Embedded Systems Design

An Introduction to Processes, Tools, & Techniques

- Hardware/Software Partitioning
- Cross-Platform Development
- Firmware Debugging
- Performance Analysis
- Testing & Integration

CRC Press
Taylor & Francis Group

ARNOLD S. BERGER

Embedded Systems Design

An Introduction to Processes, Tools, and Techniques

Arnold Berger

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

First issued in hardback 2017.

ISBN-13: 978-1-57820-073-3 (pbk)
ISBN-13: 978-1-138-43646-6 (hbk)

Cover art design: Robert Ward

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

This book is dedicated to
Shirley Berger.

Table of Contents

Preface

Why write a book about designing embedded systems? Because my experiences working in the industry and, more recently, working with students have convinced me that there is a need for such a book.

For example, a few years ago, I was the Development Tools Marketing Manager for a semiconductor manufacturer. I was speaking with the Software Development Tools Manager at our major account. My job was to help convince the customer that they should be using our RISC processor in their laser printers. Since I owned the tool chain issues, I had to address his specific issues before we could convince him that we had the appropriate support for his design team.

Since we didn't have an In-Circuit Emulator for this processor, we found it necessary to create an extended support matrix, built around a ROM emulator, JTAG port, and a logic analyzer. After explaining all this to him, he just shook his head. I knew I was in trouble. He told me that, of course, he needed all this stuff. However, what he really needed was training. The R&D Group had no trouble hiring all the freshly minted software engineers they needed right out of college. Finding a new engineer who knew anything about software development outside of Wintel or UNIX was quite another matter. Thus was born the idea that perhaps there is some need for a different slant on embedded system design.

Recently I've been teaching an introductory course at the University of Washington-Bothell (UWB). For now, I'm teaching an introduction to embedded systems. Later, there'll be a lab course. Eventually this course will

grow into a full track, allowing students to earn a specialty in embedded systems. Much of this book's content is an outgrowth of my work at UWB. Feedback from my students about the course and its content has influenced the slant of the book. My interactions with these students and with other faculty have only reinforced my belief that we need such a book.

What is this book about?

This book is not intended to be a text in software design, or even *embedded* software design (although it will, of necessity, discuss some code and coding issues). Most of my students are much better at writing code in C++ and Java than am I. Thus, my first admission is that I'm not going to attempt to teach software methodologies. What I will teach is the *how* of software development in an embedded environment. I wrote this book to help an embedded software developer understand the issues that make embedded software development different from host-based software design. In other words, what do you do when there is no `printf()` or `malloc()`?

Because this is a book about designing embedded systems, I will discuss design issues — but I'll focus on those that aren't encountered in application design. One of the most significant of these issues is processor selection. One of my responsibilities as the Embedded Tools Marketing Manager was to help convince engineers and their managers to use our processors. What are the issues that surround the choice of the right processor for any given application? Since most new engineers usually only have architectural knowledge of the Pentium-class, or SPARC processors, it would be helpful for them to broaden their processor horizon. The correct processor choice can be a "bet the company" decision. I was there in a few cases where it was such a decision, and the company lost the bet.

Why should you buy this book?

If you are one of my students.

If you're in my class at UWB, then you'll probably buy the book because it is on your required reading list. Besides, an autographed copy of the book might be valuable a few years from now (said with a smile). However, the real reason is that it will simplify note-taking. The content is reasonably faithful to the 400 or so lectures slides that you'll have to sit through in class. Seriously, though, reading this book will help you to get a grasp of the issues that embedded system designers must deal with on a daily basis. Knowing something about embedded systems will be a big help when you become a member of the next group and start looking for a job!

If you are a student elsewhere or a recent graduate.

Even if you aren't studying embedded systems at UWB, reading this book can be important to your future career. Embedded systems is one of the largest and fastest growing specialties in the industry, but the number of recent graduates who have embedded experience is woefully small. *Any* prior knowledge of the field will make you stand out from other job applicants.

As a hiring manager, when interviewing job applicants I would often "tune out" the candidates who gave the standard, "I'm flexible, I'll do anything" answer. However, once in while someone would say, "I used your stuff in school, and boy, was it ever a *kludge*. Why did you set up the trace spec menu that way?" That was the candidate I wanted to hire. If your only benefit from reading this book is that you learn some jargon that helps you make a better impression at your next job interview, then reading it was probably worth your the time invested.

