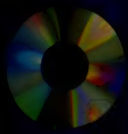
SAMS
PUBLISHING

Richard A. Burgess



CD-ROM contains source code for a 32-bit operating system, a 32-bit assembler, a 32-bit C compiler, and more!

- Master the details of 32-bit programming and operating systems
- Explore interprocess communications, hardware interfaces, systems programming, file system services, API specifications, and more
- Design and develop a feature-rich, 32-bit operating system

32

Developing Your Own 32-BIT OPERATING SYSTEM

Have you ever wanted to write your own computer operating system?

Or have you ever wanted to find out what it would take?

Or have you just wanted to see the source code with *real* comments?

If so, then this book is for you! This book contains the ideas, suggestions, and tools you will need—plus a sample operating system, MMURTL.

You will see these features of MMURTL highlighted:

Micro-kernel
Message based
Real-time
Multitasking
Multithreaded
Client/server oriented
ISA 32-bit PC-compatible

The CD-ROM contains:

Complete MMURTL operating system source code
Sample applications, including source code
A 32-bit assembler, including source code
A 32-bit C compiler, including source code

Developing Your Own 32-Bit Operating System

Richard A. Burgess

SAMS
PUBLISHING

201 West 103rd Street
Indianapolis, Indiana 46290

For Deborah, Valerie, and Heather

Copyright © 1995 by Sams Publishing

FIRST EDITION

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. For information, address Sams Publishing, 201 W. 103rd St., Indianapolis, IN 46290.

International Standard Book Number: 0-672-30655-7

Library of Congress Catalog Card Number: 94-69271

98 97 96 95 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 95-1 shows that the first printing of the book occurred in 1995.

Composed in AGaramond and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Publisher
Richard K. Swadley

Acquisitions Manager
Greg Wiegand

Managing Editor
Cindy Morrow

Acquisitions Editors
Chris Denny
Stacy Hiquet

Development Editors
Todd Bumbalough
Bradley L. Jones

Software Development Specialist
Timothy Wilson

Production Editor
Deborah Frisby

Copy Editors
Fran Blauw
Johnna VanHoose
Joe Williams

Editorial Coordinator
Bill Whitmer

Editorial Assistants
Carol Ackerman
Sharon Cox
Lynette Quinn

Technical Reviewer
David McLarty

Marketing Manager
Gregg Bushyeager

Assistant Marketing Manager
Michelle Milner

Cover Designer
Tim Amrhein

Book Designer
Alyssa Yesh

Director of Production and Manufacturing
Jeff Valler

Production Manager
Kelly Dobbs

Manufacturing Coordinator
Paul Gilchrist

Production Analysts
Angela D. Bannan
Bobbi Satterfield
Dennis Clay Hager

Support Services Supervisor
Mary Beth Wakefield

Graphics Image Specialists
Teresa Forrester
Clint Lahnen
Tim Montgomery
Dennis Sheehan
Greg Simsic
Susan VandeWalle
Jeff Yesh

Team Supervisor
Katy Bodenmiller

Page Layout
Charlotte Clapp
Mary Ann Cosby
Judy Everly
Shawn MacDonald
Tina Trettin

Proofreading
Donna Martin
Brian-Kent Proffitt
Erich J. Richter

Indexer
Greg Eldred

Overview

	Preface	xviii
I	**Introduction**	
1	Overview	3
2	General Discussion and Background	13
II	**Architecture and General Theory**	
3	The Tasking Model	25
4	Interprocess Communications	41
5	Memory Management	55
6	The Hardware Interface	81
7	OS Initialization	99
8	Programming Interfaces	115
III	**MMURTL-Specific Details**	
9	Application Programming	129
10	Systems Programming	141
11	The MMURTL Monitor Program	159
12	The Debugger	181
13	Keyboard Service	189
14	File System Service	201
15	API Specification	221
16	MMURTL Sample Software	265
IV	**The Operating System Source Code**	
17	Introduction to the Source Code	327
18	Kernel Code	335
19	Memory Management Code	377
20	Timer Management Code	405
21	Initialization Code	419
22	Job Management Code	439

23 Debugger Code 471
24 Selected Device Driver Code 497
25 Keyboard Code 541
26 Video Code 571
27 File System Code 591

V Additional Software

28 DASM: A 32-Bit Intel-Based Assembler 663
29 CM32: A 32-Bit C Compiler 697
Index 719

Contents

Part I Introduction

1 Overview **3**

What This Book Is About 5
Who Should Use This Book 6
My Basic Design Goals (Yours May Vary) 7
A Brief Description of MMURTL 9
Uses for MMURTL 9
Similarities to Other Operating Systems 10
Hardware Requirements 10

