

国外优秀信息科学与技术系列教学用书

嵌入式软件基础

—— C 语言与汇编的融合

(影印版)

FUNDAMENTALS OF EMBEDDED SOFTWARE

Where C and Assembly Meet

■ Daniel W. Lewis



高等教育出版社
Higher Education Press

国外优秀信息科学与技术系列教学用书

嵌入式软件基础

——C 语言与汇编的融合

(影印版)

FUNDAMENTALS OF EMBEDDED SOFTWARE

Where C and Assembly Meet

Daniel W. Lewis

PEARSON
Prentice
Hall

江苏工业学院图书馆
藏书章



高等教育出版社

图字：01-2003-8318 号

Fundamentals of Embedded Software: Where C and Assembly Meet

Daniel W. Lewis

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签。无标签者不得销售。

English reprint edition copyright ©2004 by **PEARSON EDUCATION ASIA LIMITED** and **HIGHER EDUCATION PRESS**. (Fundamentals of Embedded Software: Where C and Assembly Meet from Pearson Education's edition of the Work)

Fundamentals of Embedded Software: Where C and Assembly Meet by Daniel W. Lewis, Copyright ©2002.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall, Inc.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Regions of Hong Kong and Macao).

原版 ISBN: 0-13-061589-7

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR).

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

图书在版编目 (CIP) 数据

嵌入式软件基础：C 语言与汇编的融合 = Fundamental
s of Embedded Software: Where C and Assembly Mee
t / (美) 刘易斯 (Lewis, D. W.). —影印本. —北
京：高等教育出版社，2004.8
ISBN 7-04-014059-4

I. 嵌... II. 刘... III. 软件开发—高等学校—教
材—英文 IV. TP311.52

中国版本图书馆 CIP 数据核字 (2004) 第 065485 号

出版发行 社址 邮政编码 总机	高等教育出版社 北京市西城区德外大街 4 号 100011 010-82028899	购书热线 免费咨询 网 址	010-64054588 800-810-0598 http://www.hep.edu.cn http://www.hep.com.cn
经 销 印 刷	新华书店北京发行所 北京外文印刷厂		
开 本 印 张 字 数	787×1092 1/16 18 340 000	版 次 印 次 定 价	2004 年 8 月第 1 版 2004 年 8 月第 1 次印刷 35.00 元(附光盘)

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。

版权所有 侵权必究

Preface

How many computers are in your home? Most people might answer two or three. How many *microprocessors* are in your home? Think carefully before you answer. (*Hint:* It's a lot more than two or three! In fact, it's actually even a lot more than 10 or 20!) Today, microprocessors are embedded in almost every electronic appliance you can think of and many that you probably wouldn't. They have become pervasive—not only in our home, but in our workplace, our automobiles, airplanes, stoplights, supermarkets, cell phones—in short, in almost every aspect of our lives.

Embedded systems offer students an exciting opportunity to express their creativity. What student hasn't dreamed of designing the next new gadget that would capture the imagination of the public? Our challenge as educators is to capitalize on that excitement and channel the energy of those young minds to motivate mastery of a subject matter!

Objectives

The ultimate goal of this text is to lay a foundation that supports the multithreaded style of programming and high-reliability requirements of embedded software. Within this context, the following objectives were established:

1. To understand how data is represented at the machine level and to appreciate the consequences and limitations of those representations.
2. To master those language-specific features that are used most frequently in embedded systems, such as bit manipulation and variant access.
3. To obtain a programmer's view of processor architecture and how programming at the level of assembly is sometimes necessary or appropriate.
4. To learn about the various styles of I/O programming and, ultimately, how an event-driven approach allows one to separate data processing into a number of independent threads of computation.
5. To learn about nonpreemptive and preemptive multithreaded programming, shared resources and critical sections, and how scheduling can be used to manage system response time.
6. To reinforce basic programming skills by revisiting such topics as scope, parameter passing, recursion, and memory allocation.
7. To learn about the problems associated with shared memory objects, how shared memory is affected by memory allocation, and what programming practices can be used to minimize the occurrence of shared memory.

Intended Audience

This text is intended to serve as the basis for a sophomore-level course in a computer science, computer engineering, or electrical engineering curriculum. Such a course is envisioned as a replacement for the traditional course on computer organization and assembly language programming.