If you are a working engineer or developer.

If you are an experienced software developer this book will help you to see the big picture. If it's not in your nature to care about the big picture, you may be asking: "why do I need to see the big picture? I'm a software designer. I'm only concerned with technical issues. Let the marketing-types and managers worry about 'the big picture.' I'll take a good Quick Sort algorithm anytime." Well, the reality is that, as a developer, you are at the bottom of the food chain when it comes to making certain critical decisions, but you are at the top of the blame list when the project is late. I know from experience. I spent many long hours in the lab trying to compensate for a bad decision made by someone else earlier in the project's lifecycle. I remember many times when I wasn't at my daughter's recitals because I was fixing code. Don't let someone else stick you with the dog! This book will help you recognize and explain the critical importance of certain early decisions. It will equip you to influence the decisions that directly impact your success. You owe it to yourself.

If you are a manager.

Having just maligned managers and marketers, I'm now going to take that all back and say that this book is also for them. If you are a manager and want your project to go smoothly and your product to get to market on time, then this book can warn you about land mines and roadblocks. Will it guarantee success? No, but like chicken soup, it can't hurt.

I'll also try to share ideas that have worked for me as a manager. For example, when I was an R&D Project Manager I used a simple "trick" to

help to form my project team and focus our efforts. Before we even started the product definition phase I would get some foam-core poster board and build a box with it. The box had the approximate shape of the product. Then I drew a generic front panel and pasted it on the front of the box. The front panel had the project's code name, like *Gerbil*, or some other mildly humorous name, prominently displayed. Suddenly, we had a tangible prototype "image" of the product. We could see it. It got us focused. Next, I held a pot-luck dinner at my house for the project team and their significant others.[2] These simple devices helped me to bring the team's focus to the project that lay ahead. It also helped to form the "extended support team" so that when the need arose to call for a 60 or 80 hours workweek, the home front support was there.

(While that extended support is important, managers should not abuse it. As an R&D Manager I realized that I had a large influence over the engineer's personal lives. I could impact their salaries with large raises and I could seriously strain a marriage by firing them. Therefore, I took my responsibility for delivering the right product, on time, very seriously. You should too.)

Embedded designers and managers shouldn't have to make the same mistakes over and over. I hope that this book will expose you to some of the best practices that I've learned over the years. Since embedded system design seems to lie in the netherworld between Electrical Engineering and Computer Science, some of the methods and tools that I've learned and developed don't seem to rise to the surface in books with a homogeneous focus.

How is the book structured?

For the most part, the text will follow the classic embedded processor lifecycle model. This model has served the needs of marketing engineers and field sales engineers for many years. The good news is that this model is a fairly accurate representation of how embedded systems are developed. While no simple model truly captures all of the subtleties of the embedded development process, representing it as a parallel development of hardware and software, followed by an integration step, seems to capture the essence of the process.

What do I expect you to know?

Primarily, I assume you are familiar with the vocabulary of application development. While some familiarity with C, assembly, and basic digital

---

2. I can't take credit for this idea. I learned it from *Controlling Software Projects*, by Tom DeMarco (Yourdon Press, 1982), and from a videotaped series of his lectures.

circuits is helpful, it's not necessary. The few sections that describe specific C coding techniques aren't essential to the rest of the book and should be accessible to almost any programmer. Similarly, you won't need to be an expert assembly language programmer to understand the point of the examples that are presented in Motorola 68000 assembly language. If you have enough logic background to understand ANDs and ORs, you are prepared for the circuit content. In short, anyone who's had a few college-level programming courses, or equivalent experience, should be comfortable with the content.

Acknowledgments

I'd like to thank some people who helped, directly and indirectly, to make this book a reality. Perry Keller first turned me on to the fun and power of the in-circuit emulator. I'm forever in his debt. Stan Bowlin was the best emulator designer that I ever had the privilege to manage. I learned a lot about how it all works from Stan. Daniel Mann, an AMD Fellow, helped me to understand how all the pieces fit together.

The manuscript was edited by Robert Ward, Julie McNamee, Rita Sooby, Michelle O'Neal, and Catherine Janzen. Justin Fulmer redid many of my graphics. Rita Sooby and Michelle O'Neal typeset the final result. Finally, Robert Ward and my friend and colleague, Sid Maxwell, reviewed the manuscript for technical accuracy. Thank you all.

