Meets Requirements for Safety-Critical Systems

MicroC/OS-II

The Real-Time Kernel

Second Edition

Use this complete
portable, ROMable, scalable
preemptive RTOS
in your own product

JEAN J. LABROSSE

MicroC/OS-II

The Real-Time Kernel

Second Edition

Jean J. Labrosse

Focal Press
Taylor & Francis Group
NEW YORK AND LONDON

First published 2002 by CMP Books

This edition published 2015 by Focal Press
70 Blanchard Road, Suite 402, Burlington, MA 01803

and by Focal Press
2 Park Square, Milton Park, Abingdon, Oxon OX14 4RN

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

Notices
Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

ISBN 13: 978-1-57820-103-7 (hbk)

Cover art design: Robert Ward

To my loving and caring wife, Manon, and to our two lovely children, James and Sabrina.

Table of Contents

Preface . . . . . **xv**
Meets the Requirements of Safety-Critical Systems xv
What's New in this Edition? xv
µC/OS-II Goals xvii
Intended Audience xvii
What You Need to Use µC/OS-II xvii
The µC/OS Story xvii
Acknowledgments xx

Introduction . . . . . **xxi**
µC/OS-II Features xxi
Figures, Listings, and Tables xxiii
Chapter Contents xxiii
µC/OS-II Web Site xxvi

Chapter 1 Getting Started with µC/OS-II **1**
1.00 Installing µC/OS-II 1
1.01 Example #1 2
1.02 Example #2 10
1.03 Example #3 20
1.04 Example #4 31

Chapter 2 Real-time Systems Concepts 35
2.00 Foreground/Background Systems 36
2.01 Critical Sections of Code 37
2.02 Resources 37
2.03 Shared Resources 37
2.04 Multitasking 37
2.05 Tasks 37
2.06 Context Switches (or Task Switches) 39
2.07 Kernels 39
2.08 Schedulers 40
2.09 Non-Preemptive Kernels 40
2.10 Preemptive Kernels 42
2.11 Reentrant Functions 43
2.12 Round-Robin Scheduling 45
2.13 Task Priorities 45
2.14 Static Priorities 45
2.15 Dynamic Priorities 45
2.16 Priority Inversions 45
2.17 Assigning Task Priorities 48
2.18 Mutual Exclusion 49
2.19 Deadlock (or Deadly Embrace) 57
2.20 Synchronization 57
2.21 Event Flags 59
2.22 Intertask Communication 60
2.23 Message Mailboxes 60
2.24 Message Queues 61
2.25 Interrupts 62
2.26 Interrupt Latency 62
2.27 Interrupt Response 63
2.28 Interrupt Recovery 64
2.29 Interrupt Latency, Response, and Recovery 64
2.30 ISR Processing Time 66
2.31 Nonmaskable Interrupts 66
2.32 Clock Tick 68
2.33 Memory Requirements 70
2.34 Advantages and Disadvantages of Real-Time Kernels 71
2.35 Real-Time Systems Summary 71

Chapter 3 Kernel Structure 73
3.00 Critical Sections, OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() 74
3.01 Tasks 78
3.02 Task States 79
3.03 Task Control Blocks (OS_TCB) 81
3.04 Ready List 88
3.05 Task Scheduling 90
3.06 Task Level Context Switch, OS_TASK_SW() 92
3.07 Locking and Unlocking the Scheduler 96
3.08 Idle Task 98
3.09 Statistics Task 99
3.10 Interrupts Under µC/OS-II 103
3.11 Clock Tick 108
3.12 µC/OS-II Initialization 111
3.13 Starting µC/OS-II 114
3.14 Obtaining the Current µC/OS-II Version 116

Chapter 4 Task Management 117
4.00 Creating a Task, OSTaskCreate() 118
4.01 Creating a Task, OSTaskCreateExt() 120
4.02 Task Stacks 123
4.03 Stack Checking, OSTaskStkChk() 125
4.04 Deleting a Task, OSTaskDel() 129
4.05 Requesting to Delete a Task, OSTaskDelReq() 132
4.06 Changing a Task's Priority,OSTaskChangePrio() 136
4.07 Suspending a Task, OSTaskSuspend() 139
4.08 Resuming a Task, OSTaskResume() 141
4.09 Getting Information about a Task, OSTaskQuery() 142

Chapter 5 Time Management 145
5.00 Delaying a Task, OSTimeDly() 146
5.01 Delaying a Task, OSTimeDlyHMSM() 148
5.02 Resuming a Delayed Task,OSTimeDlyResume() 150
5.03 System Time, OSTimeGet() and OSTimeSet() 151

Chapter 6 Event Control Blocks 153
6.00 Placing a Task in the ECB Wait List 156
6.01 Removing a Task from an ECB Wait List 157
6.02 Finding the Highest Priority Task Waiting on an ECB 157

6.03 List of Free ECBs 159
6.04 Initializing an ECB, OS_EventWaitListInit() 160
6.05 Making a Task Ready, OS_EventTaskRdy() 161
6.06 Making a Task Wait for an Event, OS_EventTaskWait() . . 163
6.07 Making a Task Ready Because of a Timeout, OS_EventTO() 164

Chapter 7 Semaphore Management 165
7.00 Creating a Semaphore, OSSemCreate() 166
7.01 Deleting a Semaphore, OSSemDel() 168
7.02 Waiting on a Semaphore (Blocking), OSSemPend() 171
7.03 Signaling a Semaphore, OSSemPost() 173
7.04 Getting a Semaphore Without Waiting (Non-blocking), OSSemAccept() 175
7.05 Obtaining the Status of a Semaphore, OSSemQuery() 176

Chapter 8 Mutual Exclusion Semaphores 179
8.00 Creating a Mutex, OSMutexCreate() 183
8.01 Deleting a Mutex, OSMutexDel() 185
8.02 Waiting on a Mutex (Blocking), OSMutexPend() 188
8.03 Signaling a Mutex, OSMutexPost() 191
8.04 Getting a Mutex without Waiting (Non-blocking), OSMutexAccept() 194
8.05 Obtaining the Status of a Mutex, OSMutexQuery() 195

Chapter 9 Event Flag Management 199
9.00 Event Flag Internals 200
9.01 Creating an Event Flag Group, OSFlagCreate() 203
9.02 Deleting an Event Flag Group, OSFlagDel() 204
9.03 Waiting for Event(s) of an Event Flag Group, OSFlagPend() 207
9.04 Setting or Clearing Event(s) in an Event Flag Group, OSFlagPost() 215
9.05 Looking for Event(s) of an Event Flag Group, OSFlagAccept() 224
9.06 Querying an Event Flag Group, OSFlagQuery() 227

Chapter 10 Message Mailbox Management 229
10.00 Creating a Mailbox, OSMboxCreate() 230
10.01 Deleting a Mailbox, OSMboxDel() 232

10.02 Waiting for a Message at a Mailbox, OSMboxPend() 235
10.03 Sending a Message to a Mailbox, OSMboxPost() 238
10.04 Sending a Message to a Mailbox, OSMboxPostOpt() 239
10.05 Getting a Message without Waiting (Non-blocking), OSMboxAccept() 241
10.06 Obtaining the Status of a Mailbox, OSMboxQuery() 242
10.07 Using a Mailbox as a Binary Semaphore 244
10.08 Using a Mailbox Instead of OSTimeDly() 245

Chapter 11 Message Queue Management 247
11.00 Creating a Message Queue, OSQCreate() 251
11.01 Deleting a Message Queue, OSQDel() 253
11.02 Waiting for a Message at a Queue (Blocking), OSQPend() ..256
11.03 Sending a Message to a Queue (FIFO), OSQPost() 259
11.04 Sending a Message to a Queue (LIFO), OSQPostFront() 261
11.05 Sending a Message to a Queue (FIFO or LIFO), OSQPostOpt() 262
11.06 Getting a Message Without Waiting, OSQAccept() 265
11.07 Flushing a Queue, OSQFlush() 267
11.08 Obtaining the Status of a Queue, OSQQuery() 268
11.09 Using a Message Queue When Reading Analog Inputs ... 270
11.10 Using a Queue as a Counting Semaphore 271

Chapter 12 Memory Management 273
12.00 Memory Control Blocks 274
12.01 Creating a Partition, OSMemCreate() 276
12.02 Obtaining a Memory Block, OSMemGet() 279
12.03 Returning a Memory Block, OSMemPut() 280
12.04 Obtaining Status of a Memory Partition, OSMemQuery() ...282
12.05 Using Memory Partitions 283
12.06 Waiting for Memory Blocks from a Partition 285

Chapter 13 Porting µC/OS-II 287
13.00 Development Tools 289
13.01 Directories and Files 290
13.02 INCLUDES.H 291
13.03 OS_CPU.H 291
13.04 OS_CPU_C.C 297
13.05 OS_CPU_A.ASM 304
13.06 Testing a Port 310

OSCtxSw() 322
OSInitHookBegin() 323
OSInitHookEnd() 324
OSIntCtxSw() 325
OSStartHighRdy() 326
OSTaskCreateHook() 327
OSTaskDelHook() 328
OSTaskIdleHook() 329
OSTaskStatHook() 330
OSTaskStkInit() 331
OSTaskSwHook() 333
OSTCBInitHook() 334
OSTickISR() 335
OSTimeTickHook() 336

Chapter 14 80x86 Port 337
Real Mode, Large Model with Emulated Floating-Point Support
14.00 Development Tools 339
14.01 Directories and Files 340
14.02 INCLUDES.H 341
14.03 OS_CPU.H 341
14.04 OS_CPU_C.C 345
14.05 OS_CPU_A.ASM 357
14.06 Memory Usage 370

Chapter 15 80x86 Port 377
Real Mode, Large Model with Hardware Floating-Point Support
15.00 Development Tools 377
15.01 Directories and Files 380
15.02 INCLUDES.H 380
15.03 OS_CPU.H 381
15.04 OS_CPU_C.C 383
15.05 OS_CPU_A.ASM 393
15.06 Memory Usage 402

Chapter 16 µC/OS-II Reference Manual 405
OS_ENTER_CRITICAL() 406
OS_EXIT_CRITICAL() 406

OSFlagAccept() 407
OSFlagCreate() 409
OSFlagDel() 410
OSFlagPend() 412
OSFlagPost() 414
OSFlagQuery() 416
OSInit() 417
OSIntEnter() 418
OSIntExit() 420
OSMboxAccept() 421
OSMboxCreate() 422
OSMboxDel() 423
OSMboxPend() 425
OSMboxPost() 427
OSMboxPostOpt() 429
OSMboxQuery() 431
OSMemCreate() 433
OSMemGet() 435
OSMemPut() 437
OSMemQuery() 439
OSMutexAccept() 441
OSMutexCreate() 443
OSMutexDel() 445
OSMutexPend() 447
OSMutexPost() 449
OSMutexQuery() 451
OSQAccept() 453
OSQCreate() 454
OSQDel() 455
OSQFlush() 457
OSQPend() 458
OSQPost() 460
OSQPostFront() 462
OSQPostOpt() 464
OSQQuery() 466
OSSchedLock() 468
OSSchedUnlock() 469
OSSemAccept() 470
OSSemCreate() 471
OSSemDel() 472
OSSemPend() 474

OSSemPost() 476
OSSemQuery() 478
OSStart() 480
OSStatInit() 481
OSTaskChangePrio() 482
OSTaskCreate() 483
OSTaskCreateExt() 487
OSTaskDel() 493
OSTaskDelReq() 495
OSTaskQuery() 497
OSTaskResume() 499
OSTaskStkChk() 500
OSTaskSuspend() 502
OSTimeDly() 504
OSTimeDlyHMSM() 505
OSTimeDlyResume() 507
OSTimeGet() 508
OSTimeSet() 509
OSTimeTick() 510
OSVersion() 512

Chapter 17 µC/OS-II Configuration Manual 513

17.00 Miscellaneous 513
17.01 Event Flags 516
17.02 Message Mailboxes 516
17.03 Memory Management 517
17.04 Mutual Exclusion Semaphores 517
17.05 Message Queues 518
17.06 Semaphores 519
17.07 Task Management 519
17.08 Time Management 520
17.09 Function Summary 520

Chapter 18 PC Services 525

18.00 Character-Based Display 525
18.01 Saving and Restoring DOS's Context 529
18.02 Elapsed-Time Measurement 531
18.03 Miscellaneous 531
18.04 Interface Functions 532
PC_DispChar() 533
PC_DispClrCol() 534

PC_DispClrRow() 535
PC_DispClrScr() 536
PC_DispStr() 537
PC_DOSReturn() 539
PC_DOSSaveReturn() 540
PC_ElapsedInit() 541
PC_ElapsedStart() 542
PC_ElapsedStop() 544
PC_GetDateTime() 545
PC_GetKey() 546
PC_SetTickRate() 547
PC_VectGet() 548
PC_VectSet() 549
18.05 Bibliography 550

Appendix A C Coding Conventions. 551
A.1 Header 552
A.2 Include Files 552
A.3 Naming Identifiers 553
A.4 Acronyms, Abbreviations, and Mnemonics 554
A.5 Comments 556
A.6 #defines 557
A.7 Data Types 557
A.8 Local Variables 558
A.9 Function Prototypes 559
A.10 Function Declarations 559
A.11 Indentation 560
A.12 Statements and Expressions 563
A.13 Structures and Unions 564
A.14 Bibliography 564

Appendix B Licensing Policy for µC/OS-II 567
B.1 Colleges and Universities 567
B.2 Commercial Use 567

Appendix C µC/OS-II Quick Reference 569
Miscellaneous 570
Task Management 571
Time Management 573

Semaphore Management 574
Mutual Exclusion Semaphore Management 575
Event Flag Management 576
Message Mailbox Management 577
Message Queue Management 579
Memory Management 581

Appendix D TO Utility 583

Appendix E Bibliography 585

Appendix F Companion CD 587
F.1 Files and Directories 589

Index 593

What's on the CD-ROM? 614

Preface

Ten years ago (1992), I wrote my first book called, *µC/OS, The Real-Time Kernel.* Towards the end of 1998, it was replaced by *MicroC/OS-II, The Real-Time Kernel*. The word *Micro* now replaces the Greek letter *µ* on the book cover because bookstores didn't know how to file µC/OS properly. However, for all intents and purposes, MicroC/OS and µC/OS are synonymous, and, in this book, I mostly use µC/OS-II. This is the second edition of µC/OS-II but, in a way, the third edition of the µC/OS series.

Meets the Requirements of Safety-Critical Systems

In July of 2000, µC/OS-II was certified in an avionics product by the Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of the RTCA DO-178B standard for software used in avionics equipment. In order to meet the requirements of this standard, it must be possible to demonstrate through documentation and testing that the software is both robust and safe. This issue is particularly important for an operating system as it demonstrates that it has the proven quality to be usable in any application. Every feature, function, and line of code of µC/OS-II has been examined and tested to demonstrate that it is safe and robust enough to be used in safety-critical systems where human life is on the line.