The text presents assembly the way it is most commonly used in practice—to implement small, fast, or special-purpose routines called from a main program written in a high-level language such as C. It thus covers processor organization and assembly language only from a “need-to-know” point of view, rather than as a primary objective. This approach affords time within both the text and the course to cover assembly in the context of embedded software. As a result, students not only learn that assembly still has an important role to play, but their discovery of multithreaded programming, preemptive and nonpreemptive systems, shared resources, and scheduling helps sustain their interest, feeds their curiosity, and strengthens their preparation for subsequent courses on operating systems, real-time systems, networking, and microprocessor-based design.

At most institutions, the introductory programming sequence (CS1, CS2) is no longer taught in a procedural programming language such as C or Pascal; rather, the popular approach is to now use an object-oriented programming language such as C++ or Java. Despite the change, it is not uncommon to find one or more upper division courses still using a procedural language, nor is it uncommon to find such languages still in use in industry. At the author’s institution, we solved this paradox by redesigning our traditional assembly-language course around the material in this book; it not only created room in an already packed curriculum to cover the procedural approach and to introduce the popular topic of embedded systems, but it has also offered an opportunity to strengthen student comprehension of parameter passing, scope, and memory allocation schemes that they were first introduced to in CS1 and CS2.

The text assumes that students already know how to program in C, C++, or Java, and that the similarity among the low-level syntax of those languages makes it relatively easy to move to C from either C++ or Java. Rather than covering C in excruciating detail, the text emphasizes those features of C that are employed more frequently in embedded applications, and introduces the procedural style through examples and programming assignments that include large amounts of prewritten source code. In principle, the only absolute prerequisite is thus a CS1 course that uses C, C++, or Java. However, additional programming maturity such as acquired from a CS2 course on data structures is strongly recommended.

Programming Assignments and the CD-Rom

The text is complemented by a collection of programming assignments described in Appendix D. Given that the text is aimed at sophomores, the assignments are intended primarily to illustrate a topic from the text, rather than as extended programming projects. As such, most of the source code for each assignment is provided on the CD-Rom and students are asked to focus only on those parts that relate directly to

the topic. For example, the last three assignments provide complete source code to graphically demonstrate problems associated with shared resources, priority inversion, and deadlock, and require the student to correct these problems by modifying specific parts of the code by using strategies presented in the text.

The programming assignments and when they should be scheduled relative to material in the text is summarized as follows:

Programming Assignment	Relevant Chapter	Comment
1	n/a	Introduction to C and the DJGPP compiler; requires no knowledge of material in the text, and may be scheduled during the first week of the course.
2	2	Fixed-precision real numbers in C.
3	3	Macros and packed operands in C.
4	n/a	makefiles: Typically assigned while studying Chapter 4, but has no direct relationship to that material.
5 and 6	5	Assembly-language programming.
7	6	Interrupt-driven I/O in assembly.
8	7	Multi-threaded programming with a nonpreemptive kernel.
9	7	Preemptive kernels, shared resources, and semaphores.
10 and 11	8	Scheduling problems (priority inversion and deadlock).

On the CD-Rom you'll also find all the software tools needed to develop embedded applications under Microsoft Windows 9X, 2000 and NT: the DJGPP port of the gnu C compiler and linker, a compatible assembler and run-time libraries. A boot loader is provided to load (and execute) the embedded application into memory from diskette. Where possible, source code for each of these tools has also been included. Directions for using each of these tools are found at the beginning of Appendix D.

Choice of Platform

The text uses the ubiquitous PC as a platform for learning about processor architecture, assembly language, and embedded software. Two primary factors motivated this choice: (1) Students will ultimately encounter the Intel architecture due to its dominance within the PC market, and (2) choosing to build embedded applications on the PC has the added benefit of allowing instructors to offer an associated laboratory component without investing in specialized single board computers.

Most assembly-language programming textbooks for the Intel processor cover the original "real" mode of the 8088. However, the protected mode of the 386 and later Intel processors is much more representative of modern architecture and is actually much easier to program when configured to use a "flat" memory model. Although real mode is covered briefly in the text, the emphasis is on protected mode and is used in all the of assembly-language examples.

One should not construe that a PC is the best (or even an appropriate) platform for building *actual* embedded systems. Most embedded applications have no need for many of the PC's standard peripherals (e.g., display, keyboard, disk), and the power-on boot procedure in the ROM BIOS doesn't even support diskless applications. Trying to write initialization code to replace the BIOS is difficult at best, because it requires knowledge of chipset-dependent features that vary from PC to PC and which are often proprietary.