Arnold Berger
Sammamish, Washington
September 27, 2001

Introduction

The arrival of the microprocessor in the 1970s brought about a revolution of control. For the first time, relatively complex systems could be constructed using a simple device, the microprocessor, as its primary control and feedback element. If you were to hunt out an old Teletype ASR33 computer terminal in a surplus store and compare its innards to a modern color inkjet printer, there's quite a difference.

Automobile emissions have decreased by 90 percent over the last 20 years, primarily due to the use of microprocessors in the engine-management system. The open-loop fuel control system, characterized by a carburetor, is now a fuel-injected, closed-loop system using multiple sensors to optimize performance and minimize emissions over a wide range of operating conditions. This type of performance improvement would have been impossible without the microprocessor as a control element.

Microprocessors have now taken over the automobile. A new luxury-class automobile might have more than 70 dedicated microprocessors, controlling tasks from the engine spark and transmission shift points to opening the window slightly when the door is being closed to avoid a pressure burst in the driver's ear.

The F-16 is an unstable aircraft that cannot be flown without on-board computers constantly making control surface adjustments to keep it in the air. The pilot, through the traditional controls, sends requests to the computer to change the plane's flight profile. The computer attempts to comply with those requests to the extent that it can and still keep the plane in the air.

A modern jetliner can have more than 200 on-board, dedicated microprocessors.

The most exciting driver of microprocessor performance is the games market. Although it can be argued that the game consoles from Nintendo, Sony, and Sega are not really embedded systems, the technology boosts that they are driving are absolutely amazing. Jim Turley[1], at the Microprocessor Forum, described a 200MHz reduced instruction set computer (RISC) processor that was going into a next-generation game console. This processor could do a four-dimensional matrix multiplication in one clock cycle at a cost of $25.

Why Embedded Systems Are Different

Well, all of this is impressive, so let's delve into what makes embedded systems design different — at least different enough that someone has to write a book about it. A good place to start is to try to enumerate the differences between your desktop PC and the typical embedded system.

- Embedded systems are dedicated to specific tasks, whereas PCs are generic computing platforms.
- Embedded systems are supported by a wide array of processors and processor architectures.
- Embedded systems are usually cost sensitive.
- Embedded systems have real-time constraints.

You'll have ample opportunity to learn about *real time*. For now, real-time events are external (to the embedded system) events that must be dealt with when they occur (in real time).

- If an embedded system is using an operating system at all, it is most likely using a real-time operating system (RTOS), rather than Windows 9X, Windows NT, Windows 2000, Unix, Solaris, or HP-UX.
- The implications of software failure is much more severe in embedded systems than in desktop systems.
- Embedded systems often have power constraints.
- Embedded systems often must operate under extreme environmental conditions.
- Embedded systems have far fewer system resources than desktop systems.
- Embedded systems often store all their object code in ROM.

- Embedded systems require specialized tools and methods to be efficiently designed.
- Embedded microprocessors often have dedicated debugging circuitry.

Embedded systems are dedicated to specific tasks, whereas PCs are generic computing platforms

Another name for an embedded microprocessor is a *dedicated* microprocessor. It is programmed to perform only one, or perhaps, a few, specific tasks. Changing the task is usually associated with obsolescing the entire system and redesigning it. The processor that runs a mobile heart monitor/defibrillator is not expected to run a spreadsheet or word processor.

Conversely, a general-purpose processor, such as the Pentium on which I'm working at this moment, must be able to support a wide array of applications with widely varying processing requirements. Because your PC must be able to service the most complex applications with the same performance as the lightest application, the processing power on your desktop is truly awesome.

Thus, it wouldn't make much sense, either economically or from an engineering standpoint, to put an AMD-K6, or similar processor, inside the coffeemaker on your kitchen counter.

That's not to say that someone won't do something similar. For example, a French company designed a vacuum cleaner with an AMD 29000 processor. The 29000 is a 32-bit RISC CPU that is far more suited for driving laser-printer engines.

Embedded systems are supported by a wide array of processors and processor architectures

Most students who take my Computer Architecture or Embedded Systems class have never programmed on any platform except the X86 (Intel) or the Sun SPARC family. The students who take the Embedded Systems class are rudely awakened by their first homework assignment, which has them researching the available trade literature and proposing the optimal processor for an assigned application.