What's New in this Edition?

This book has been completely revised since the first edition of *MicroC/OS-II, The Real-Time Kernel*.

More Chapters

The previous edition contained 12 chapters while this edition has 18. I decided to break the old Chapter 6 (Intertask Communications & Synchronization) into six chapters. I now dedicate a whole chapter to event control blocks (ECBs), one for semaphores, one for mutual exclusion semaphores, one for event flags, one for message mailboxes, and finally, one for message queues.

The previous edition contained a port for the Intel 80x86 family of processors, but this port only handled context switching of integer registers. I added a chapter that describes a port that also saves and restores floating-point registers, which are common to the 80486 and Pentium processors.

I also added a chapter that describes the services I use from a PC.

Finally, I added two appendices: Coding Conventions and a µC/OS-II Quick Reference.

Removed Chapters

I decided to remove the chapter on porting µC/OS to µC/OS-II because very few people are still using µC/OS because µC/OS-II offers so much more.

I also removed the appendix on `HPLISTC` because most good code editors allow you to neatly print source listings.

Removed Code Listings

I decided to remove the code listings that were found in Appendices A, B, and C. I have three reasons for removing the listings. First, this edition contains over 150 pages of new material. If I were to leave the listings in the appendices, this book would exceed 750 pages and would be a monster to carry around (it's already big as it is). The second reason is that the code comes on the companion CD, and it's better to refer to the code using a computer anyway. Also, the code is already described in the book, so the appendices were a duplication of the code. Finally, like any piece of software, µC/OS-II is subject to changes and upgrades. Because of this, the listings in the appendices become obsolete over time and thus have little value.

Additional Services

The code for µC/OS-II is basically the same as the previous edition, except for the addition of new services. The previous edition contained the following services:

- Time management
- Binary and counting semaphores
- Message mailboxes
- Message queues
- Fixed-sized memory block manager

This new edition adds:

- Mutual exclusion semaphores (mutexes)
- Event flags

More Examples

In some of the chapters, I added examples on how you can use the services described.

New Structure

I rearranged the structure of the book to make it much more usable. I found that the way the code was described was cumbersome, and I decided to completely redo it. You should notice that when I reference a specific element in a figure, I use the letter *F* followed by the figure number. The number in parenthese following the figure number represents a specific element in the figure to which I am

trying to bring your attention. **F1.2(3)** thus means "please look at the item numbered "**3**" in Figure 1.2. I used this scheme in the previous edition, but this time I decided to place these reference markers in the margin instead of burying them in the text. I find that it's a lot easier to follow the code or figure using this scheme and I hope you do too.

µC/OS-II Goals

My most important goal is to demystify real-time kernel internals. By understanding how a kernel works, you are in a better position to determine whether you need a kernel for your own products. Most of the concepts presented in this book are applicable to a large number of commercial kernels. My next most important goal is to provide you with a quality product that you can potentially use in your own products. µC/OS-II is not freeware nor is it open source code. If you use µC/OS-II in a commercial product, you need to license its use (see Appendix B, "Licensing Policy for µC/OS-II").

Intended Audience

This book is intended for embedded system programmers, consultants, and students interested in real-time operating systems. µC/OS-II is a high performance, deterministic, real-time kernel and can be (and has been) used in commercial embedded products.

Instead of writing your own kernel, you should consider µC/OS-II. You will find, as I did, that writing a kernel is not as easy as it first looks.

I'm assuming that you know C and have a minimum knowledge of assembly language. You should also understand microprocessor architectures.

What You Need to Use µC/OS-II

The code supplied with this book assumes that you are using an IBM-PC/AT or compatible (80386 minimum) computer running under DOS 4.x or higher. The code was compiled with the Borland C++ v4.51. You should have about 10 MB of free disk space on your hard drive. I actually compiled and executed the sample code provided in this book on a 300 MHz Pentium II computer running Microsoft's Windows 2000. I have successfully compiled and run the code on Windows 95, 98, and NT-based machines.

To use µC/OS-II on a different target processor (other than a PC), you need to either port µC/OS-II to that processor yourself or obtain such a port from the official µC/OS-II Web site at `http://www.uCOS-II.com`.. You also need appropriate software development tools, such as an ANSI C compiler, an assembler, linker/locator, and some way of debugging your application.

The µC/OS Story

Many years ago, I designed a product based on an Intel 80C188 at Dynalco Controls, and I needed a real-time kernel. I had been using a well-known kernel (I'll call it kernel A) in my work for a previous employer, but it was too expensive for the application I was designing. I found a lower-cost kernel ($1,000 at the time) (I'll call it kernel B) and started the design. I spent about two months trying to get a couple of very simple tasks to run. I was calling the vendor almost on a daily basis for help to make it

work. The vendor claimed that kernel B was written in C (the language); however, I had to initialize every single object using assembly language code. Although the vendor was very patient, I decided that I had had enough. The product was falling behind schedule, and I really didn't want to spend my time debugging this low-cost kernel. It turns out that I was one of the vendor's first customers, and the kernel really was not fully tested and debugged.

To get back on track, I decided to go back and use kernel A. The cost was about $5,000 for five development seats, and I had to pay a per-usage fee of about $200 for each unit that was shipped. This was a lot of money at the time, but it bought some peace of mind. I got the kernel up and running in about two days. Three months into the project, one of my engineers discovered what looked like a bug in the kernel. I sent the code to the vendor, and, sure enough, the bug was confirmed as being in the kernel. The vendor provided a 90-day warranty but that had expired, so, in order to get support, I had to pay an additional $500 per year for maintenance. I argued with the salesperson for a few months that they should fix the bug because I was actually doing them a favor. They wouldn't budge. Finally, I gave in and bought the maintenance contract, and the vendor fixed the bug six months later. Yes, six months later! I was furious and, most importantly, late delivering the product. In all, it took close to a year to get the product to work reliably with kernel A. I must admit, however, that I have had no problems with it since.

As this was going on, I naively thought that it couldn't be that difficult to write a kernel. All it needs to do is save and restore processor registers. That's when I decided to write my own kernel (part time, nights and weekends). It took me about a year to get the kernel to work as well, and, in some ways better, than kernel A. I didn't want to start a company and sell it because there were already about 50 kernels out there, so why have another one?

Then I thought of writing a paper for a magazine. First, I went to *C User's Journal* (CUJ) because the kernel was written in C. I had heard CUJ was offering $100 per published page when other magazines were only paying $75 per page. My paper had 70 or so pages, so that would be nice compensation for all the time I spent working on my kernel. Unfortunately, the article was rejected for two reasons. First, the article was too long, and the magazine didn't want to publish a series. Second, they didn't want "another kernel article."

I decided to turn to *Embedded Systems Programming* (ESP) magazine because my kernel was designed for embedded systems. I contacted the editor of ESP (Mr. Tyler Sperry) and told him that I had a kernel I wanted to publish in his magazine. I got the same response from Tyler that I did from CUJ: "Not another kernel article?" I told him that this kernel was different — it was preemptive, it was comparable to many commercial kernels, and the source code could be posted on the ESP BBS (bulletin board system). I was calling Tyler two or three times a week, basically begging him to publish my article. He finally gave in, probably because he was tired of my calls. My article was edited down from 70 pages to about 30 pages and was published in two consecutive months (May and June 1992). The article was probably the most popular article in 1992. ESP had over 500 downloads of the code from the BBS in the first month. Tyler might have feared for his life because kernel vendors were upset that he published a kernel in his magazine. I guess that these vendors must have recognized the quality and capabilities of µC/OS (called µCOS then). The article was really the first that exposed the internal workings of a real-time kernel, so some of the secrets were out.

About the time the article came out in ESP, I got a call from Dr. Bernard (Berney) Williams at CMP Books, CMP Media LLC (publisher of CUJ), six months after the initial contact with CUJ. He left a message with my wife and told her that he was interested in the article. I called him back and said, "Don't you think you are a little bit late with this? The article is being published in ESP." Berney said, "No, No, you don't understand. Because the article is so long, I want to make a book out of it." Initially, Berney simply wanted to publish what I had (as is), so the book would only have 80 pages or so. I told him that if I was going to write a book, I wanted to do it right. I then spent about six months

adding content to what is now known as the first edition. In all, the book was published at about 250 pages. I changed the name from µCOS to µC/OS because ESP readers had been calling it "mucus," which didn't sound very healthy. Come to think of it, maybe it was a kernel vendor that first came up with the name. Anyway, *µC/OS, The Real-Time Kernel* was born. Sales were somewhat slow to start. Berney and I had projected about 4,000 to 5,000 copies would be sold in the life of the book, but at the rate it was selling, I thought we'd be lucky if it sold 2,000 copies. Berney insisted that these things take time to get known, so he continued advertising in CUJ for about a year.

A month or so before the book came out, I went to my first Embedded Systems Conference (ESC) in Santa Clara, California (September 1992). I met Tyler Sperry for the first time, and I showed him a copy of the first draft of my book. He very quickly glanced at it and asked if I would like to speak at the next Embedded Systems Conference in Atlanta. Not knowing any better, I said I would and asked him what I should talk about. He suggested "Using Small Real-Time Kernels." On the trip back from California, I was thinking, "What did I get myself into? I've never spoken in front of a bunch of people before. What if I make a fool of myself? What if what I speak about is common knowledge? People pay good money to attend this conference." For the next six months, I prepared my lecture. At the conference, I had more than 70 attendees. In the first twenty minutes, I must have lost one pound of sweat. After my lecture, about 15 people or so came up to me to say that they were very pleased with the lecture and liked my book. I was invited back to the conference but could not attend the one in Santa Clara that year (1993) because my wife was due to have our second child, Sabrina. I was able to attend the next conference in Boston (1994), and I have been a regular speaker at ESC ever since. For the past several years, I've been on the conference Advisory Committee. I now do at least three lectures at every conference and each has attendance between 100 and 300 people. My lectures are almost always ranked among the top 10% at the conference.

To date, well over 25,000 copies of my µC/OS and µC/OS-II books have been sold around the world. I have received and answered thousands of e-mails from over 44 countries. I still try to answer every single one. I believe that if you take the time to write me, I owe you a response. In 1995, *µC/OS, The Real-Time Kernel* was translated into Japanese and published in Japan in a magazine called *Interface*. In 2001, µC/OS-II was translated into Chinese. A Korean translation came out in early 2002. A Japanese translation of µC/OS-II is in the works and should be available in 2002.

µC/OS and µC/OS-II have been ported to over 40 different processor architectures, and the number of ports is increasing. You should consult the µC/OS-II Web site at `http://www.uCOS-II.com` to see if the processor you intend to use is available.

In 1994, I decided to write a second book: *Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C* (ESBB). A second edition of ESBB was published in 2000. For some reason, ESBB has not been as popular as µC/OS, although it contains a lot of valuable information not found anywhere else. I always thought that it would be an ideal book for people just starting in the embedded world.

In 1998, I opened the official µC/OS Web site `http://www.uCOS-II.com`. I intend this site to contain ports, application notes, links, answers to frequently asked questions (FAQs), upgrades for µC/OS-II, and more. All I need is time!

In 2001, I started a news group to allow users to share information and their experiences with µC/OS-II.

Back in 1992, I never imagined that writing an article would change my life as it has. I met a lot of very interesting people and made a number of good friends in the process.

Thanks for choosing this book, and I hope you enjoy it!

Acknowledgments

First and foremost, I would like to thank my wife for her support, encouragement, understanding, and especially patience. Once again, I underestimated the amount of work for this edition — it was supposed to take just a few weeks and be out by January 2002. I would also like to thank my children, James (age 11) and Sabrina (age 8), for putting up with the long hours I had to spend in front of the computer.

A very special thanks to Mr. Gino Vannelli for creating such wonderful music. As far as I'm concerned, Gino redefines the word "perfection." Thanks, Gino, for being with me (in music) for almost 30 years.

I would also like to thank all the fine people at CMP Books for their help in making this book a reality and for putting up with my insistence on having things done my way.

Finally, I would like to thank all the people who have purchased my *µC/OS*, *µC/OS-II,* and *Embedded Systems Building Blocks* books over the years.

Introduction

This book describes the design and implementation of µC/OS-II (pronounced "Micro C O S 2"), which stands for *Micro-Controller Operating System, Version 2*.

µC/OS-II is a completely portable, ROMable, scalable, preemptive, real-time, multitasking kernel. µC/OS-II is written in ANSI C and contains a small portion of assembly language code to adapt it to different processor architectures. To date, µC/OS-II has been ported to over 40 different processor architectures, ranging from 8- to 64-bit CPUs.

µC/OS-II is based on *µC/OS, The Real-Time Kernel* that was first published in 1992. Thousands of people around the world are using µC/OS and µC/OS-II in all kinds of applications, such as cameras, avionics, high-end audio equipment, medical instruments, musical instruments, engine controls, network adapters, highway telephone call boxes, ATM machines, industrial robots, and more. Numerous colleges and universities have also used µC/OS and µC/OS-II to teach students about real-time systems.

µC/OS-II is upward compatible with µC/OS v1.11 (the last released version of µC/OS) but provides many improvements. If you currently have an application that runs with µC/OS, it should run virtually unchanged with µC/OS-II. All of the services (i.e., function calls) provided by µC/OS have been preserved. You may, however, have to change include files and product build files to point to the new filenames.

The companion CD for this book contains all the source code for µC/OS-II and ports for the Intel 80x86 processor running in *real mode* and for the *large model*. The code was developed and executed on a PC running Microsoft Windows 2000 but should work just as well on Windows 95, 98, Me, NT, and XP. Examples run in a DOS-compatible box under these environments. Development was done using the Borland International C/C++ compiler v4.51. Although µC/OS-II was developed and tested on a PC, µC/OS-II was actually targeted for embedded systems and can be ported easily to many different processor architectures.

µC/OS-II Features

Source Code As I mentioned previously, the companion CD contains all the source code for µC/OS-II (about 5,500 lines). I went to a lot of effort to provide you with a high-quality product. You might not agree with some of the style constructs that I use, but you should agree that the code is both clean and very consistent. Many commercial real-time kernels are provided in source form. I challenge you to find any such code that is as neat, consistent, well commented, and well organized as µC/OS-II. Also, I

believe that simply giving you the source code is not enough. You need to know how the code works and how the different pieces fit together. This book provides that type of information. The organization of a real-time kernel is not always apparent when staring at many source files and thousands of lines of code.