Acknowledgments

This book and the software on the companion CD would not have been possible without the efforts of a number of people. Of them all, I owe a special thanks to the students and teaching assistants of COEN 20 at Santa Clara University who suffered through the rough early versions of the manuscript and helped to debug the libepc library routines.

I also owe a debt of gratitude to many of my colleagues: To Qiang Li and Neil Quinn, who endured the unenviable task of teaching from my materials and who graciously suggested a number of improvements to the text. To Hal Brown and Vasu Alagar, who helped me develop a mathematical representation of fixed-point multiplication of real numbers that led to an efficient straight-line implementation in assembly and an explanation of the algorithm in section 2.3.3 that students find relatively easy to understand. To Darren Atkinson, who spent the better part of two days in my office tracking down an innocuous little software bug that only appeared when I upgraded to a more recent version of the compiler. And a hearty thanks to early reviewers of my manuscript, whose comments helped to strengthen the final book. They are Dr. Walter Higgins of Arizona State University, Dr. Karram Mossaad of the University of Texas at Austin, and Dr. Dana Lasher of North Carolina State University. Special thanks to Dr. Lasher, who contributed the telephone call analogy that appears in the opening paragraph of section 6.4 on Interrupt-Driven I/O.

We should *all* be grateful to those unselfish programmers who have created professional quality software tools and then made them readily available for anyone to use: To DJ Delorie for his DJGPP port of the GNU C compiler, to Simon Tatham and Julian Hall for their NASM assembler, to Jean Labrosse for his μ C/OS-II pre-emptive real-time kernel, and to Dennis Saunders of MIX Software for their Multi-C non-preemptive real-time kernel.

Daniel W. Lewis

Prentice-Hall companion website:
www.prenhall.com/divisions/ems/app/lewis

出版说明

20 世纪末,以计算机和通信技术为代表的信息科学和技术对世界经济、科技、军事、教育和文化等产生了深刻影响。信息科学技术的迅速普及和应用,带动了世界范围信息产业的蓬勃发展,为许多国家带来了丰厚的回报。

进入 21 世纪,尤其随着我国加入 WTO,信息产业的国际竞争将更加激烈。我国信息产业虽然在 20 世纪末取得了迅猛发展,但与发达国家相比,甚至与印度、爱尔兰等国家相比,还有很大差距。国家信息化的发展速度和信息产业的国际竞争能力,最终都将取决于信息科学技术人才的质量和数量。引进国外信息科学和技术优秀教材,在有条件的学校推动开展英语授课或双语教学,是教育部为加快培养大批高质量的信息技术人才采取的一项重要举措。

为此,教育部要求由高等教育出版社首先开展信息科学和技术教材的引进试点工作。同时提出了两点要求,一是要高水平,二是要低价格。在高等教育出版社和信息科学技术引进教材专家组的努力下,经过比较短的时间,第一批由教育部高等教育司推荐的 20 多种引进教材已经陆续出版。这套教材出版后受到了广泛的好评,其中有不少是世界信息科学技术领域著名专家、教授的经典之作和反映信息科学技术最新进展的优秀作品,代表了目前世界信息科学技术教育的一流水平,而且价格也是最优惠的,与国内同类自编教材相当。这套教材基本覆盖了计算机科学与技术专业的课程体系,体现了权威性、系统性、先进性和经济性等特点。

目前,教育部正在全国 35 所高校推动示范性软件学院的建设,这也是加快培养信息科学技术人才的重要举措之一。为配合软件学院的教学工作,结合各软件学院的教学计划和课程设置,高等教育出版社近期聘请有关专家和软件学院的教师遴选推荐了一批相应的原版教学用书,正陆续组织出版,以方便各软件学院开展双语教学。

我们希望这些教学用书的引进出版,对于提高我国高等学校信息科学技术的教学水平,缩小与国际先进水平的差距,加快培养一大批具有国际竞争力的高质量信息技术人才,起到积极的推动作用。同时我们也欢迎广大教师和专家们对我们的教材引进工作提出宝贵的意见和建议。联系方式: hep.cs@263.net。

高等教育出版社

二〇〇二年九月

The greatest challenge to writing a book is finding the time, and working on it at home in the evenings and on weekends often becomes an unavoidable way of life for several months. Throughout it all my wife and children have been more understanding, patient, and supporting than I would ever have expected. Thank you. This book is dedicated to you.

