, often initially zeroing in on constructs that are indeed fine. Don't let that quirk turn your people off. Tame it, and then reap great reductions in debugging times.

Debugging eats 50% of most projects' schedules. The average developer has a 5–10% error rate. Anything that trims that even a smidgen saves big bucks.

Make sure the developers aren't cheating their tools. Warning levels on compilers, for instance, should be set to the lowest possible level so all warnings are displayed. And then insist the team writes warning-free code. It's astonishing how we ship firmware that spews warnings when compiled. The compiler, which understands the language's syntax far better than any of your people, is in effect shouting "Look here. Here! This is scary!" How can anyone ignore such a compelling danger sign?

Write warning-free code so that maintenance people in months or decades won't be baffled by the messages. "Is it supposed to do this? Or did I reinstall the compiler incorrectly? Which of these is important?" This means changing the way they write C. Use explicit casting. Use parentheses when there's any doubt. These are all good programming practices anyway, with zero cost in engineering, execution speed, or code size. What's the downside?

Editors, compilers, linkers, and debuggers are essential and non-negotiable tools as it's impossible to do any development without these. Consider others. Complexity analyzers can yield tremendous insight into functions, identifying "bad code" early, before the team wastes their time and spirits trying to beat the cruddy code into submission. See www.chris-lott.org/resources/cmetrics/ for a list of freebies. Bug tracking software helps identify problem areas—see a list of resources at http://www.aptest.com/resources.html.

Most firmware developers are desperate for better debugging tools. Unhappily, the grand old days of in-circuit emulators are over. These tools provided deep insight into the intrinsically hard-to-probe embedded system. Their replacement, the BDM, offers far less capability. Have mercy on your folks and insist the hardware team dedicate a couple of spare parallel output bits just to the software people. They'll use these along with instrumented code for a myriad of debugging tasks, especially for hard-to-measure performance issues.

## C.6 Peopleware

Your developers—not tools, not widgets, not components—are your prime resource. As one wag noted, "My inventory walks out the door each night."

I've recommended several books. Please, though, read *Peopleware* by DeMarco and Lister (1999, Dorset House Publishing, New York). It's a slender volume that you'll plow through in just a couple of enjoyable hours. Pursuing the elusive underpinnings of software productivity, for 10 years the authors conducted a "coding war" between volunteering companies.

The results? Well, at first the data was a scrambled mess. Nothing correlated. Teams that excelled on the projects (by any measure: speed, bug count, matching specs) were neither more highly paid nor more experienced than the losers. Crunching every parameter revealed the answer: developers imprisoned in noisy cubicles, those who had no defense against frequent interruptions, did poorly.

How poorly? The numbers are breathtaking. The best quartile was 300% more productive than the lowest 25%. Yet privacy was the only difference between the groups.

Think about it—would you like 3× faster development?

It takes your developers 15 min, on average, after being interrupted to being totally and productively engaged in the cyberworld of coding. Yet a mere 11 min passes between interruptions for the average developer. Ever wonder why firmware costs so much? Email, the phone, people looking for coffee filters all clamor for attention.

Sadly, most developers live in cubicles today, which are, as Dilbert so astutely noted, "anti-productivity pods." Next time you hire someone peer into his cube occasionally. At first he's anxious to work hard, focus, and crank out a great product. He'll try to tune out the poor sod in the next cube who's jabbering on the phone with his lawyer about the divorce. But we're all human; after a week or so he's leaning back from the keyboard, ears raised to get the latest developments. Is that a productive environment?

I advise you to put your developers in private offices, with doors and off-switches on the phones. Every time I've fought this battle with management I've lost, usually because the interior designers promise cubes offer more "flexibility." But even cubicles have options.

Encourage your people to identify their most productive hours, that time of day when their brains are engaged and working at max efficiency. Me, I'm a morning person. Others have different habits. But find those productive hours and help them shield themselves from interruptions for about 3 h a day. In that short time, with the 3× productivity boost, they'll get an entire day's work done. The other 5 h can be used for meetings, email, phone contacts, supporting other projects, etc.

Give your folks a curtain to pull across the cube's opening. Obviously a curtain rod would decapitate employees, generally a bad idea despite the legions of unemployed engineers clamoring for work. Use a Velcro strip to secure the curtain in place. Put a sign on the curtain labeled "enter and die"; the sign and curtain go up during the employee's three superprogramming hours per day. Train the team to respect their colleagues' privacy during these quiet hours. At first they'll be frantic: "But I've GOT to know the input parameters to this function or I'm stuck!" With time they'll learn when Joe, Mary, or Bob will be busy and plan ahead. Similarly, if *you* really need a project update and Shirley has her curtain up, back slowly and quietly away. Wait till the hours of silence are over.