Portable Most of µC/OS-II is written in highly portable ANSI C, with target microprocessor-specific code written in assembly language. Assembly language is kept to a minimum to make µC/OS-II easy to port to other processors. Like µC/OS, µC/OS-II can be ported to a large number of microprocessors, as long as the microprocessor provides a stack pointer and the CPU registers can be pushed onto and popped from the stack. Also, the C compiler should provide either in-line assembly or language extensions that allow you to enable and disable interrupts from C. µC/OS-II can run on most 8-, 16-, 32-, or even 64-bit microprocessors or microcontrollers and digital signal processors (DSP).

All the ports that currently exist for µC/OS can be converted to µC/OS-II in about an hour. Also, because µC/OS-II is upward compatible with µC/OS, your µC/OS applications should run on µC/OS-II with few or no changes. Check for the availability of ports on the µC/OS-II Web site at `www.uCOS-II.com`.

ROMable µC/OS-II was designed for embedded applications, which means that if you have the proper tool chain (i.e., C compiler, assembler, and linker/locator), you can actually embed µC/OS-II as part of a product.

Scalable I designed µC/OS-II so that you can use only the services you need in your application, which means that a product can use just a few µC/OS-II services, while another product can benefit from the full set of features. Scalability allows you to reduce the amount of memory (both RAM and ROM) needed by µC/OS-II on a per-product basis. Scalability is accomplished with the use of conditional compilation. Simply specify (through `#define` constants) which features you need for your application or product. I did everything I could to reduce both the code and data space required by µC/OS-II.

Preemptive µC/OS-II is a fully preemptive real-time kernel, which means that µC/OS-II always runs the highest priority task that is ready. Most commercial kernels are preemptive, and µC/OS-II is comparable in performance with many of them.

Multitasking µC/OS-II can manage up to 64 tasks; however, I recommend that you reserve eight of these tasks for µC/OS-II, leaving your application up to 56 tasks. Each task has a unique priority assigned to it, which means that µC/OS-II cannot do round-robin scheduling. There are thus 64 priority levels.

Deterministic Execution times for most of µC/OS-II functions and services are deterministic, which means that you can always know how much time µC/OS-II will take to execute a function or a service. Except for `OSTimeTick()` and some of the event flag services, execution times of µC/OS-II services do not depend on the number of tasks running in your application.

Task Stacks Each task requires its own stack; however, µC/OS-II allows each task to have a different stack size, which allows you to reduce the amount of RAM needed in your application. With µC/OS-II's stack-checking feature, you can determine exactly how much stack space each task actually requires.

Services µC/OS-II provides a number of system services, such as semaphores, mutual exclusion semaphores, event flags, message mailboxes, message queues, fixed-sized memory partitions, task management, time management functions, and more.

Interrupt Management Interrupts can suspend the execution of a task. If a higher priority task is awakened as a result of the interrupt, the highest priority task runs as soon as all nested interrupts complete. Interrupts can be nested up to 255 levels deep.

Robust and Reliable µC/OS-II is based on µC/OS, which has been used in hundreds of commercial applications since 1992. µC/OS-II uses the same core and most of the same functions as µC/OS, yet offers many more features. Also, in July of 2000, µC/OS-II was certified in an avionics product by the Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of the RTCA DO-178B standard for software used in avionics equipment. In order to meet the requirements of this standard, it must be possible to demonstrate through documentation and testing that the software is both robust and safe. This issue is particularly important for an operating system as it demonstates that it has the proven quality to be usable in any application. Every feature, function, and line of code of µC/OS-II has been examined and tested to demonstrate that it is safe and robust enough to be used in safety-critical systems where human life is on the line.

Figures, Listings, and Tables

You will notice that when I reference a specific element in a figure, I use the letter "F" followed by the figure number. The number in parenthesis following the figure number represents a specific element in the figure that I am trying to bring your attention to. **F1.2(3)** thus means "please look at the item numbered "**3**" in Figure **1.2**".

Chapter Contents

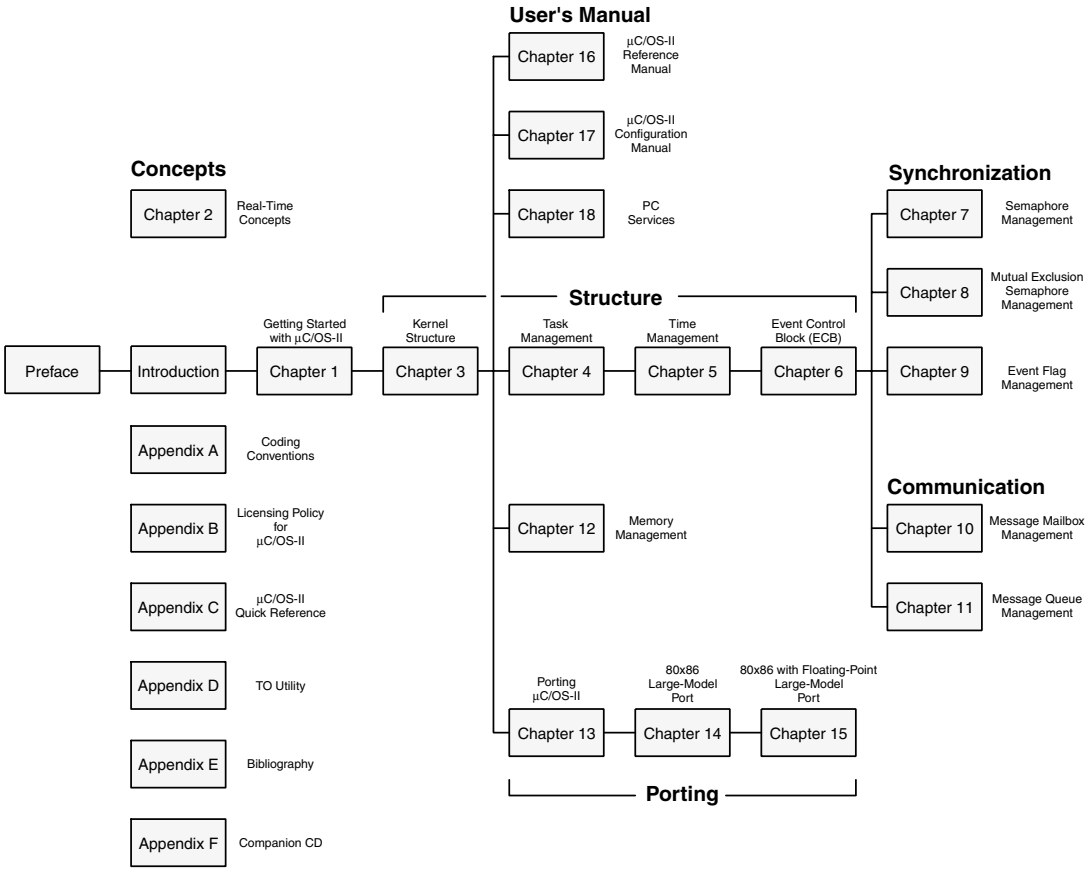
Figure I.1 shows the layout and the flow of this book. I thought this diagram would be useful to understand the relationship between the chapters. Chapter 2 is a standalone chapter and doesn't depend on any other chapter. As a minimum, I recommend that you read the Preface, the Introduction, Chapter 1 and Chapter 3. Then with the knowledge you will have gained about µC/OS-II, you ought to be able to start using µC/OS-II and thus move to Chapters 16 and 17 to understand what features are available. If you want to further your understanding of µC/OS-II, you can proceed with Chapters 4, 5, and 6. After you understand Chapter 6, you can either jump to the synchronization or communication services.

Chapter 1, Getting Started with µC/OS-II This chapter is designed to allow you to experiment with µC/OS-II immediately. In fact, I assume you know little about µC/OS-II and multitasking; concepts are introduced as needed. This chapter has been completely re-written from the previous edition.

Chapter 2, Real-time Systems Concepts Here, I introduce you to some real-time systems concepts, such as foreground/background systems, critical sections, resources, multitasking, context switching, scheduling, reentrancy, task priorities, mutual exclusion, semaphores, intertask communications, interrupts, and more.

Chapter 3, Kernel Structure This chapter introduces you to µC/OS-II and its internal structure. You will learn about tasks, task states, and task control blocks; how µC/OS-II implements a ready list, task scheduling, and the idle task; how to determine CPU usage; how µC/OS-II handles interrupts; how to initialize and start µC/OS-II; and more.

Figure I.1 Book layout and flow.



Chapter 4, Task Management This chapter describes µC/OS-II services that create a task, delete a task, check the size of a task's stack, change a task's priority, suspend and resume a task, and get information about a task.

Chapter 5, Time Management This chapter describes how µC/OS-II can suspend a task's execution until some user-specified time expires, how such a task can be resumed, and how to get and set the current value of a 32-bit tick counter.

Chapter 6, Event Control Blocks This chapter describes a data structure that is used by most of the kernel objects to do synchronization and communication. This data structure allows tasks and Interrupt Service Routines (ISR) to communicate with one another and share resources. This chapter is a prerequisite to Chapters 7 through 11.

Chapter 7, Semaphore Management A semaphore is a kernel object that your tasks needs to acquire in order to gain exclusive access to shared resources. This chapter describes how semaphores are implemented in µC/OS-II.

Chapter 8, Mutual Exclusion Semaphores A mutual exclusion semaphores (mutex) is a binary semaphore that allows you to gain exclusive access to a resource. The mutex reduces priority inversion issues by automatically changing a task's priority if needed. This chapter describes how (mutex) are implemented in µC/OS-II. Mutexes are new services in this edition.

Chapter 9, Event Flag Management Event flags are bits for which a task can wait. A task can wait for one or more of these bits to be set or cleared. This chapter shows how event flags are implemented and describes the services that are available to your application. Event flags are new services in this edition.

Chapter 10, Message Mailbox Management A message mailbox allows your tasks to send messages to one another. This chapter shows how these services are implemented.

Chapter 11, Message Queue Management A message queue is like a message mailbox, except that it allows multiple messages to be sent to one or more tasks. This chapter shows how message queues are implemented.

Chapter 12, Memory Management This chapter describes the µC/OS-II dynamic memory allocation feature using fixed-sized memory blocks.

Chapter 13, Porting µC/OS-II This chapter describes in general terms what needs to be done to adapt µC/OS-II to different processor architectures. This chapter has been completely rewritten from the previous edition.

Chapter 14, 80x86 Port Real Mode, Large Model with Emulated Floating-Point Support This chapter describes how µC/OS-II was ported to the Intel/AMD 80x86 processor architecture running in real mode and for the large-memory model.

Chapter 15, 80x86 Port Real Mode, Large Model with Hardware Floating-Point Support This chapter is an extension of the previous one, except that it shows how you can add the floating-point registers of the 80486, 5x86, and Pentium processors to the context switch. This chapter is new to this edition.

Chapter 16, µC/OS-II Reference Manual This chapter describes each of the functions (i.e., services) provided by µC/OS-II from an application developer's standpoint. Each function contains a brief description, its prototype, the name of the file where the function is found, a description of the function arguments and the return value, special notes, and examples. Many new services have been added in this edition (mutexes and event flags), and these have been added in this chapter.

Chapter 17, µC/OS-II Configuration Manual This chapter describes each of the `#define` constants used to configure µC/OS-II for your application. Configuring µC/OS-II allows you to use only the services required by your application. This gives you the flexibility to reduce the µC/OS-II memory footprint (code and data space). This new edition contains more than three times as many configuration options to allow you to reduce the amount of code and data space needed by µC/OS-II.

Chapter 18, PC Services The examples of Chapter 1 assume the use of a IBM/PC compatible computer. This new chapter shows how I encapsulated some of the services available from a PC.

Appendix A, C Coding Conventions This appendix shows the coding conventions that I used in this book and in my everyday activities.

Appendix B, Licensing Policy for µC/OS-II This appendix describes the licensing policy for distributing µC/OS-II in source and object form.

Appendix C, µC/OS-II Quick Reference This appendix provides a quick reference to µC/OS-II's services.

Appendix D, `TO` Utility `TO` is a DOS utility that allows you to navigate between DOS directories without having to type long `CD path` commands.

Appendix E, Bibliography This appendix provides a bibliography of reference material that you might find useful if you are interested in getting further information about embedded real-time systems.

Appendix F, Companion CD This appendix tells you how to install µC/OS-II and describes what's on the companion CD.

µC/OS-II Web Site

To provide better support to you, I created the µC/OS-II Web site (`http://www.uCOS-II.com`). You can obtain information about

- news on µC/OS and µC/OS-II,
- upgrades,
- bug fixes,
- availability of ports,
- answers to frequently asked questions (FAQs),
- application notes,
- books,
- classes,
- links to other Web sites, and more.

Chapter 1

1

Getting Started with µC/OS-II

This chapter provides four examples on how to use µC/OS-II. I decided to include this chapter early in the book so you could start using µC/OS-II as soon as possible. In fact, I assume you know little about µC/OS-II and multitasking; concepts are introduced as needed.

The sample code was compiled using the Borland C/C++ compiler v4.51, and options were selected to generate code for an Intel/AMD 80186 processor (large-memory model). The code was actually run and tested on a 300MHz Intel Pentium II PC, running in a DOS window using Microsoft Windows 2000. For all intents and purposes, a Pentium can be viewed as a superfast 80186 processor. The Borland C/C++ v4.51 (called the *Borland Turbo C++ 4.5*) is available from `www.Borland.com`, and I was assured by Borland that readers would still be able to purchase this compiler for a number of years to come.

I chose a PC as my target system for a number of reasons. First and foremost, it's a lot easier to test code on a PC than on any other embedded environment (i.e., evaluation board or emulator): there are no EPROMs or Flash to burn and no downloads to EPROM emulators, or CPU emulators. You simply compile, link, and run. Second, the 80186 object code (real mode, large model) generated using the Borland C/C++ compiler is compatible with all 80x86 derivative processors from Intel, AMD, and others.

1.00 Installing µC/OS-II

This book includes a companion CD, and you should refer to Appendix F for instruction on how to install the source of µC/OS-II and executables of the examples on your computer. The installation assumes that you are installing the software on a Windows 95, 98, Me, NT, 2000, or XP computer.

1.01 Example #1

Example #1 demonstrates basic multitasking capabilities of µC/OS-II. Ten tasks display a number between 0 and 9 at random locations on the screen. Each task displays only one of the number. In other words, one task displays 0 at random locations, another task displays 1, and so on.