Contents

Preface *xiii*

Chapter 1 **Introduction** *1*

- 1.1 What is an Embedded System? *1*
- 1.2 What's Unique About the Design Goals for Embedded Software? *3*
- 1.3 What Does "Real-Time" Mean? *5*
- 1.4 What Does "Multitasking" Mean? *6*
- 1.5 How Powerful Are Embedded Processors? *7*
- 1.6 What Programming Languages Are Used? *7*
- 1.7 What Is a "Real-Time Kernel"? *8*
- 1.8 How Is Building an Embedded Application Unique? *9*
- 1.9 How Big Are Typical Embedded Programs? *11*
- 1.10 The Software Used in This Book *12*
- Problems *14*

Chapter 2 **Data Representation** *15*

- 2.1 Fixed-Precision Binary Numbers *15*
 - 2.1.1 Positional Number Systems *16*
 - 2.1.2 Binary-to-Decimal Conversion *17*
 - 2.1.3 Decimal-to-Binary Conversion *17*
 - 2.1.4 Counting *19*
 - 2.1.5 Fixed Precision and Rollover *19*
 - 2.1.6 Hexadecimal Representation *20*
- 2.2 Binary Representation of Integers *21*
 - 2.2.1 Signed Integers *21*
 - 2.2.2 Positive and Negative Representations of the Same Magnitude *22*
 - 2.2.3 Interpreting the Value of a 2's-Complement Number *23*
 - 2.2.4 More on Range and Overflow *24*
 - 2.2.5 2's Complement and Hardware Complexity *25*
- 2.3 Binary Representation of Real Numbers *28*
 - 2.3.1 Fixed-Point Representation *28*
 - 2.3.2 Fixed-Point Using a Universal 16.16 Format *30*
 - 2.3.3 Fixed-Point Using a Universal 32.32 Format *32*
 - 2.3.4 Floating-Point Representation *35*

- 2.4 ASCII Representation of Text 37
- 2.5 Binary-Coded Decimal (BCD) 39
- Problems 40

Chapter 3 Getting the Most Out of C 43

- 3.1 Integer Data Types 43
- 3.2 Mixing Data Types 46
- 3.3 Useful Typedefs and Defines 47
- 3.4 Manipulating Bits in Memory 48
 - 3.4.1 Testing Bits 50
 - 3.4.2 Setting, Clearing, and Inverting Bits 51
 - 3.4.3 Extracting Bits 52
 - 3.4.4 Inserting Bits 52
- 3.5 Manipulating Bits in I/O Ports 53
 - 3.5.1 Write-Only I/O Ports 53
 - 3.5.2 Ports Differentiated by Reads Versus Writes 54
 - 3.5.3 Ports Differentiated by Sequential Access 54
 - 3.5.4 Ports Differentiated by Bits in the Written Data 55
- 3.6 Accessing Memory-Mapped I/O Devices 55
 - 3.6.1 Accessing Data Through a Pointer 55
 - 3.6.2 Arrays, Pointers, and the “Address Of” Operator 56
- 3.7 Structures 58
 - 3.7.1 Packed Structures 59
 - 3.7.2 Bit Fields 60
- 3.8 Variant Access 61
 - 3.8.1 Casting the Address of an Object 61
 - 3.8.2 Using Unions 63
- Problems 63

Chapter 4 A Programmer’s View of Computer Organization 65

- 4.1 Memory 65
- 4.2 The Central Processing Unit (CPU) 67
 - 4.2.1 The Arithmetic and Logic Unit (ALU) 67
 - 4.2.2 Other Registers 68
 - 4.2.3 The Control Unit 69
- 4.3 Input/Output (I/O) 70
- 4.4 Introduction to the Intel Architecture 71
 - 4.4.1 Instruction Formats 72
 - 4.4.2 Instruction Operands 73
 - 4.4.3 Operand Restrictions 74

4.4.4	Registers	75
4.4.5	The Stack	77
4.5	The Intel Real Mode Architecture	78
4.5.1	Segmented Addressing	79
4.5.2	Addressing Modes	81
4.6	The Intel Protected Mode Architecture	83
4.6.1	Segment Registers and The Global Descriptor Table	84
4.6.2	The Flat Memory Model	85
4.6.3	Addressing Modes	85
4.7	Operand and Address-Size Override Prefixes	86
4.8	The Intel Data Manipulation Instructions	86
4.8.1	Data Movement, Stack, and I/O Instructions	87
4.8.2	Arithmetic Instructions	89
4.8.3	Bitwise Instructions	91
4.8.4	Shift Instructions	91
	Problems	93