These students are learning that today more than 140 different microprocessors are available from more than 40 semiconductor vendors[2]. These vendors are in a daily battle with each other to get the design-win (be the processor of choice) for the next wide-body jet or the next Internet-based soda machine.

In Chapter 2, you'll learn more about the processor-selection process. For now, just appreciate the range of available choices.

Embedded systems are usually cost sensitive

I say "usually" because the cost of the embedded processor in the Mars Rover was probably not on the design team's top 10 list of constraints. However, if you save 10 cents on the cost of the Engine Management Computer System, you'll be a hero at most automobile companies. Cost does matter in most embedded applications.

The cost that you must consider most of the time is system cost. The cost of the processor is a factor, but, if you can eliminate a printed circuit board and connectors and get by with a smaller power supply by using a highly integrated microcontroller instead of a microprocessor and separate peripheral devices, you have potentially a greater reduction in system costs, even if the integrated device is significantly more costly than the discrete device. This issue is covered in more detail in Chapter 3.

Embedded systems have real-time constraints

I was thinking about how to introduce this section when my laptop decided to back up my work. I started to type but was faced with the hourglass symbol because the computer was busy doing other things. Suppose my computer wasn't sitting on my desk but was connected to a radar antenna in the nose of a commercial jetliner. If the computer's main function in life is to provide a collision alert warning, then suspending that task could be disastrous.

Real-time constraints generally are grouped into two categories: *time-sensitive constraints* and *time-critical constraints*. If a task is time critical, it must take place within a set window of time, or the function controlled by that task fails. Controlling the flight-worthiness of an aircraft is a good example of this. If the feedback loop isn't fast enough, the control algorithm becomes unstable, and the aircraft won't stay in the air.

A time-sensitive task can die gracefully. If the task should take, for example, 4.5ms but takes, on average, 6.3ms, then perhaps the inkjet printer will print two pages per minute instead of the design goal of three pages per minute.

If an embedded system is using an operating system at all, it is most likely using an RTOS

Like embedded processors, embedded operating systems also come in a wide variety of flavors and colors. My students must also pick an embedded operating system as part of their homework project. RTOSs are not democratic. They need not give every task that is ready to execute the time it needs. RTOSs give the highest priority task that needs to run all the time it needs. If other tasks fail to get sufficient CPU time, it's the programmer's problem.

Another difference between most commercially available operating systems and your desktop operating system is something you won't get with an RTOS. You won't get the dreaded Blue Screen of Death that many Windows 9X users see on a regular basis.

The implications of software failure are much more severe in embedded systems than in desktop systems

Remember the Y2K hysteria? The people who were really under the gun were the people responsible for the continued good health of our computer-based infrastructure. A lot of money was spent searching out and replacing devices with embedded processors because the #$%%$ thing got the dates all wrong.

We all know of the tragic consequences of a medical radiation machine that miscalculates a dosage. How do we know when our code is bug free? How do you completely test complex software that must function properly under all conditions?

However, the most important point to take away from this discussion is that software failure is far less tolerable in an embedded system than in your average desktop PC. That is not to imply that software never fails in an embedded system, just that most embedded systems typically contain some mechanism, such as a *watchdog timer*, to bring it back to life if the software loses control. You'll find out more about software testing in Chapter 9.

Embedded systems have power constraints

For many readers, the only CPU they have ever seen is the Pentium or AMD K6 inside their desktop PC. The CPU needs a massive heat sink and fan assembly to keep the processor from baking itself to death. This is not a particularly serious constraint for a desktop system. Most desktop PC's have plenty of spare space inside to allow for good airflow. However, consider an embedded system attached to the collar of a wolf roaming around Wyoming or Montana. These systems must work reliably and for a long time on a set of small batteries.

How do you keep your embedded system running on minute amounts of power? Usually that task is left up to the hardware engineer. However, the division of responsibility isn't clearly delineated. The hardware designer might or might not have some idea of the software architectural constraints. In general, the processor choice is determined outside the range of hearing of the software designers. If the overall system design is on a tight power budget, it is likely that the software design must be built around a system in which the processor is in "sleep mode" most of the time and only wakes up when a timer tick occurs. In other words, the system is completely interrupt driven.

Power constraints impact every aspect of the system design decisions. Power constraints affect the processor choice, its speed, and its memory architecture. The constraints imposed by the system requirements will likely determine whether the software must be written in assembly language, rather than C or C++, because the absolute maximum performance must be achieved within the power budget. Power requirements are dictated by the CPU clock speed and the number of active electronic components (CPU, RAM, ROM, I/O devices, and so on).