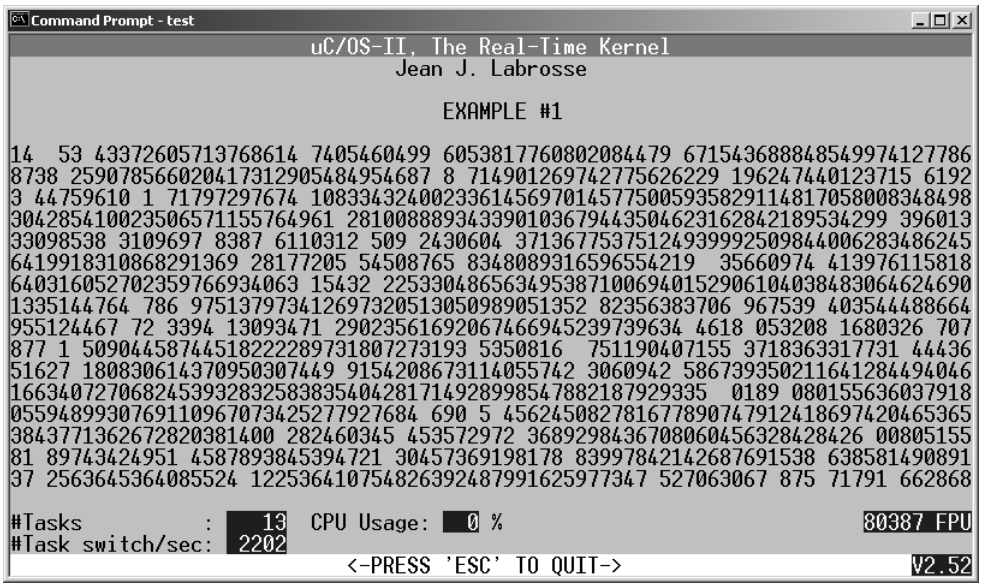
The code for Example #1 is found in the `\SOFTWARE\uCOS-II\EX1_x86L\BC45` directory of the installation drive (the default is C:). You can open a DOS window (called Command Prompt in Microsoft Windows 2000) and type

```
CD \SOFTWARE\uCOS-II\Ex1_x86L\BC45\TEST
```

The CD command allows you to change directory and, in this case, go to the `TEST` directory of Example #1. The `TEST` directory contains four files: `MAKETEST.BAT`, `TEST.EXE`, `TEST.LNK`, and `TEST.MAK`. To execute Example #1, simply type `TEST` at the command line prompt. The DOS window runs the `TEST.EXE` program.

After about one second, you should see the DOS window randomly fill up with numbers between 0 and 9, as shown in Figure 1.1.

Figure 1.1 Example #1 running in a DOS window.



Example #1 consists of 13 tasks, as displayed in the lower left of Figure 1.1. µC/OS-II creates two internal tasks: the idle task and a task that determines CPU usage. The code in Example #1 creates the other 11 tasks.

The source code for Example #1 is found in `TEST.C`, in the `SOURCE` directory. You can get there from the `TEST` directory by typing

```
CD  ..\SOURCE
```

Portions of `TEST.C` are shown in Listing 1.1. You can examine the actual code using your favorite code editor.

Listing 1.1 Example #1, TEST.C.

```
#include  "includes.h"                                                    (1)

#define       TASK_STK_SIZE            512                              (2)
#define       N_TASKS                   10


OS_STK        TaskStk[N_TASKS][TASK_STK_SIZE];                           (3)
OS_STK        TaskStartStk[TASK_STK_SIZE];                               (4)
char          TaskData[N_TASKS];                                         (5)
OS_EVENT     *RandomSem;                                                 (6)
```

Note: To describe listings and figures, I place a reference in the margin. The reference corresponds to an element of the listing or figure to which I want to bring your attention. For example, L1.1(1) means: "please refer to Listing 1.1 and locate the item (1)." This notation also applies to figures and thus F3.1(2) means: "please look at Figure 3.1 and examine item (2)."

L1.1(1) First, you notice that there is only a single `#include` statement. That's because I like to place all my header files in a master header file called `INCLUDES.H`. Each source file always references this single `include` file, and thus I never need to worry about determining which headers I need; they all get included via `INCLUDES.H`. You can use your code editor to view the contents of `INCLUDES.H`, which is also found in the `SOURCE` directory.

µC/OS-II is a multitasking kernel and allows you to have up to 63 application tasks. µC/OS-II decides when to switch from one task to an other, based on information you provide to µC/OS-II. One of the items you must tell µC/OS-II is the priority of your tasks. Changing between tasks is called a *context switch*.

I will return to Listing 1.1 later as needed. Like most C programs, we need a `main()`, as shown in Listing 1.2.

Listing 1.2 Example #1, TEST.C, main().

```
void  main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);                     (1)

    OSInit();                                                             (2)

    PC_DOSSaveReturn();                                                   (3)
    PC_VectSet(uCOS, OSCtxSw);                                            (4)

    RandomSem   = OSSemCreate(1);                                         (5)
```

Listing 1.2 Example #1, TEST.C, main(). (Continued)

```
    OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 0); (6)

    OSStart();                                                               (7)
}
```

L1.2(1) `main()` starts by clearing the screen to ensure that no characters are left over from the previous DOS session. The function `PC_DispClrScr()` is found in a file called `PC.C` (see Chapter 18, "PC Services" for details). `PC.C` contains functions that provide services if you are running in a DOS environment (or a window under the Microsoft Windows 95, 98, Me, NT, 2000, or XP operating systems). The `PC_` prefix allows you to easily determine the name of the file from which the function comes; in this case, `PC.C`. You should note that I specified white letters on a black background. Because the screen will be cleared, I simply could have specified a black background and not specified a foreground. If I did this, and you decided to return to the DOS prompt, you would not see anything on the screen! It's always better to specify a visible foreground just for this reason.

L1.2(2) A requirement of µC/OS-II is that you call `OSInit()` before you invoke any of its other services. `OSInit()` creates two tasks: an idle task, which executes when no other task is ready to run, and a statistic task, which computes CPU usage.

L1.2(3) The current DOS environment is saved by calling `PC_DOSSaveReturn()`, which allows you to return to DOS as if you had never started µC/OS-II. You can refer to Chapter 18, "PC Services" for a description of what `PC_DOSSaveReturn()` does.

L1.2(4) `main()` calls `PC_VectSet()` (see Chapter 18, "PC Services") to install the µC/OS-II context-switch handler. Task-level context switching is done by µC/OS-II by issuing an 80x86 `INT` instruction to this vector location. I decided to use vector `0x80` (i.e., `128`) because it's not used by either DOS or the BIOS.

L1.2(5) A binary semaphore is created to guard access to the random-number generator function provided by the Borland C/C++ library. A semaphore is an object provided by the kernel to prevent multiple tasks from accessing the same resource (in this case a function) at the same time. I decided to use a semaphore because I didn't know whether or not the random-generator function was reentrant; I assumed it was not. By initializing the semaphore to 1, I'm telling µC/OS-II to allow only one task to access the random-generator function at any given time. A semaphore must be created before it can be used, which is done by calling `OSSemCreate()` and specifying its initial value. `OSSemCreate()` returns a handle [see Listing 1.1(6)] to the semaphore, which must be used to reference this particular semaphore.

L1.2(6) Before starting multitasking, you have to create at least one task. For this example, I called this task `TaskStart()`. You create a task because you want to tell µC/OS-II to manage the task. The `OSTaskCreate()` function receives four arguments. The first argument is a pointer to the task's address, in this case `TaskStart()`. The second argument is a pointer to data that you want to pass to the task when it first starts. In this case, there is nothing to pass, and thus I passed a `NULL` pointer. It could, however, have been anything. I'll discuss the use of this argument in Example #4. The third argument is the task's top-of-stack (TOS). With µC/OS-II, as with most preemptive kernels, each task requires its own stack space. Each task in µC/OS-II can have a different size, but, for simplicity, I made them all the same. On the 80x86 CPU, the stack grows downwards, and thus we must pass the highest, most valid TOS

address to `OSTaskCreate()`. In this case, the stack is called `TaskStartStk[]` and is allocated at compile time. A stack must be declared having a type `OS_STK` [see Listing 1.1(4)]. The size of the stack is declared in Listing 1.1(2). For the 80x86, an `OS_STK` is a 16-bit value, and thus the size of the stack is 1024 bytes. Finally, we must specify the priority of the task being created. The lower the priority number, the higher the priority (i.e., its importance).

As previously mentioned, µC/OS-II allows you to create up to 63 tasks. However, each task must have a unique priority number between 0 and 62. You're the one that actually decides what priority to give your tasks, based on your application requirements. Priority level 0 is the highest priority.

L1.2(7) `OSStart()` is then called to start multitasking and give control to µC/OS-II. It is very important that you create at least one task before calling `OSStart()`. Failure to do this action will certainly make your application crash. In fact, you might always want to create only one task if you are planning on using the CPU usage statistic task.

`OSStart()`'s job is to determine which, of all the tasks created, is the most important one (highest priority) and start executing this task. In our case, µC/OS-II created two low priority tasks: the idle task and the statistic task. `main()` created `TaskStart()` with a priority of 0. As I mentioned, priority 0 is the highest priority, and thus `OSStart()` starts executing `TaskStart()`.

You should note that `OSStart()` doesn't return to `main()`. However, if you call `PC_DOSReturn()`, multitasking is halted, and your application returns to DOS (but not `main()`). In an embedded system, there is no need for an equivalent function to `PC_DOSReturn()` because you would most likely not be returning to anything!

As I mentioned in the previous section, `OSStart()` selects `TaskStart()` as the most important task to run first. `TaskStart()` is shown in Listing 1.3.

Listing 1.3 Example #1, TEST.C, TaskStart().

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    char       s[100];
    INT16S     key;


    pdata = pdata;                                                    (1)

    TaskStartDispInit();                                              (2)

    OS_ENTER_CRITICAL();                                              (3)
    PC_VectSet(0x08, OSTickISR);                                      (4)
    PC_SetTickRate(OS_TICKS_PER_SEC);                                 (5)
    OS_EXIT_CRITICAL();                                               (6)
```

Listing 1.3 Example #1, TEST.C, TaskStart(). (Continued)

```
    OSStatInit();                                                    (7)

    TaskStartCreateTasks();                                          (8)

    for (;;) {                                                       (9)
        TaskStartDisp();                                            (10)

        if (PC_GetKey(&key) == TRUE) {                              (11)
            if (key == 0x1B) {                                      (12)
                PC_DOSReturn();                                     (13)
            }
        }

        OSCtxSwCtr = 0;                                             (14)
        OSTimeDlyHMSM(0, 0, 1, 0);                                  (15)
    }
}
```

L1.3(1) `TaskStart()` begins by setting `pdata` to itself. I do this because some compilers complain (error or warning) if `pdata` is not referenced. In other words, I fake the usage of `pdata`! `pdata` is a pointer passed to your task when the task is created. The second argument passed in `OSTaskCreate()` is none other than the argument `pdata` of a task [see L1.2(6)]. Because I passed a `NULL` pointer [again see L1.2(6)], I am not passing anything to `TaskStart()`.

L1.3(2) `TaskStart()` then calls `TaskStartDispInit()` to initialize the display, as shown in Figure 1.2. `TaskStartDispInit()` makes 25 consecutive calls to `PC_DispStr()` (see Chapter 18, "PC Services") to fill the 25 lines of text of a typical DOS window.

L1.3(3) `TaskStart()` then invokes the macro `OS_ENTER_CRITICAL()`. `OS_ENTER_CRITICAL()` is basically a processor-specific macro, and it's used to disable interrupts (see Chapter 13, Porting µC/OS-II).

L1.3(4) µC/OS-II, like all kernels, requires a time source to keep track of delays and timeouts. In real mode, the PC offers such a time source, which occurs every 54.925ms (18.20648Hz) and is called a tick. `PC_VectSet()` allows us to replace the address where the PC goes to service the DOS tick with one that is used by µC/OS-II. However, µC/OS-II still calls the DOS tick handler every 54.925ms. This technique is called *chaining* and is set up by `PC_DOSSaveReturn()` (see Chapter 18, "PC Services").

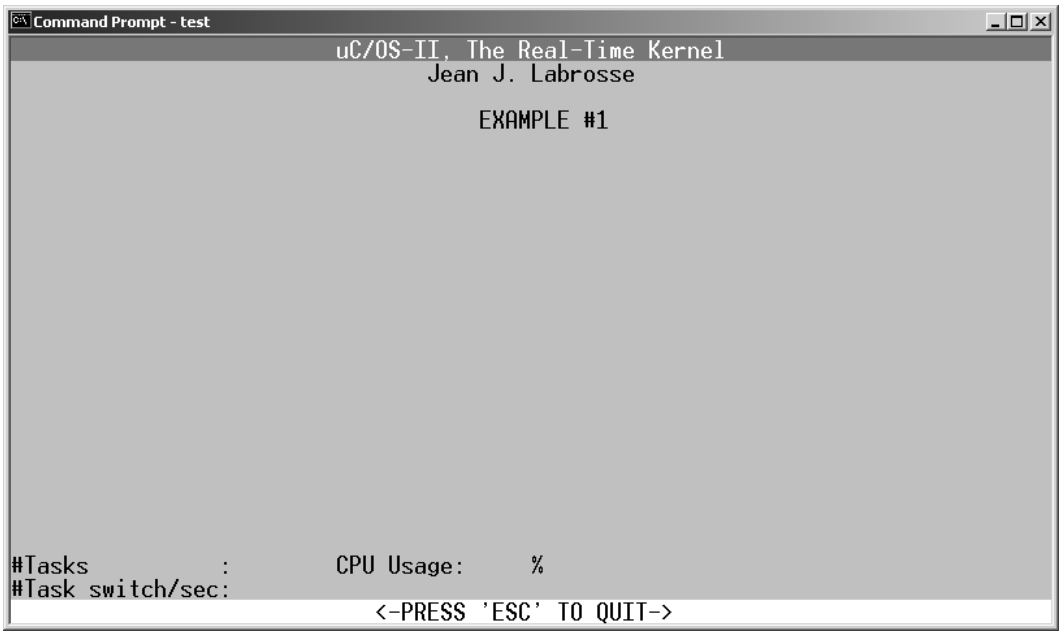
L1.3(5) We then change the tick rate from 18.2Hz to 200Hz. I selected 200Hz because it's almost an exact multiple of 18.2Hz (i.e., 11 times faster). I never quite understood why IBM selected 18.2Hz instead of 20Hz as the tick rate on the original PC. Instead of setting up the 82C54 timer to divide the timer input frequency by 59,659 to obtain a nice 20Hz, it appears that they left the 16-bit timer to overflow every 65,536 pulses! Changing the tick rate is handled by another PC service called `PC_SetTickRate()`, which is passed the desired tick rate (`OS_TICKS_PER_SEC` is set to 200 in `OS_CPU.H`).

L1.3(6) We then invoke the macro `OS_EXIT_CRITICAL()`. `OS_EXIT_CRITICAL()` is a processor-specific macro and is used to reenable interrupts (see Chapter 13, "Porting µC/OS-II"). `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` must be used in pairs.

L1.3(7) `OSStatInit()` is called to determine the speed of your CPU (see Chapter 3, "Getting Started with µC/OS-II"). This function allows µC/OS-II to know what percentage of the CPU is actually being used by all the tasks.

L1.3(8) `TaskStart()` then calls `TaskStartCreateTasks()` to let µC/OS-II manage more tasks. Specifically, we are adding `N_TASKS` identical tasks [see Listing 1.1(2)]. `TaskStartCreateTasks()` is shown in Listing 1.4.

Figure 1.2 Initialization of the display by `TaskStartDispInit()`.



Listing 1.4 Example #1, `TEST.C`, `TaskStartCreateTasks()`.

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;


    for (i = 0; i < N_TASKS; i++) {
        TaskData[i] = '0' + i;                                    (1)
        OSTaskCreate(Task,                                        (2)
                     (void *)&TaskData[i],                        (3)
```

Listing 1.4 Example #1, TEST.C, TaskStartCreateTasks(). (Continued)

```
                    &TaskStk[i][TASK_STK_SIZE - 1],                    (4)
                    i + 1);                                            (5)
    }
}
```

L1.4(1) An array is initialized to contain the ASCII characters 0 to 9 [see also Listing 1.1(5)].