Chapter 5 Mixing C and Assembly 96

5.1	Programming in Assembly	96
5.2	Register Usage Conventions	98
5.3	Typical Use of Addressing Options	98
5.3.1	Accessing Data Whose Address is a Constant	99
5.3.2	Accessing Data Whose Address is a Variable	100
5.4	Instruction Sequencing	101
5.4.1	Compound Conditionals	102
5.4.2	If-Then-Else Statements	104
5.4.3	Building Loops	105
5.4.4	Faster Loops with String Instructions	106
5.5	Procedure Call and Return	107
5.6	Parameter Passing	108
5.7	Retrieving Parameters	110
5.8	Everything is Pass by Value	112
5.9	Temporary Variables	112
	Problems	115

Chapter 6 Input/Output Programming 117

6.1	The Intel I/O Instructions	118
6.2	Synchronization, Transfer Rate, and Latency	118
6.3	Polled Waiting Loops	119

6.4	Interrupt-Driven I/O	121
6.4.1	The Hardware Response	121
6.4.2	The Interrupt Service Routine	124
6.4.3	Programmable Interrupt Controllers	125
6.4.4	Buffers and Queues	126
6.4.5	Writing Interrupt Service Routines in Assembly	128
6.4.6	Writing Interrupt Service Routines in C	129
6.4.7	Nonmaskable Interrupts	130
6.4.8	Software Interrupts	130
6.4.9	Exceptions	132
6.5	Direct Memory Access	132
6.5.1	Double Buffering	133
6.6	Comparison of Methods	134
	Problems	135

Chapter 7 Concurrent Software 138

7.1	Foreground/Background Systems	138
7.1.1	Thread State and Serialization	139
7.1.2	Managing Latency	139
7.1.3	Preventing Interrupt Overrun	143
7.1.4	Moving Work into the Background	144
7.2	Multithreaded Programming	145
7.2.1	Concurrent Execution of Independent Threads	145
7.2.2	Context Switching	146
7.2.3	Nonpreemptive (Cooperative) Multitasking	147
7.2.4	Preemptive Multitasking	147
7.3	Shared Resources and Critical Sections	148
7.3.1	Disabling Interrupts	150
7.3.2	Disabling Task Switching	150
7.3.3	Spin Locks	151
7.3.4	Mutex Objects	152
7.3.5	Semaphores	152
	Problems	153

Chapter 8 Scheduling 155

8.1	Thread States	155
8.2	Pending Threads	156
8.3	Context Switching	157
8.4	Round-Robin Scheduling	158

8.5	Priority-Based Scheduling	159
8.5.1	Priority Inversion	159
8.5.2	The Priority Inheritance Protocol	160
8.5.3	The Priority Ceiling Protocol	161
8.6	Assigning Priorities	161
8.6.1	Deadline-Driven Scheduling	161
8.6.2	Rate-Monotonic Scheduling	162
8.7	Deadlock	163
8.8	Watchdog Timers	164
	Problems	166

Chapter 9 Memory Management 168

9.1	Objects in C	168
9.2	Scope	169
9.2.1	Refining Local Scope	169
9.2.2	Refining Global Scope	170
9.3	Lifetime	171
9.4	Automatic Allocation	172
9.4.1	Storage Class "Register"	173
9.5	Static Allocation	174
9.6	Three Programs to Distinguish Static from Automatic	174
9.6.1	Object Creation	175
9.6.2	Object Initialization	175
9.6.3	Object Destruction	176
9.7	Dynamic Allocation	177
9.7.1	Fragmentation	178
9.7.2	Memory Allocation Pools	179
9.8	Automatic Allocation with Variable Size (alloca)	179
9.8.1	Variable-Size Arrays	180
9.9	Recursive Functions and Memory Allocation	181
	Problems	182

Chapter 10 Shared Memory 189

10.1	Recognizing Shared Objects	189
10.1.1	Shared Global Data	190
10.1.2	Shared Private Data	190
10.1.3	Shared Functions	190
10.2	Reentrant Functions	190