Thus, from the perspective of the software designer, the power constraints could become the dominant system constraint, dictating the choice of software tools, memory size, and performance headroom.

Speed vs. Power

Almost all modern CPUs are fabricated using the Complementary Metal Oxide Silicon (CMOS) process. The simple gate structure of CMOS devices consists of two MOS transistors, one N-type and one P-type (hence, the term complementary), stacked like a totem pole with the N-type on top and the P-type on the bottom. Both transistors behave like perfect switches. When the output is high, or logic level 1, the P-type transistor is turned off, and the N-type transistor connects the output to the supply voltage (5V, 3.3V, and so on), which the gate outputs to the rest of the circuit.

When the logic level is 0, the situation is reversed, and the P-type transistor connects the next stage to ground while the N-type transistor is turned off. This circuit topology has an interesting property that makes it attractive from a power-use viewpoint. If the circuit is static (not changing state), the power loss is extremely small. In fact, it would be zero if not for a small amount of leakage current inherent in these devices at normal room temperature and above.

When the circuit is switching, as in a CPU, things are different. While a gate switches logic levels, there is a period of time when the N-type and P-type transistors are simultaneously on. During this brief window, current can flow from the supply voltage line to ground through both devices. Current flow means power dissipation and that means heat. The greater the clock speed, the greater the number of switching cycles taking place per second, and this means more power loss. Now, consider your 500MHz Pentium or Athlon processor with 10 million or so transistors, and you can see why these desktop machines are so power hungry. In fact, it is almost a perfect linear relationship between CPU speed and power dissipation in modern processors. Those of you who overclock your CPUs to wring every last ounce of performance out of it know how important a good heat sink and fan combination are.

Embedded systems must operate under extreme environmental conditions

Embedded systems are everywhere. Everywhere means everywhere. Embedded systems must run in aircraft, in the polar ice, in outer space, in the trunk of a black Camaro in Phoenix, Arizona, in August. Although making sure that the system runs under these conditions is usually the domain of the hardware designer, there are implications for both the hardware and software. Harsh environments usually mean more than temperature and humidity. Devices that are qualified for military use must meet a long list of environmental requirements and have the documentation to prove it. If you've wondered why a simple processor, such as the 8086 from Intel, should cost several thousands of dollars in a missile, think paperwork and environment. The fact that a device must be qualified for the environment in which it will be operating, such as deep space, often dictates the selection of devices that are available.

The environmental concerns often overlap other concerns, such as power requirements. Sealing a processor under a silicone rubber conformal coating because it must be environmentally sealed also means that the capability to dissipate heat is severely reduced, so processor type and speed is also a factor.

Unfortunately, the environmental constraints are often left to the very end of the project, when the product is in testing and the hardware designer discovers that the product is exceeding its thermal budget. This often means slowing the clock, which leads to less time for the software to do its job, which translates to further refining the software to improve the efficiency of the code. All the while, the product is still not released.

Embedded systems have far fewer system resources than desktop systems

Right now, I'm typing this manuscript on my desktop PC. An oldies CD is playing through the speakers. I've got 256MB of RAM, 26GB of disk space, and assorted ZIP, JAZZ, floppy, and CD-RW devices on a SCSI card. I'm looking at a beautiful 19-inch CRT monitor. I can enter data through a keyboard and a mouse. Just considering the bus signals in the system, I have the following:

- Processor bus
- AGP bus
- PCI bus
- ISA bus
- SCSI bus
- USB bus

- Parallel bus
- RS-232C bus

An awful lot of system resources are at my disposal to make my computing chores as painless as possible. It is a tribute to the technological and economic driving forces of the PC industry that so much computing power is at my fingertips.

Now consider the embedded system controlling your VCR. Obviously, it has far fewer resources that it must manage than the desktop example. Of course, this is because it is dedicated to a few well-defined tasks and nothing else. Being engineered for cost effectiveness (the whole VCR only cost $80 retail), you can't expect the CPU to be particularly general purpose. This translates to fewer resources to manage and hence, lower cost and simplicity. However, it also means that the software designer is often required to design standard input and output (I/O) routines repeatedly. The number of inputs and outputs are usually so limited, the designers are forced to overload and serialize the functions of one or two input devices. Ever try to set the time in your super exercise workout wristwatch after you've misplaced the instruction sheet?