L1.4(2) The loop initializes `N_TASKS` identical tasks called `Task()`. `Task()` is responsible for placing an ASCII character at a random location on the screen. In fact, each instance of `Task()` places a different character.

L1.4(3) Each of these task receive a pointer to the array of ASCII characters. Each task in fact receives a pointer to a different character.

L1.4(4) Again, each task requires its own stack space [see Listing 1.1(3)].

L1.4(5) With µC/OS-II, each task must have a unique priority. Because priority number 0 is already used by `TaskStart()`, I decided to create tasks with priorities 1 through 10.

As each task is created, µC/OS-II determines whether the created task is more important than the creator. If the created task had a higher priority, then µC/OS-II would immediately run the created task. However, because `TaskStart()` has the highest priority (priority 0), none of the created tasks execute just yet.

We can now resume discussion of Listing 1.3.

L1.3(9) With µC/OS-II, each task must be an infinite loop.

L1.3(10) `TaskStartDisp()` is called to display information at the bottom of the DOS window (see Figure 1.1). Specifically, `TaskStartDisp()` prints the number of tasks created, the current CPU usage in percentage, the number of context switches, the version of µC/OS-II, and, finally, whether your processor has a floating-point unit (FPU) or not.

L1.3(11) `TaskStart()` then checks to see if you pressed a key by calling `PC_GetKey()`.

L1.3(12)

L1.3(13) `TaskStart()` determines whether you pressed the Esc key on your keyboard and, if so, calls `PC_DOSReturn()` to exit this example and return to the DOS prompt. You can find out how this action is done by referring to Chapter 18, "PC Services."

L1.3(14) If you didn't press the Esc key, the global variable `OSCtxSwCtr` (the context-switch counter) is cleared so that we can display the number of context switches in one second.

L1.3(15) Finally, `TaskStart()` is suspended (does not run) for one complete second by calling `OSTimeDlyHMSM()`. The `HMSM` stands for hours, minutes, seconds, and milliseconds and corresponds to the arguments passed to `OSTimeDlyHMSM()`. Because `TaskStart()` is suspended for one second, µC/OS-II starts executing the next most important task, in this case `Task()` at priority 1. You should note that without `OSTimeDlyHMSM()` (or other similar functions), `TaskStart()` would be a true infinite loop, and other tasks would never get a chance to run.

The code for `Task()` is shown in Listing 1.5.

L1.5(1) As I previously mentioned, a µC/OS-II task is typically an infinite loop.

Listing 1.5 Example #1, TEST.C, Task().

```
void  Task (void *pdata)
{
    INT8U  x;
    INT8U  y;
    INT8U  err;

    for (;;) {                                                          (1)
        OSSemPend(RandomSem, 0, &err);                                  (2)
        x = random(80);                                                 (3)
        y = random(16);                                                 (4)
        OSSemPost(RandomSem);                                           (5)

        PC_DispChar(x, y + 5, *(char *)pdata, DISP_FGND_LIGHT_GRAY);    (6)
        OSTimeDly(1);                                                   (7)
    }
}
```

L1.5(2) The task starts by acquiring the semaphore, which guards access to the Borland compiler random-number-generator function. To call the semaphore, call `OSSemPend()` and pass it the handle [see L1.1(6)] of the semaphore, which was created to guard access to the random-number-generator function. The second argument of `OSSemPend()` is used to specify a time-out. A value of 0 means that this task will wait forever for the semaphore. Because the semaphore was initialized with a count of one and no other task has requested the semaphore, `Task()` is allowed to continue execution. If the semaphore was owned by another task, µC/OS-II would have suspended this task and executed the next most important task.

L1.5(3) The random-number-generator function is called and a value between 0 and 79 (inclusively) is returned. This value happens to be the x-coordinate where we want to display the character 0 (for this task) on the screen.

L1.5(4) Again, the random-number-generator is called, and returns a number between 0 and 15 (inclusively). This value is used to determine the y-coordinate of the character to display.

L1.5(5) The semaphore is released by calling `OSSemPost()`. Here we simply need to specify the semaphore handle.

L1.5(6) We can now display the character that was passed to `Task()` when `Task()` was created. For the first instance of `Task()`, the character is 0, and is the last instance, it's 9. I added an offset of five lines from the top so that I don't overwrite the header at the top of the display (see Figure 1.1).

L1.5(7) Finally, `Task()` calls `OSTimeDly()` to tell µC/OS-II that it's done executing and to give other tasks a chance to run. The value of 1 means that I want this task to delay for one clock tick, or 5ms because the tick rate is 200Hz. When `OSTimeDly()` is called, µC/OS-II suspends the calling function and executes the next most important task. In this case, it is another instance of `Task()`, which displays 1. This process goes on for all instances of `Task()`, and thus that's why Figure 1.1 looks the way it does.

If you have the Borland C/C++ v4.5x compiler installed in the C:\BC45 directory, you can experiment with TEST.C. After modifying TEST.C, you can type MAKETEST from the command prompt of the TEST directory to build a new TEST.EXE. If you don't have the Borland C/C++ v4.5x compiler or you have it installed in a different directory, you can make the appropriate changes to TEST.MAK, INCLUDES.H, and TEST.LNK.

The SOURCE directory contains four files: INCLUDES.H, OS_CFG.H, TEST.C, and TEST.LNK. OS_CFG.H is used to determine µC/OS-II configuration options. TEST.LNK is the linker-command file for the Borland linker, TLINK.

1.02 Example #2

Example #2 demonstrates the stack-checking feature of µC/OS-II. The amount of stack space used by each task is displayed along with the amount of free stack space. Also, Example #2 shows the execution time of the stack-checking function OSTaskStkChk() because it depends on the size of each stack. It turns out that a heavily used stack requires less processing time.

The code for Example #2 is found in the \SOFTWARE\uCOS-II\EX2_x86L\BC45 directory. You can open a DOS window and type

```
CD  \SOFTWARE\uCOS-II\Ex2_x86L\BC45\TEST
```

To execute Example #2, type TEST at the command prompt. The DOS window runs the TEST.EXE program.

After about one second, you should see the screen shown in Figure 1.3.

Example #2 consists of nine tasks, as displayed in the lower left of Figure 1.3. Of those nine tasks, µC/OS-II creates two internal tasks: the idle task and a task that determines CPU usage. Example #2 creates the other seven tasks.

Example #2 shows you how you can display task statistics beyond the number of tasks created, the number of context switches, and the CPU usage. Specifically, Example #2 shows you how you can find out how much stack space each task is actually using and how much execution time it takes to determine the size of each task stack.

Example #2 makes use of the extended task-create function (OSTaskCreateExt()) and the µC/OS-II stack-checking feature [OSTaskStkChk()]. Stack checking is useful when you don't actually know ahead of time how much stack space you need to allocate for each task. In this case, you allocate much more stack space than you think you need and let µC/OS-II tell you exactly how much stack space is actually used. You obviously need to run the application long enough and under your worst case conditions to get valid numbers. Your final stack size should accommodate system expansion, so make sure you allocate between 10–25% more. In safety-critical applications, however, you might even want to consider 100% more! What you get from stack checking is a ballpark figure; you are not looking for an exact stack usage.

The µC/OS-II stack-checking function fills the stack of a task with zeros when the task is created. You accomplish this by telling OSTaskCreateExt() that you want to clear the stack upon task creation and that you want to check the stack (i.e., by setting the OS_TASK_OPT_STK_CLR and OS_TASK_OPT_STK_CHK for the opt argument). If you intend to create and delete tasks, you should set these options so that a new stack is cleared every time the task is created. You should note that having OSTaskCreateExt() clear the stack increases execution overhead, which obviously depends on the stack size.

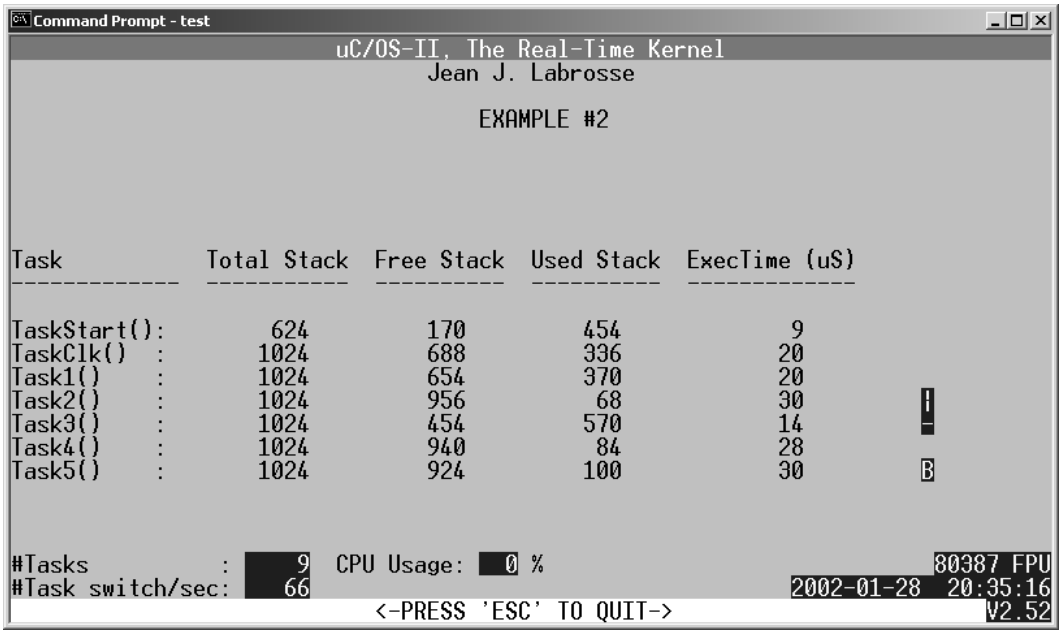
µC/OS-II scans the stack, starting at the bottom until it finds a nonzero entry. As the stack is scanned, µC/OS-II increments a counter that indicates how many entries are free.

The source code for Example #2 is found in TEST.C, in the SOURCE directory. To get there from the TEST directory, type

```
CD  ..\SOURCE
```

Portions of `TEST.C` are shown in Listing 1.6. You can examine the actual code using your favorite code editor.

Figure 1.3 Example #2 running in a DOS window.



Listing 1.6 Example #2, `TEST.C`.

```
#include "includes.h"                                                    (1)

#define          TASK_STK_SIZE     512                                   (2)

#define          TASK_START_ID       0                                   (3)
#define          TASK_CLK_ID         1
#define          TASK_1_ID           2
#define          TASK_2_ID           3
#define          TASK_3_ID           4
#define          TASK_4_ID           5
#define          TASK_5_ID           6

#define          TASK_START_PRIO    10                                   (4)
#define          TASK_CLK_PRIO      11
#define          TASK_1_PRIO        12
#define          TASK_2_PRIO        13
#define          TASK_3_PRIO        14
```

Listing 1.6 Example #2, TEST.C. (Continued)

```
#define          TASK_4_PRIO        15
#define          TASK_5_PRIO        16

OS_STK        TaskStartStk[TASK_STK_SIZE];                                  (5)
OS_STK        TaskClkStk[TASK_STK_SIZE];
OS_STK        Task1Stk[TASK_STK_SIZE];
OS_STK        Task2Stk[TASK_STK_SIZE];
OS_STK        Task3Stk[TASK_STK_SIZE];
OS_STK        Task4Stk[TASK_STK_SIZE];
OS_STK        Task5Stk[TASK_STK_SIZE];

OS_EVENT     *AckMbox;                                                      (6)
OS_EVENT     *TxMbox;
```

Based on what you learned in Example #1, you should recognize:

L1.6(1) `INCLUDES.H` as the master include file.

L1.6(2) The size of each task's stack (`TASK_STK_SIZE`). Again, I made all stack sizes the same for simplicity, but, with µC/OS-II, the stack size for each task can be different.

L1.6(5) The storage for the task stacks.

`main()` for Example #2 is shown in Listing 1.7 and looks very similar to the `main()` of Example #1. I only describe the differences.

Listing 1.7 Example #2, TEST.C, main().

```
void main (void)
{
    OS_STK *ptos;
    OS_STK *pbos;
    INT32U  size;

    PC_DispClrScr(DISP_FGND_WHITE);

    OSInit();

    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);

    PC_ElapsedInit();                                                      (1)

    ptos        = &TaskStartStk[TASK_STK_SIZE - 1];                        (2)
```

Listing 1.7 Example #2, TEST.C, main(). (Continued)

```
    pbos        = &TaskStartStk[0];
    size        = TASK_STK_SIZE;
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);                          (3)
    OSTaskCreateExt(TaskStart,                                           (4)
                    (void *)0,
                    ptos,                                                (5)
                    TASK_START_PRIO,                                     (6)
                    TASK_START_ID,                                       (7)
                    pbos,                                                (8)
                    size,                                                (9)
                    (void *)0,                                          (10)
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);         (11)

    OSStart();
}
```

L1.7(1) `main()` calls `PC_ElapsedInit()` to initialize the elapsed-time-measurement function that is used to measure the execution time of `OSTaskStkChk()`. This function basically measures the execution time (i.e., overhead) of two functions: `PC_ElapsedStart()` and `PC_ElapsedStop()`. By measuring this time, we can determine fairly precisely how long it takes to execute code that's wrapped between these two calls.