Have them turn off their phone during this time. If Mary's spouse needs her to pick up milk on the way home, well, that's perfect voicemail fodder. If the kids are in the hospital, then the phone attendant can break in on her quiet time.

The study took place before email was common. You know, that little bleep that alerts you to the same tired old joke that's been circulating around the Net for the last three months while diverting attention from the problem at hand. Every few seconds, it seems. Tell your people to disable email while cloistered.

When I talk to developers about the interruption curse they complain that the boss is the worst offender. Resist the temptation to interrupt. Remember just how productive that person is at the moment, and wait till the curtain comes down.

(If you're afraid the employee is hiding behind the curtain surfing the Net or playing Doom, well, there are far more severe problems than just productivity issues. Without trust—mutual trust—any engineering department is in trouble.)

## C.7 Other Tidbits

Where should you use your best people? It's natural to put the superprogrammers on the biggest and most complex projects. Resist that urge—it's wrong.

Capers Jones showed that the best people excel on small (1 man-month) projects, typically being six times more productive than the worst members of the team. That advantage diminishes as the system grows. On an 8 man-month effort the ratio shrinks to under 3 to 1. At 64 man-months it's about 1.5 to 1, and much beyond that the best do as badly as the worst. Or the worst as well as the best. Whatever.

That observation tells us something important about how we partition big projects. Find ways to break big systems down into many small, mostly independent parts. Or at least strip out as much as possible from the huge carcass of code you're planning to generate, putting the removed sections into their own tasks or even separate processors. Give these smaller sections to the superprogrammers. They'll crank out solutions fast.

An example: suppose an I/O device, say an optical encoder, is tied to your system. Remove it. Add a CPU, a cheap PIC, ATMEL, Z8, or similar sub-$1 part, just to manage that one device. Have it return its data in engineering units: "the shaft angle is 27°." Even a slowly rotating encoder would generate thousands of interrupts a second, a burden to even the fastest CPU that's also tasked with many other activities. Yet even a tiny microcontroller can easily handle the data if there's nothing else going on. One smart developer can crank out perfect I/O code in little time.

(An important rule of thumb states that 90% loaded systems double development time, compared to one of 70% or less; 95% loading triples development time.)

While cleverly partitioning the project for the sake of accelerating the development schedule, think like the customer does, not as the firmware folks do. The customer only sees features; never objects, ISRs, or functions. Features are what sell the product.

That means break the development effort down into feature chunks. The first feature of all, of course, is a simple skeleton that sets up the peripherals and gets to main(). That and a few critical ISRs, perhaps an RTOS and the like, form the backbone upon which everything else is built.

Beyond the backbone are the things the customer will see. In a digital camera there's a handler for the CCD, an LCD subsystem, some sort of Flash filesystem. Cool tricks like image enhancement, digital zoom, and much more will be the sizzle that excites marketing. None of those, of course, has much to do with the basic camera functionality.

Create a list of the features and prioritize. What's most important? Least? Then … and this is the trick … implement the most important features first.

Does that sound trite? It is, yet every time I look at a product in trouble no one has taken this step. Developers have virtually every feature half-implemented. The ship date arrives and nothing works. Worse, there's no clear recovery strategy since so much effort has been expended on things that are not terribly important.

So in a panic management starts tossing out features. One 2002 study showed that 74% of projects wind up with 30% or more of the features being eliminated. Not only is that a terrible waste—these are partially implemented features—but the product goes to market late, with a subset of its functionality. If the system were built as I'm recommending, even schedule slippages would, at worst, result in scrubbing a few requirements that had as yet not consumed engineering time. Failure, sure, but failure in a rather successful way.

Finally, did you know great code, the really good stuff, that has the highest reliability, costs the same as cruddy software? This goes against common sense. Of course, all things being equal, highly safety-critical code is much more expensive than consumer-quality junk.

But what if we don't hold all things equal? O. Benediktsson (*Safety Critical Software and Development Productivity*, conference proceedings, Second World Conference on Software Quality, September 2000) showed that using higher and higher levels of disciplined software process lets one build higher-rel software at a constant cost. If your projects march from low reliability along an upwards line to truly safety-critical code, and if your outfit follows, in his study, increasing levels of the Capability Maturity Model, the cost remains constant.

Makes one think. And hopefully, it makes one rein in the hackers who are more focused on cranking code than specifying, designing, and carefully implementing a world-class product.

# Index