Embedded systems store all their object code in ROM

Even your PC has to store some of its code in ROM. ROM is needed in almost all systems to provide enough code for the system to initialize itself (boot-up code). However, most embedded systems must have all their code in ROM. This means severe limitations might be imposed on the size of the code image that will fit in the ROM space. However, it's more likely that the methods used to design the system will need to be changed because the code is in ROM.

As an example, when the embedded system is powered up, there must be code that initializes the system so that the rest of the code can run. This means establishing the run-time environment, such as initializing and placing variables in RAM, testing memory integrity, testing the ROM integrity with a checksum test, and other initialization tasks.

From the point of view of debugging the system, ROM code has certain implications. First, your handy debugger is not able to set a breakpoint in ROM. To set a breakpoint, the debugger must be able to remove the user's instruction and replace it with a special instruction, such as a TRAP instruction or software interrupt instruction. The TRAP forces a transfer to a convenient entry point in the debugger. In some systems, you can get around this problem by loading the application software into RAM. Of course, this assumes sufficient RAM is available to hold of all the applications, to store variables, and to provide for dynamic memory allocation.

Of course, being a capitalistic society, wherever there is a need, someone will provide a solution. In this case, the specialized suite of tools that have evolved to support the embedded system development process gives you a way around this dilemma, which is discussed in the next section.

Embedded systems require specialized tools and methods to be efficiently designed

Chapters 4 through 8 discuss the types of tools in much greater detail. The embedded system is so different in so many ways, it's not surprising that specialized tools and methods must be used to create and test embedded software. Take the case of the previous example—the need to set a breakpoint at an instruction boundary located in ROM.

A ROM Emulator

Several companies manufacture hardware-assist products, such as ROM emulators. Figure 1 shows a product called *NetROM*, from Applied Microsystems Corporation. NetROM is an example of a general class of tools called *emulators*. From the point of view of the target system, the ROM emulator is designed to look like a standard ROM device. It has a connector that has the exact mechanical dimensions and electrical characteristics of the ROM it is emulating. However, the connector's job is to bring the signals from the ROM socket on the target system to the main circuitry, located at the other end of the cable. This circuitry provides high-speed RAM that can be written to quickly via a separate channel from a host computer. Thus, the target system sees a ROM device, but the software developer sees a RAM device that can have its code easily modified and allows debugger breakpoints to be set.

Figure 1 NetROM.

In the context of this book, the term *hardware-assist* refers to additional specialized devices that supplement a software-only debugging solution. A ROM emulator, manufactured by companies such as Applied Microsystems and Grammar Engine, is an example of a hardware-assist device.

Embedded microprocessors often have dedicated debugging circuitry

Perhaps one of the most dramatic differences between today's embedded microprocessors and those of a few years ago is the almost mandatory inclusion of dedicated debugging circuitry in silicon on the chip. This is almost counter-intuitive to all of the previous discussion. After droning on about the cost sensitivity of embedded systems, it seems almost foolish to think that every microprocessor in production contains circuitry that is only necessary for debugging a product under development. In fact, this was the prevailing sentiment for a while. Embedded-chip manufacturers actually built special versions of their embedded devices that contained the debug circuitry and made them available (or not available) to their tool suppliers. In the end, most manufacturers found it more cost-effective to produce one version of the chip for all purposes. This didn't stop them from restricting the information about how the debug circuitry worked, but every device produced did contain the debug "hooks" for the hardware-assist tools.

What is noteworthy is that the manufacturers all realized that the inclusion of on-chip debug circuitry was a requirement for acceptance of their devices in an embedded application. That is, unless their chip had a good solution for embedded system design and debug, it was not going to be a serious contender for an embedded application by a product-development team facing time-to-market pressures.

Summary

Now that you know what is different about embedded systems, it's time to see how you actually tame the beast. In the chapters that follow, you'll examine the embedded system design process step by step, as it is practiced.

The first few chapters focus on the process itself. I'll describe the design life cycle and examine the issues affecting processor selection. The later chapters focus on techniques and tools used to build, test, and debug a complete system.

I'll close with some comments on the business of embedded systems and on an emerging technology that might change everything.

Although engineers like to think design is a rational, requirements-driven process, in the real world, many decisions that have an enormous impact on the design process are made by non-engineers based on criteria that might have little to do with the project requirements. For example, in many