L1.7(2)

L1.7(3) `TaskStart()` in Example #2 is invoking the floating-point emulation library instead of making use of the floating-point unit (FPU), which is present on 80486 and higher-end PCs. The Borland compiler defaults to use its emulation library if an FPU is not detected. In other words, if you were to run `TEST.EXE` on a DOS-based machine equiped with an Intel 80386EX (without an 80387 coprocessor), then the floating-point unit would be emulated. The emulation library is unfortunately non-reentrant, and we have to trick it in order to allow multiple tasks to do floating-point math. For now, let me just say that we have to modify the task stack to accommodate the floating-point emulation library. This modification is accomplished by calling `OSTaskStkInit_FPE_x86()` (see Chapter 14, "80x86 Port"). You should notice from Figure 1.3 that the stack size reported for `TaskStart()` is 624 instead of 1024. That's because `OSTaskStkInit_FPE_x86()` reserves the difference for the floating-point emulation library.

L1.7(4) Instead of calling `OSTaskCreate()` to create `TaskStart()`, we must call `OSTaskCreateExt()` [the extended version of `OSTaskCreate()`] because we modified the stack and also because we want to check the stack size at run time (described later).

L1.7(5) `OSTaskStkInit_FPE_x86()` modifies the top-of-stack pointer, so we must pass the new pointer to `OSTaskCreateExt()`.

L1.7(6) Instead of passing a hard-coded priority (as I did in Example #1), I created a `#define` symbol [see L1.6(4)].

L1.7(7) `OSTaskCreateExt()` requires that you pass a task identifier (ID). The actual value can be anything because this field is not actually used by µC/OS-II at this time.

L1.7(8) `OSTaskStkInit_FPE_x86()` modifies the bottom-of-stack pointer, so we must pass the new pointer to `OSTaskCreateExt()`.

L1.7(9) `OSTaskStkInit_FPE_x86()` also modifies the size of the stack, so we must pass the new size to `OSTaskCreateExt()`.

L1.7(10) One of `OSTaskCreateExt()`'s arguments is a task-control-block (TCB) extension pointer. This argument is not used in Example #2, so we simply pass a `NULL` pointer.

L1.7(11) Finally, the last argument to `OSTaskCreateExt()` is a set of options (i.e., bits) that tell `OSTaskCreateExt()` that we are doing stack-size checking and that we want to clear the stack when the task is created.

`TaskStart()` is similar to the one described in Example #1 and is shown in Listing 1.8. Again, I only describe the differences.

Listing 1.8 Example #2, TEST.C, TaskStart().

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16S     key;

    pdata = pdata;

    TaskStartDispInit();                                                    (1)

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();

    OSStatInit();

    AckMbox = OSMboxCreate((void *)0);                                      (2)
    TxMbox  = OSMboxCreate((void *)0);

    TaskStartCreateTasks();                                                 (3)

    for (;;) {
        TaskStartDisp();

        if (PC_GetKey(&key)) {
```

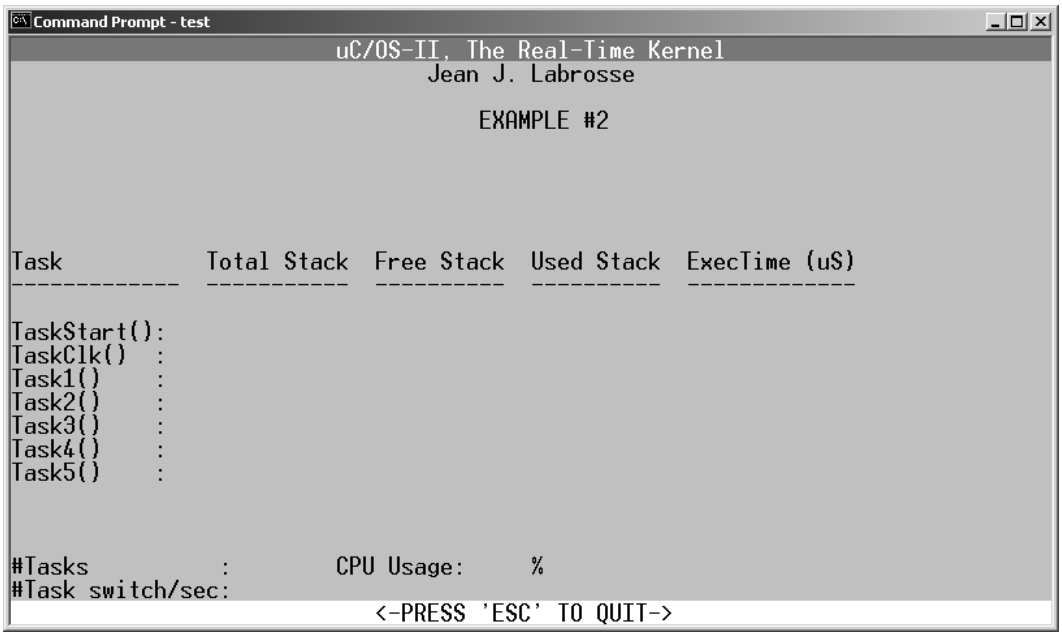
Listing 1.8 Example #2, TEST.C, TaskStart(). (Continued)

```
            if (key == 0x1B) {
                PC_DOSReturn();
            }
        }

        OSCtxSwCtr = 0;
        OSTimeDly(OS_TICKS_PER_SEC);                                    (4)
    }
}
```

L1.8(1) Although the function call is identical, `TaskStartDispInit()` initializes the display, as shown in Figure 1.4.

Figure 1.4 Initialization of the display by TaskStartDispInit().



L1.8(2) µC/OS-II allows you to have tasks or ISRs send messages to other tasks. In Example #2, I have Task 4 send a message to Task 5, and Task 5 will respond back to Task 4 with an acknowledgment message (described later). For this purpose, we need to create two kernel objects that are called *mailboxes*. A mailbox allows a task or an ISR to send a pointer to another task. The mailbox only has room for a single pointer. What the pointer points to is application specific, and, of course both the sender and the receiver need to agree about the contents of the message.

L1.8(3) `TaskStartCreateTasks()` creates six tasks using `OSTaskCreateExt()`. These tasks are not doing floating-point operations, and thus there is no need to call `OSTaskStkInit_FPE_x86()` to modify the stacks. However, I am doing stack checking on these tasks, so I call `OSTaskCreateExt()` with the proper options set.

L1.8(4) In Example #1, I called `OSTimeDlyHMSM()` to delay `TaskStart()` for one second. I decided to use `OSTimeDly(OS_TICKS_PER_SEC)` to show you that you can use either method. However, `OSTimeDly()` is slightly faster than `OSTimeDlyHMSM()`.

The code for `Task1()` is shown in Listing 1.9. `Task1()` checks the size of the stack for each of the seven application tasks (the six tasks created by `TaskStart()` and `TaskStart()` itself).

Listing 1.9 Example #2, TEST.C, Task1().

```
void  Task1 (void *pdata)
{
    INT8U       err;
    OS_STK_DATA data;
    INT16U      time;
    INT8U       i;
    char        s[80];


    pdata = pdata;
    for (;;) {
        for (i = 0; i < 7; i++) {
            PC_ElapsedStart();                                                  (1)
            err  = OSTaskStkChk(TASK_START_PRIO + i, &data);                    (2)
            time = PC_ElapsedStop();                                            (3)
            if (err == OS_NO_ERR) {
                sprintf(s, "%4ld        %4ld        %4ld        %6d",          (4)
                        data.OSFree + data.OSUsed,
                        data.OSFree,
                        data.OSUsed,
                        time);
                PC_DispStr(19, 12 + i, s, DISP_FGND_YELLOW);                    (5)
            }
        }
        OSTimeDlyHMSM(0, 0, 0, 100);                                            (6)
    }
}
```

L1.9(1)

L1.9(3) The execution time of `OSTaskStkChk()` is measured by wrapping `OSTaskStkChk()` with calls to `PC_ElapsedStart()` and `PC_ElapsedStop()`. `PC_ElapsedStop()` returns the time difference in microseconds.

L1.9(2) `OSTaskStkChk()` is a service provided by µC/OS-II to allow your code to determine the actual stack usage of a task. You call `OSTaskStkChk()` by passing it the task priority of the task you want to check. The second argument to the function is a pointer to a data structure

that holds information about the task's stack. Specifically, `OS_STK_DATA` contains the number of bytes used and the number of bytes free. `OSTaskStkChk()` returns an error code that indicates whether the call was successful. It would not be successful if I had passed the priority number of a task that didn't exist.

L1.9(4)

L1.9(5) The information retrieved by `OSTaskStkChk()` is formatted into a string and displayed.

L1.9(6) I decided to execute this task 10 times per second, but, in an actual product or application, you would most likely run stack checking every few seconds or so. In other words, it would make no sense to consume valuable CPU-processing time to determine worst-case stack growth.

The code for `Task2()` and `Task3()` is shown in Listing 1.10. Both of these tasks display a spinning wheel. The two tasks are almost identical. `Task3()` allocates and initializes a dummy array of 500 bytes. I wanted to consume stack space to show you that `OSTaskStkChk()` would report that `Task3()` has 502 bytes less than `Task2()` on its stack (500 bytes for the array and two bytes for the 16-bit integer). `Task2()`'s wheel spins clockwise at five rotations per second, and `Task3()`'s wheel spins counterclockwise at 2.5 rotations per second. `Task4()` and `Task5()` are shown in Listing 1.11.

Note: If you run Example #2 in a window under Microsoft Windows 95, 98, Me, NT, 2000, or XP, the rotation might not appear as quick. Simply press and hold the Alt key and then press the Enter key on your keyboard to make the DOS window use the whole screen. You can go back to window mode by repeating the operation.

Listing 1.10 Example #2, `TEST.C`, `Task2()` and `Task3()`.

```
void  Task2 (void *data)
{
    data = data;
    for (;;) {
        PC_DispChar(70, 15, '|',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '/',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '-',  DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
        PC_DispChar(70, 15, '\\', DISP_FGND_WHITE + DISP_BGND_RED);
        OSTimeDly(10);
    }
}
```

Listing 1.10 Example #2, TEST.C, Task2() and Task3(). (Continued)

```
void  Task3 (void *data)
{
    char    dummy[500];
    INT16U  i;

    data = data;
    for (i = 0; i < 499; i++) {
        dummy[i] = '?';
    }
    for (;;) {
        PC_DispChar(70, 16, '|',  DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '\\', DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '-',  DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '/',  DISP_FGND_WHITE + DISP_BGND_BLUE);
        OSTimeDly(20);
    }
}
```

Listing 1.11 Example #2, TEST.C, Task4() and Task5().

```
void  Task4 (void *data)
{
    char   txmsg;
    INT8U  err;

    data  = data;
    txmsg = 'A';
    for (;;) {
        OSMboxPost(TxMbox, (void *)&txmsg);                    (1)
        OSMboxPend(AckMbox, 0, &err);                          (2)
        txmsg++;                                               (3)
        if (txmsg == 'Z') {
            txmsg = 'A';
        }
    }
```

Listing 1.11 Example #2, TEST.C, Task4() and Task5(). (Continued)

```
}
void  Task5 (void *data)
{
    char  *rxmsg;
    INT8U  err;


    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);                          (4)
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW + DISP_BGND_RED);        (5)
        OSTimeDlyHMSM(0, 0, 1, 0);                                            (6)
        OSMboxPost(AckMbox, (void *)1);                                       (7)
    }
}
```

L1.11(1) `Task4()` sends a message (an ASCII character) to `Task5()` by posting the message to the `TxMbox`.

L1.11(2) `Task4()` then waits for an acknowledgment from `Task5()` by waiting on the `AckMbox`. The second argument to the `OSMboxPend()` call specifies a timeout, and I specified to wait forever because I passed a value of 0. By specifying a non-zero value, `Task4()` would have given up waiting after the specified timeout. The timeout is specified as an integral number of clock ticks.

L1.11(3) The message is changed when `Task5()` acknowledges the previous message.

L1.11(4) When `Task5()` starts execution, it immediately waits (forever) for a message to arrive through the mailbox `TxMbox`.

L1.11(5) When the message arrives, `Task5()` displays it on the screen.

L1.11(6)

L1.11(7) `Task5()` then waits for one second before acknowledging `Task4()`. I decided to wait for one second so that you could see it change on the screen. In fact, there must either be a delay in `Task5()` or one in `Task4()`, otherwise all lower priority tasks would not be allowed to run!

Finally, the code for `TaskClk()` is shown in Listing 1.12. This task executes every second, simply obtains the current date and time from a PC service called `PC_GetDateTime()` (see Chapter 18, "PC Services"), and displays it on the screen.

Listing 1.12 Example #2, TEST.C, TaskClk().

```
void  TaskClk (void *data)
{
    char s[40];


    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(60, 23, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

If you have the Borland C/C++ v4.5x compiler installed in the `C:\BC45` directory, you can experiment with `TEST.C`. After modifying `TEST.C`, you can type `MAKETEST` from the command prompt of the `TEST` directory to build a new `TEST.EXE`. If you don't have the Borland C/C++ v4.5x compiler or you have it installed in a different directory, you can make changes to `TEST.MAK`, `INCLUDES.H`, and `TEST.LNK` accordingly.

The `SOURCE` directory contains four files: `INCLUDES.H`, `OS_CFG.H`, `TEST.C`, and `TEST.LNK`. `OS_CFG.H` is used to determine µC/OS-II configuration options. `TEST.LNK` is the linker-command file for the Borland linker, `TLINK`.

1.03 Example #3

Example #3 shows how you can extend the functionality of µC/OS-II. Specifically, Example #3 uses the TCB extension capability of `OSTaskCreateExt()`, the user-defined context-switch hook [`OSTaskSwHook()`], the user-defined statistic-task hook [`OSTaskStatHook()`], and message queues. In this example, you should see how easy it is to determine how many times a task executes and how much time a task takes to execute. The execution time can be used to determine the CPU usage of a task relative to the other tasks.