2 General Discussion and Background **13**

Where Does One Begin? 14
You Are Where You Were When 14
The Development Platform and the Tools 15
The Chicken and the Egg 16
Early Hardware Investigation 17
The Real Task at Hand (Pun Intended) 20
A Starting Point 20
The Critical Path 21
Working Attitude 22

Part II Architecture and General Theory

3 The Tasking Model **25**

Terms 26
Tasks 26
The Kernel 27
Preemptive Multitasking 27
Resource Management 27
Task Management 28
CPU Time 30
Single Tasking 30
Multitasking 31
Cooperative Multitasking 31
Preemptive Multitasking 32
Task Scheduling 32
Who, When, and How Long? 32
Scheduling Techniques 33
Other Scheduling Factors 35

Mixed Scheduling Techniques 36
The Tasking Model I Chose 38
Interrupt Handling 39

4 Interprocess Communications 41

Messaging and Tasks 42
Synchronization 42
Semaphores 43
Pipes 44
Messages 44
Reentrancy Issues 51
Interrupt Latency 53
Memory Management Issues 53

5 Memory Management 55

Basic Terms 56
Memory Model 57
Simple Flat Memory 57
Paged Flat Memory 58
Demand-Paged Flat Memory 59
Virtual Paged Memory 59
Demand Paged Virtual Memory 61
Segmented Memory 61
Memory Management 62
Tracking Linear Memory Allocation 63
Tracking Physical Memory 69
Initialization of Memory Management 69
Memory Protection 70
An Intel-Based Memory-Management Implementation 70
A Few More Words On Segmentation 71
How MMURTL Uses Paging 72

6 The Hardware Interface 81

Hardware Isolation 82
The CPU 82
The Bus Structure 83
Serial I/O 84
Parallel I/O 85
Block-Oriented Devices 85
The Keyboard 86
Video 87

Direct Memory Access (DMA) 89
Timers 97
The Priority Interrupt Controller Unit (PICU) 98

7 The OS Initialization 99

Getting Booted 100
Boot ROM 100
The Boot Sector 101
Operating System Boot Image 110
Other Boot Options 111
Basic Hardware Initialization 111
Static Tables and Structures 112
Initialization of Task Management 112
Initialization of Memory 113
Dynamic Tables and Structures 114

8 Programming Interfaces 115

Application Programming Interface 116
Documenting Your API 116
Procedural Interfaces 116
Systems Programming Interface 120
Internal Cooperation 120
Device Driver Interfaces 120
A Device-Driver Interface Example 121

Part III MMURTL-Specific Details

9 Application Programming 129

Terminology 130
Understanding 32-Bit Software 130
Operating-System Calling Conventions 131
Stack Usage 132
Memory Management 132
Operating-System Protection 134
Application Messaging 135
Starting a New Thread 136
Job Control Block 136
Basic Keyboard 138
Basic Video 138
Terminating Your Application 140
Replacing Your Application 140

10 Systems Programming 141
Writing Message-Based System Services ... 142
Initializing Your System Service ... 142
A Simple System Service Example ... 143
The Request Block ... 144
Items in the Request Block ... 144
Asynchronous Services ... 146
System Service Error Handling ... 146
Writing Device Drivers ... 147
Device Driver Theory ... 148
Building Device Drivers ... 148
How Callers Reach Your Driver ... 149
Device Driver Setup and Installation ... 149
System Resources for Device Drivers ... 150
Detailed Device Interface Specification ... 152
Initializing Your Driver ... 157
Operating System Functions for Device Drivers ... 158
Standard Device Error Codes ... 158

11 The MMURTL Monitor Program 159
Active Job (Video and Keyboard) ... 160
Initial Jobs ... 160
Monitor Function Keys ... 161
Monitor Program Theory ... 161
Performance Monitoring ... 162
Monitor Source Listing ... 163

12 The Debugger 181
Using the Debugger ... 182
Entering the Debugger ... 182
Exiting the Debugger ... 182
Debugger Display ... 183
Debugger Function Keys ... 183
Debugging Your Application ... 185
Debugger Theory ... 185

13 Keyboard Service 189
Available Service ... 190
Read Keyboard ... 191
Notify On Global Keys ... 191
Cancel Notify On Global Keys ... 192
Assign Keyboard ... 192