10.3	Read-Only Data	191
10.3.1	Type Qualifier "const"	191
10.4	Coding Practices to Avoid	192
10.4.1	Functions That Keep Internal State in Local Static Objects	192
10.4.2	Functions That Return the Address of a Local Static Object	194
10.5	Accessing Shared Memory	195
10.5.1	The Effect of Processor Word Size	196
10.5.2	Read-Only and Write-Only Access	197
10.5.3	Type Qualifier "volatile"	198
	Problems	200

Chapter 11 System Initialization 203

11.1	Memory Layout	203
11.2	The CPU	204
11.2.1	Setting Up a Flat Memory Model	204
11.2.2	Switching into Protected Mode	207
11.3	C Run-Time Environment	207
11.3.1	Copying from ROM to RAM	208
11.3.2	Zeroing Uninitialized Statics	208
11.3.3	Setting Up a Heap	209
11.4	System Timer	211
11.4.1	Timer 0: Timer Tick	211
11.4.2	Timer 1: Memory Refresh	212
11.4.3	Timer 2: Speaker Frequency	212
11.5	Interrupt System	213
11.5.1	Initializing the IDT	213
11.5.2	Initializing the 8259 PICs	215
11.5.3	Installing a New Interrupt Service Routine	216

Appendix A : Contents of the CD-Rom 219

Appendix B : The DJGPP C/C++ Compiler 220

Installation	220
Compilation	221
On-Line Documentation (Info)	222

Appendix C : The NASM Assembler 223

Installation	223
Running NASM	223

Appendix D : Programming Projects 225

- Files Required from the CD-ROM for All Applications 225
- Files Required for Nonpreemptive Multithreaded Applications 225
- Files Required for Preemptive Multithreaded Applications 226
- Compiling and Assembling Your Embedded Application 226
- Linking Your Embedded Application 226
- Preparing the Boot Diskette 227
- Running Your Embedded Application 227
 - Program 1: Getting Started with the DJGPP Compiler Tools 228
 - Program 2: Using Fixed-Point Real Numbers 230
 - Program 3: Using Macros and Packed Operands 231
 - Program 4: Using “Makefiles” 232
 - Program 5: Coding Extended Precision Multiplication in Assembly 235
 - Program 6: Coding Extended Precision Division in Assembly 237
 - Program 7: Polled Waiting Loop and Interrupt-Driven I/O 238
 - Program 8: A Simple Nonpreemptive Multithreaded Application 240
 - Program 9: Preemptive Kernels and Shared Resources 242
 - Program 10: Avoiding Unbounded Priority Inversion 245
 - Program 11: Avoiding Deadlock 246

Appendix E : The libeipc Library 247

- Memory Layout and Initialization 247
- Display Functions (display.c) 248
- Window Functions (window.c) 250
- Keyboard Functions (keyboard.c) 251
- Speaker Functions (speaker.c) 252
- Timer Functions (timer.c, cycles.asm) 252
- Interrupt Vector Access Functions (init-idt.c) 253
- Dynamic Memory Allocation Functions (heap.c) 254
- Fixed Point (fixedpt.asm) 254
- Interfunction Jumps (setjmp.asm) 255
- Miscellaneous Functions (init-crt.c) 256

Appendix F : The Boot Loader 257**Index 258**

C H A P T E R 1

Introduction

1.1 WHAT IS AN EMBEDDED SYSTEM?

Embedded systems are electronic devices that incorporate microprocessors within their implementations. The main purposes of the microprocessor are to simplify system design and provide flexibility. Having a microprocessor in the device means that removing bugs, making modifications, or adding new features are only matters of rewriting the software that controls the device. Unlike PCs, however, embedded systems may not have a disk drive and so the software is often stored in a read-only memory (ROM) chip; this means that modifying the software requires either replacing or “reprogramming” the ROM.

As Table 1-1 indicates, embedded systems are found in a wide range of application areas. Originally they were used only for expensive industrial-control applications, but as technology brought down the cost of dedicated processors, they began to appear in moderately expensive applications such as automobiles, communications and office equipment, and televisions. Today’s embedded systems are so inexpensive that they are used in almost every electronic product in our life.

In many cases we’re not even aware that a computer is present and so don’t realize just how pervasive they have become. For example, although the typical family may own only one or two personal computers, the number of embedded computers found within their home and cars and among their personal belongings is much greater.

What is often surprising is that embedded processors account for virtually 100% of worldwide microprocessor production! For every microprocessor produced for use

FIGURE 1-1 NASA’s Mars Sojourner Rover used an Intel 80C85 8-bit microprocessor.
Courtesy of NASA/JPL.