The code for Example #3 is found in the `\SOFTWARE\uCOS-II\EX3_x86L\BC45` directory. You can open a DOS window and type

```
CD  \SOFTWARE\uCOS-II\Ex3_x86L\BC45\TEST
```

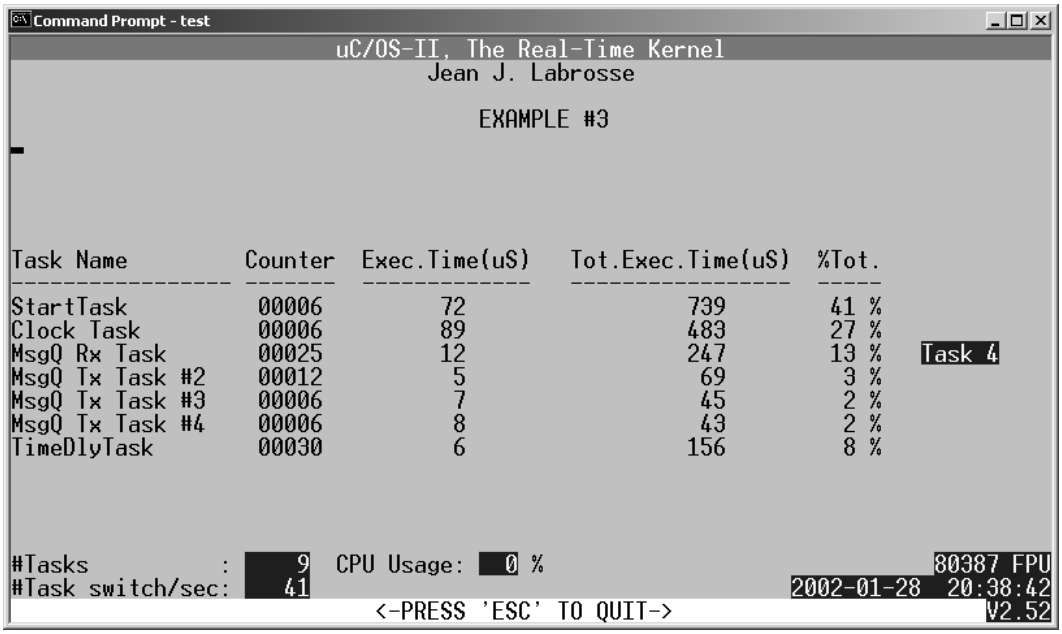
As usual, to execute Example #3, type `TEST` at the command prompt. The DOS window runs the `TEST.EXE` program.

After about one second, you should see the screen shown in Figure 1.5. I let `TEST.EXE` run for a couple of seconds before I captured the screen shot. Seven tasks are shown along with how many times they executed (*Counter* column), the execution time of each task in microseconds

(*Exec.Time(uS)* column), the total execution time since I started (*Tot.Exec.Time(uS)* column), and finally, the percentage of execution time of each task relative to the other tasks (*%Tot.* column).

Example #3 consists of nine tasks, as displayed in the lower left of Figure 1.5. Of those nine tasks, µC/OS-II creates two internal tasks: the idle task and a task that determines CPU usage. Example #3 creates the other seven tasks.

Figure 1.5 Example #3 running in a DOS window.



Portions of `TEST.C` are shown in Listing 1.13. You can examine the actual code using your favorite code editor.

Listing 1.13 Example #3, TEST.C.

```
#include  "includes.h"

#define          TASK_STK_SIZE     512

#define          TASK_START_ID       0
#define          TASK_CLK_ID         1
#define          TASK_1_ID           2
#define          TASK_2_ID           3
#define          TASK_3_ID           4
#define          TASK_4_ID           5
#define          TASK_5_ID           6
```

Listing 1.13 Example #3, TEST.C. (Continued)

```
#define          TASK_START_PRIO    10
#define          TASK_CLK_PRIO      11
#define          TASK_1_PRIO        12
#define          TASK_2_PRIO        13
#define          TASK_3_PRIO        14
#define          TASK_4_PRIO        15
#define          TASK_5_PRIO        16

#define          MSG_QUEUE_SIZE     20

typedef struct {                                                    (1)
    char    TaskName[30];
    INT16U  TaskCtr;
    INT16U  TaskExecTime;
    INT32U  TaskTotExecTime;
} TASK_USER_DATA;

OS_STK          TaskStartStk[TASK_STK_SIZE];
OS_STK          TaskClkStk[TASK_STK_SIZE];
OS_STK          Task1Stk[TASK_STK_SIZE];
OS_STK          Task2Stk[TASK_STK_SIZE];
OS_STK          Task3Stk[TASK_STK_SIZE];
OS_STK          Task4Stk[TASK_STK_SIZE];
OS_STK          Task5Stk[TASK_STK_SIZE];

TASK_USER_DATA  TaskUserData[7];                                    (2)

OS_EVENT     *MsgQueue;                                             (3)
void         *MsgQueueTbl[20];
```

L1.13(1) A data structure is created to hold additional information about a task. Specifically, the data structure allows you to add a name to a task (µC/OS-II doesn't directly provide this feature), keep track of how many times a task has executed, how long a task takes to execute, and finally the total time a task has executed.

L1.13(2) An array of the `TASK_USER_DATA` structure is allocated to hold information about each task created (except the idle and statistic tasks).

L1.13(3) µC/OS-II provides another message-passing mechanism called a *message queue*. A message queue is like a mailbox except that instead of being able to send a single pointer, a queue can hold more than one message (i.e., pointers). A message queue thus allows your tasks or ISRs to send messages to other tasks. What each of the pointers point to is application specific, and, of course, both the sender and the receiver need to agree about the contents of the

messages. Two elements are needed to create a message queue: an `OS_EVENT` structure and an array of pointers. The depth of the queue is determined by the number of pointers allocated in the pointer array. In this case, the message queue contains 20 entries.

`main()` is shown in Listing 1.14. Once more, only the new features are described.

Listing 1.14 Example #3, TEST.C, main().

```
void  main (void)
{
    PC_DispClrScr(DISP_BGND_BLACK);

    OSInit();

    PC_DOSSaveReturn();

    PC_VectSet(uCOS, OSCtxSw);

    PC_ElapsedInit();

    strcpy(TaskUserData[TASK_START_ID].TaskName, "StartTask");                (1)
    OSTaskCreateExt(TaskStart,
                    (void *)0,
                    &TaskStartStk[TASK_STK_SIZE - 1],
                    TASK_START_PRIO,
                    TASK_START_ID,
                    &TaskStartStk[0],
                    TASK_STK_SIZE,
                    &TaskUserData[TASK_START_ID],                             (2)
                    0);
    OSStart();
}
```

L1.14(1) Before a task is created, we assign a name to the task using the ANSI C library function `strcpy()`. The name is stored in the data structure [see L1.13(1)] assigned to the task.

L1.14(2) `TaskStart()` is created using `OSTaskCreateExt()` and passed a pointer to its user data structure. The TCB of each task in µC/OS-II can store a pointer to a user-provided data structure (see Chapter 3, "Kernel Structure" for details). This feature allows you to extend the functionality of µC/OS-II, as you will see shortly.

The code for `TaskStart()` is shown in Listing 1.15.

Listing 1.15 Example #3, TEST.C, TaskStart().

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT16S     key;


    pdata = pdata;

    TaskStartDispInit();

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);
    OS_EXIT_CRITICAL();

    OSStatInit();

    MsgQueue = OSQCreate(&MsgQueueTbl[0], MSG_QUEUE_SIZE);                    (1)

    TaskStartCreateTasks();                                                   (2)

    for (;;) {
        TaskStartDisp();

        if (PC_GetKey(&key)) {
            if (key == 0x1B) {
                PC_DOSReturn();
            }
        }

        OSCtxSwCtr = 0;
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

L1.15(1) Not much has been added except the creation of the message queue that is used by `Task1()`, `Task2()`, `Task3()`, and `Task4()`.

L1.15(2) As with Example #2, `TaskStartCreateTasks()` create six tasks. The difference is that each task is assigned an entry in the `TaskUserData[]` array. As each task is created, it's assigned a name just as I did when I created `TaskStart()` [see L1.14(1)].

As soon as `TaskStart()` calls `OSTimeDly(OS_TICKS_PER_SEC)`, µC/OS-II locates the next highest priority task that's ready to run, which is `Task1()`. Listing 1.16 shows the code for `Task1()`, `Task2()`, `Task3()`, and `Task4()` because I discuss them next.

Listing 1.16 Example #3, TEST.C, Task1() through Task4().

```
void  Task1 (void *pdata)
{
    char  *msg;
    INT8U  err;


    pdata = pdata;
    for (;;) {
        msg = (char *)OSQPend(MsgQueue, 0, &err);                          (1)
        PC_DispStr(70, 13, msg, DISP_FGND_YELLOW + DISP_BGND_BLUE);        (2)
        OSTimeDlyHMSM(0, 0, 0, 100);                                       (3)
    }
}

void  Task2 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 2");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                                (4)
        OSTimeDlyHMSM(0, 0, 0, 500);                                       (5)
    }
}
```

Listing 1.16 Example #3, TEST.C, Task1() through Task4(). (Continued)

```
void  Task3 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 3");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                                    (6)
        OSTimeDlyHMSM(0, 0, 0, 500);
    }
}

void  Task4 (void *pdata)
{
    char  msg[20];


    pdata = pdata;
    strcpy(&msg[0], "Task 4");
    for (;;) {
        OSQPost(MsgQueue, (void *)&msg[0]);                                    (7)
        OSTimeDlyHMSM(0, 0, 0, 500);
    }
}
```

L1.16(1) `Task1()` waits forever for a message to arrive through a message queue.

L1.16(2) When a message arrives, it is displayed on the screen.

L1.16(3) The task is delayed for 100ms to allow you to see the message received.

L1.16(4) `Task2()` sends the message "Task 2" to `Task1()` through the message queue.

L1.16(5) `Task2()` waits for half a second before sending another message.

L1.16(6)

L1.16(7) `Task3()` and `Task4()` send their messages and also wait half a second between messages.

Another task, `Task5()` (not shown) does nothing useful except delay itself for 1/10 of a second. Note that all µC/OS-II tasks must call a service provided by µC/OS-II to wait either for time to expire or for an event to occur. If this action is not done, the task prevents all lower priority tasks from running.

Finally, `TaskClk()` (also not shown) displays the current date and time once a second.

Events happen behind the scenes that are not apparent just by looking at the tasks in `TEST.C`. µC/OS-II is provided in source form, and it's quite easy to add functionality to µC/OS-II through special

functions called *hooks*. As of v2.52, nine hook functions exist, and the prototypes for these functions are shown in Listing 1.17.

Listing 1.17 µC/OS-II's hooks.

```
void  OSInitHookBegin(void);
void  OSInitHookEnd(void);
void  OSTaskCreateHook(OS_TCB *ptcb);
void  OSTaskDelHook(OS_TCB *ptcb);
void  OSTaskIdleHook(void);
void  OSTaskStatHook(void);
void  OSTaskSwHook(void);
void  OSTCBInitHook(OS_TCB *ptcb);
void  OSTimeTickHook(void);
```

The hook functions are normally found in a file called `OS_CPU_C.C` and are generally written by the person who does the port for the processor you intend to use. However, if you set a configuration constant called `OS_CPU_HOOKS_EN` to 0, you can declare the hook functions in a different file. `OS_CPU_HOOKS_EN` is one of many configuration constants found in the header file `OS_CFG.H`. Every project that uses µC/OS-II needs its own version of `OS_CFG.H` because you might want to configure µC/OS-II differently for each projet. Each example provided in this book contains its own `OS_CFG.H` in the `SOURCE` directory.

In Example #3, I set `OS_CPU_HOOKS_EN` to 0 and redefined the functionality of the hook functions in `TEST.C`. As shown in Listing 1.18, seven of the nine hooks don't actually do anything and thus don't contain any code.

Listing 1.18 Example #3, `TEST.C`, empty hook functions.

```
void  OSInitHookBegin (void)
{
}

void  OSInitHookEnd (void)
{
}

void  OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}

void  OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}
```

Listing 1.18 Example #3, TEST.C, empty hook functions. (Continued)

```
void  OSTaskIdleHook (void)
{
}

void  OSTCBInitHook (OS_TCB *ptcb)
{
    ptcb = ptcb;
}

void  OSTimeTickHook (void)
{
}
```

The code for `OSTaskSwHook()` is shown in Listing 1.19 and allows us to measure the execution time of each task, keeps track of how often each task executes, and accumulates total execution times of each task. `OSTaskSwHook()` is called when µC/OS-II switches from a low priority task to a higher priority task.

Listing 1.19 The task switch hook, OSTaskSwHook().

```
void  OSTaskSwHook (void)
{
    INT16U           time;
    TASK_USER_DATA  *puser;


    time  = PC_ElapsedStop();                                          (1)
    PC_ElapsedStart();                                                 (2)
    puser = OSTCBCur->OSTCBExtPtr;                                     (3)
    if (puser != (TASK_USER_DATA *)0) {                                (4)
        puser->TaskCtr++;                                              (5)
        puser->TaskExecTime     = time;                                (6)
        puser->TaskTotExecTime += time;                                (7)
    }
}
```

L1.19(1) A timer on the PC obtains the execution time of the task being switched out through `PC_ElapsedStop()`.

L1.19(2) It is assumed that the timer was started by calling `PC_ElapsedStart()` when the task was switched in. The first context switch probably reads an incorrect value, but this is not really critical.

L1.19(3) When `OSTaskSwHook()` is called, the global pointer `OSTCBCur` points to the TCB of the current task, while `OSTCBHighRdy` points to the TCB of the new task. In this case, however, we don't use `OSTCBHighRdy`. `OSTaskSwHook()` retrieves the pointer to the TCB extension that was passed in `OSTaskCreateExt()`.

L1.19(4) We then check to make sure we don't de-reference a `NULL` pointer. In fact, some of the tasks in this example do not contain a TCB extension pointer: the idle and the statistic tasks.

L1.19(5) We increment a counter that indicates how many times the task has executed. This counter is useful to determine if a particular task is running.

L1.19(6) The measured execution time (in microseconds) is stored in the TCB extension.

L1.19(7) The total execution time (in microseconds) of the task is also stored in the TCB extension. This element allows you to determine the percent of time each task takes with respect to other tasks in an application (discussed shortly).

When enabled (see `OS_TASK_STAT_EN` in `OS_CFG.H`), the statistic task `OSTaskStat()` calls the user-definable function `OSTaskStatHook()` that is shown in Listing 1.20. `OSTaskStatHook()` is called every second.

Listing 1.20 The statistic task hook, `OSTaskStatHook()`.

```
void  OSTaskStatHook (void)
{
    char    s[80];
    INT8U   i;
    INT32U  total;
    INT8U   pct;


    total = 0L;
    for (i = 0; i < 7; i++) {
        total += TaskUserData[i].TaskTotExecTime;                    (1)
        DispTaskStat(i);                                             (2)
    }
```

Listing 1.20 The statistic task hook, `OSTaskStatHook()`. (Continued)

```
    if (total > 0) {
        for (i = 0; i < 7; i++) {
            pct = 100 * TaskUserData[i].TaskTotExecTime / total;              (3)
            sprintf(s, "%3d %%", pct);
            PC_DispStr(62,                                                    (4)
                       i + 11,
                       s,
                       DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        }
    }
    if (total > 1000000000L) {
        for (i = 0; i < 7; i++) {
            TaskUserData[i].TaskTotExecTime = 0L;
        }
    }
}
```

L1.20(1) The total execution time of all the tasks (except the statistic task) is computed.

L1.20(2) Individual statistics are displayed at the proper location on the screen by `DispTaskStat()`, which takes care of converting the values into ASCII. In addition, `DispTaskStat()` also displays the name of each task.

L1.20(3)

L1.20(4) The percent execution time is computed for each task and displayed.

If you have the Borland C/C++ v4.5x compiler installed in the `C:\BC45` directory, you can experiment with `TEST.C`. After modifying `TEST.C`, you can type `MAKETEST` from the command prompt of the `TEST` directory to build a new `TEST.EXE`. If you don't have the Borland C/C++ v4.5x compiler or your have it installed in a different directory, you can make changes to `TEST.MAK`, `INCLUDES.H`, and `TEST.LNK` accordingly.

The `SOURCE` directory contains four files: `INCLUDES.H`, `OS_CFG.H`, `TEST.C`, and `TEST.LNK`. `OS_CFG.H` is used to determine µC/OS-II configuration options. `TEST.LNK` is the linker-command file for the Borland linker, `TLINK`.

1.04 Example #4

µC/OS-II is written entirely in C and requires some processor-specific code to adapt it to different processors. This processor-specific code is called a *port*. This book comes with two ports for the Intel 80x86 family of processors: Ix86L (see Chapter 14) and Ix86L-FP (see Chapter 15). Ix86L is used with 80x86 processors that are not fortunate enough to have an FPU, and Ix86L is used in all the examples so far. You should note that Ix86L still runs on 80x86 processors that do have an FPU. Ix86L-FP allows your applications to use the floating-point hardware capabilities of higher-end 80x86 compatible processors. Example #4 uses Ix86L-FP.

In this example, I created 10 identical tasks, each running 200 times per second. Each task computes the sine and cosine of an angle (in degrees). The angle being computed by each task is offset by 36 degrees (360 degrees divided by 10 tasks) from each other. Every time the task executes, it increments the angle to compute by 0.01 degree.

The code for Example #4 is found in the `\SOFTWARE\uCOS-II\EX4_x86L.FP\BC45` directory. You can open a DOS window and type

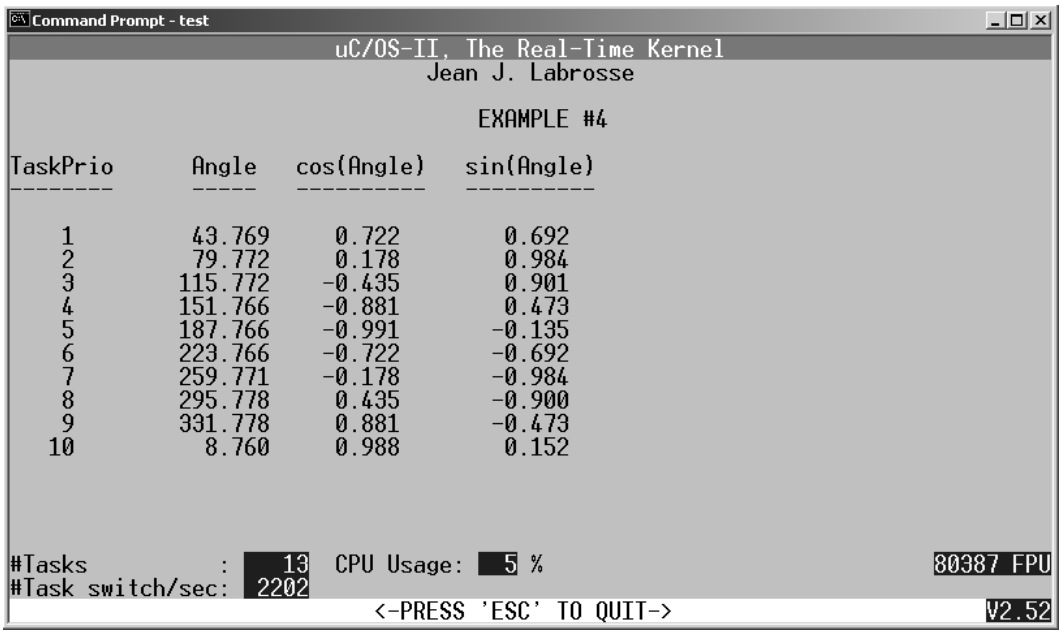
```
CD  \SOFTWARE\uCOS-II\Ex4_x86L.FP\BC45\TEST
```

As usual, to execute Example #4, simply type `TEST` at the command line prompt. The DOS window runs the `TEST.EXE` program.

After about two seconds, you should see the screen shown in Figure 1.6. I let `TEST.EXE` run for a few seconds before I captured the screen shot.

Example #4 consists of 13 tasks, as displayed in the lower left of Figure 1.6. Of those 13 tasks, µC/OS-II creates two internal tasks: the idle task and a task that determines CPU usage. Example #4 creates the other 11 tasks.

Figure 1.6 Example #4 running in a DOS window.



By now, you should be able to find your way around TEST.C. Example #4 doesn't introduce too many new concepts. However, there are a few subtleties done behind the scene, which I describe after discussing a few items in TEST.C. Listing 1.21 shows the code to create the 10 identical application tasks.

Listing 1.21 Example #4, TEST.C, TaskStartCreateTasks().

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;
    INT8U  prio;


    for (i = 0; i < N_TASKS; i++) {
        prio        = i + 1;                                          (1)
        TaskData[i] = prio;                                           (2)
        OSTaskCreateExt(Task,
                        (void *)&TaskData[i],                         (3)
                        &TaskStk[i][TASK_STK_SIZE - 1],
                        prio,
                        0,
                        &TaskStk[i][0],
                        TASK_STK_SIZE,
                        (void *)0,
                        OS_TASK_OPT_SAVE_FP);                         (4)
    }
}
```

L1.21(1) Because µC/OS-II doesn't allow multiple tasks at the same priority, I offset the priority of the identical tasks by 1 because task priority #0 is assigned to TaskStart().

L1.21(2) The task priority of each task is placed in an array.

L1.21(3) µC/OS-II allows you to pass an argument to a task when the task is first started. This argument is a pointer, and I generally call it pdata (pointer to data). The task priority saved in the array is actually passed as the task argument, pdata.

L1.21(4) Each of the tasks are doing floating-point calculations, and we want to tell the port (see Chapter 15) to save the floating-point registers during a context switch.

Listing 1.22 shows the actual task code.

Listing 1.22 Example #4, TEST.C, Task().

```
void  Task (void *pdata)
{
    FP32   x;
    FP32   y;
    FP32   angle;
    FP32   radians;
    char   s[81];
    INT8U  ypos;

    ypos  = *(INT8U *)pdata + 7;
    angle = (FP32)(*(INT8U *)pdata) * (FP32)36.0;                          (1)
    for (;;) {
        radians = (FP32)2.0 * (FP32)3.141592 * angle / (FP32)360.0;        (2)
        x       = cos(radians);
        y       = sin(radians);
        sprintf(s, "   %2d       %8.3f  %8.3f     %8.3f",
                *(INT8U *)pdata, angle, x, y);
        PC_DispStr(0, ypos, s, DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        if (angle >= (FP32)360.0) {
            angle  =   (FP32)0.0;
        } else {
            angle +=   (FP32)0.01;
        }
        OSTimeDly(1);                                                      (3)
    }
}
```

L1.22(1) The argument `pdata` points to an 8-bit integer containing the task priority. To make each task calculate different angles (not that it really matters), I decided to offset each task by 36 degrees.

L1.22(2) `sin()` and `cos()` assumes radians instead of degrees, and thus the conversion.

L1.22(3) Each task is delayed by one clock tick (i.e., 50ms), and thus each task executes 200 times per second.

Except for specifying `OS_TASK_OPT_SAVE_FP` in `TaskStartCreateTasks()`, you couldn't tell from `TEST.C` that we are using a different port from the other examples. In fact, it might be a good idea to always specify the option `OS_TASK_OPT_SAVE_FP` when you create a task [using `OSTaskCreateExt()`], and, if the port supports floating-point hardware, µC/OS-II can take the necessary steps to save and retrieve the floating-point registers during a context switch. That's, in fact, one of the beauties of µC/OS-II: portability of your applications across different processors.

In order to use a different port (at least for the 80x86), you only need to change the following files:

`INCLUDES.H` (in the `SOURCE` directory):

Instead of including:

```
\software\ucos-ii\ix86l\bc45\os_cpu.h
```

you simply need to point to a different directory:

```
\software\ucos-ii\ix86l-fp\bc45\os_cpu.h
```

`TEST.LNK` (in the `SOURCE` directory):

The linker-command file includes the floating-point emulation library in the non-floating-point version:

```
C:\BC45\LIB\EMU.LIB
```

and the hardware floating-point library needs to be referenced for the code that makes use of the FPU:

```
C:\BC45\LIB\FP87.LIB
```

`TEST.MAK` (in the `TEST` directory):

The directory of the port is changed from:

```
PORT=\SOFTWARE\uCOS-II\Ix86L\BC45
```

to:

```
PORT=\SOFTWARE\uCOS-II\Ix86L-FP\BC45
```

The compiler flags in the macro `C_FLAGS` include `-f287` for the floating-point version of the code and omits it in the non-floating-point version.

Chapter 2

2

Real-time Systems Concepts

Real-time systems are characterized by the severe consequences that result if logical as well as timing correctness properties of the system are not met. Two types of real-time systems exist: *soft* and *hard*. In a soft real-time system, tasks are performed by the system as fast as possible, but the tasks don't have to finish by specific times. In hard real-time systems, tasks have to be performed not only correctly but on time. Most real-time systems have a combination of soft and hard requirements. Real-time applications cover a wide range, but most real-time systems are *embedded*. An embedded system is a computer built into a system and not seen by the user as being a computer. The following list shows a few examples of embedded systems.

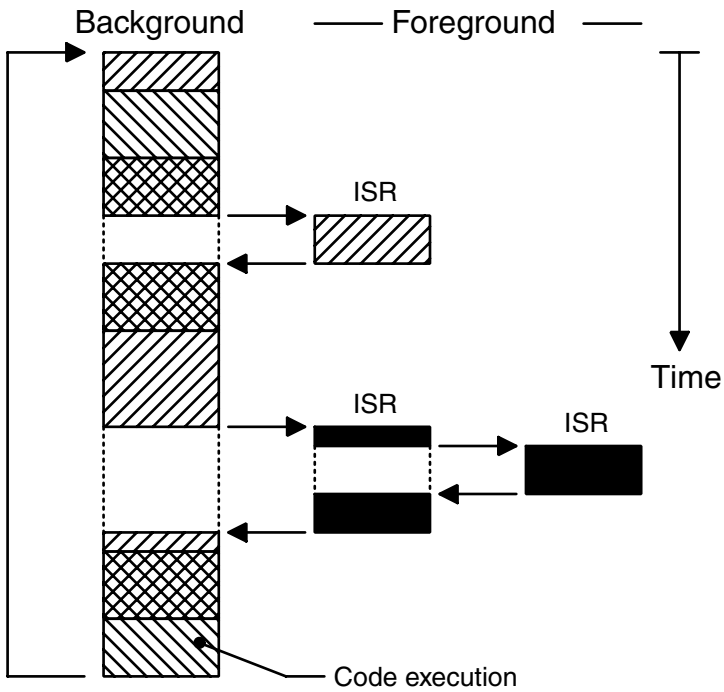
- Process control
  - Food processing
  - Chemical plants
- Automotive
  - Engine controls
  - Antilock braking systems
- Office automation
  - FAX machines
  - Copiers
- Computer peripherals
  - Printers
  - Terminals
  - Scanners
  - Modems
- Communication
  - Switches
  - Routers
- Robots
- Aerospace
  - Flight management systems
  - Weapons systems
  - Jet engine controls
- Domestic
  - Microwave ovens
  - Dishwashers
  - Washing machines
  - Thermostats

Real-time software applications are typically more difficult to design than non-real-time applications. This chapter describes real-time concepts.

2.00 Foreground/Background Systems

Small systems of low complexity are generally designed as shown in Figure 2.1. These systems are called *foreground/background systems* or *super-loops*. An application consists of an infinite loop that calls modules (i.e., functions) to perform the desired operations (background). Interrupt service routines (ISRs) handle asynchronous events (foreground). Foreground is also called *interrupt level*; background is called *task level*. Critical operations must be performed by the ISRs to ensure that they are dealt with in a timely fashion. Because of this, ISRs have a tendency to take longer than they should. Also, information for a background module that an ISR makes available is not processed until the background routine gets its turn to execute, which is called the *task-level response*. The worst case task-level response time depends on how long the background loop takes to execute. Because the execution time of typical code is not constant, the time for successive passes through a portion of the loop is nondeterministic. Furthermore, if a code change is made, the timing of the loop is affected.

Figure 2.1 Foreground/background systems.



Most high-volume microcontroller-based applications (e.g., microwave ovens, telephones, toys, and so on) are designed as foreground/background systems. Also, in microcontroller-based applications, it might be better (from a power consumption point of view) to halt the processor and perform all of the processing in ISRs.

2.01 Critical Sections of Code

A critical section of code, also called a *critical region*, is code that needs to be treated indivisibly. After the section of code starts executing, it must not be interrupted. To ensure that execution is not interrupted, interrupts are typically disabled before the critical code is executed and enabled when the critical code is finished (see also Section 2.03, "Shared Resources").

2.02 Resources

A resource is any entity used by a task. A resource can thus be an I/O device, such as a printer, a keyboard, a display, a variable, a structure, or an array.

2.03 Shared Resources

A shared resource is a resource that can be used by more than one task. Each task should gain exclusive access to the shared resource to prevent data corruption. This process is called *mutual exclusion*, and techniques to ensure mutual exclusion are discussed in Section 2.18, "Mutual Exclusion".

2.04 Multitasking

Multitasking is the process of scheduling and switching the central processing unit (CPU) between several tasks; a single CPU switches its attention between several sequential tasks. Multitasking is like foreground/background with multiple backgrounds. Multitasking maximizes the use of the CPU and also provides for modular construction of applications. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time applications. Application programs are typically easier to design and maintain if multitasking is used.

2.05 Tasks

A task, also called a *thread*, is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area (as shown in Figure 2.2).

Each task typically is an infinite loop that can be in any one of five states: *dormant*, *ready*, *running*, *waiting* (for an event), or *ISR* (interrupted) (Figure 2.3). The dormant state corresponds to a task that resides in memory but has not been made available to the multitasking kernel. A task is ready when it can execute but its priority is less than the currently running task. A task is running when it has control of the CPU. A task is waiting when it requires the occurrence of an event (for example, waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur, or time to expire). Finally, a task is in the ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt. Figure 2.3 also shows the functions provided by µC/OS-II to make a task move from one state to another.