Key Codes and Status 193
Alphanumeric Key Values 193
Shift-State Byte (Second Byte) 193
Lock-State Byte (Third Byte) 194
Numeric Pad Indicator 194
Key Codes 194
Function Strip Key Codes 196
Numeric Pad Key Codes 197
Cursor, Edit, and Special Pad Key Codes 198
Your Keyboard Implementation 198

14 File System Service 201

File Specifications 202
Network File Request Routing 202
File Handles 203
File Open Modes 203
File Access Type 203
Block Mode 203
Stream Mode 204
Logical File Address 204
File System Requests 204
Procedural Interfaces 206
Device Access Through the File System 206
File System Functions in Detail 207
OpenFile 207
CloseFile 207
ReadBlock 208
WriteBlock 209
ReadBytes 209
WriteBytes 210
GetFileLFA 211
SetFileLFA 211
GetFileSize 212
SetFileSize 213
CreateFile 213
RenameFile 214
DeleteFile 214
CreateDirectory 215
DeleteDirectory 216
GetDirSector 216

File System Theory 217
Internal File System Structures 217
File Operations 219

15 API Specification 221

Public Calls 222
Parameters to Calls (Args) 222
Stack Conventions 224
Calls Grouped by Functionality 224
Alphabetical Call Listing 227
AddCallGate 227
AddIDTGate 228
Alarm 229
AliasMem 229
AllocDMAPage 230
AllocExch 230
AllocOSPage 230
AllocPage 231
Beep 231
Chain 231
CheckMsg 232
ClrScr 232
Compare 233
CompareNCS 233
CopyData 233
CopyDataR 233
DeAliasMem 234
DeAllocExch 234
DeAllocPage 234
DeviceInit 235
DeviceOp 235
DeviceStat 236
DMASetUp 236
EditLine 237
EndOfIRQ 237
ExitJob 238
FillData 238
GetCmdLine 238
GetCMOSTime 239
GetCMOSDate 239

GetDMACount 240
GetExitJob 240
GetIRQVector 240
GetJobNum 241
GetNormVid 241
GetPath 241
GetPhyAdd 242
GetpJCB 242
GetSysIn 242
GetSysOut 243
GetTimerTick 243
GetTSSExch 243
GetUserName 243
GetVidChar 244
GetVidOwner 244
GetXY 244
InByte 244
InDWord 245
InWord 245
InWords 245
InitDevDr 245
ISendMsg 246
KillAlarm 247
LoadNewJob 247
MaskIRQ 248
MicroDelay 248
MoveRequest 248
NewTask 249
OutByte 249
OutDWord 249
OutWord 250
OutWords 250
PutVidAttrs 250
PutVidChars 250
QueryPages 251
ReadCMOS 251
ReadKbd 251
RegisterSvc 252
Request 252
Respond 254
ScrollVid 254

SendMsg 254
SetCmdLine 255
SetExitJob 255
SetIRQVector 256
SetJobName 257
SetNormVid 257
SetPath 258
SetPriority 258
SetSysIn 258
SetSysOut 259
SetUserName 259
SetVidOwner 259
SetXY 260
Sleep 260
SpawnTask 261
Tone 261
TTYOut 262
UnMaskIRQ 262
UnRegisterSvc 263
WaitMsg 263

16 MMURTL Sample Software 265

Command-Line Interpreter (CLI) 266
Internal Commands 266
External Commands 270
Global "Hot Keys" 271
CLI Source Listing 271
A Simple Editor 288
Editor Screen 288
Editor Commands 288
Editor Source Listing 290
DumbTerm 314
Print 317
Print Source Listing 317
System Service Example 322
Service.C Listing 322
TestSvc.C Listing 324

Part IV The Operating System Source Code

17 Introduction to the Source Code 327

Comments in the Code 328

Calling Conventions 328
Organization 328
Building MMURTL 331
Using Pieces of MMURTL in Your OS 332
Using Pieces of MMURTL in Other Projects 333

18 Kernel Code 335

Naming Conventions 336
Kernel Data 336
Local Kernel Helper Functions 337
enQueueMsg 337
deQueueMsg 338
deQueueTSS 339
enQueueRdy 340
deQueueRdy 340
ChkRdyQ 341
Internal Public Helper Functions 342
RemoveRdyJob 342
GetExchOwner 344
SetExchOwner 345
SendAbort 346
Public Kernel Functions 348
Request() 348
Respond() 351
MoveRequest() 354
SendMsg() 355
ISendMsg() 357
WaitMsg() 359
CheckMsg() 363
NewTask() 366
SpawnTask() 368
AllocExch() 370
DeAllocExch() 371
GetTSSExch() 373
SetPriority() 374
GetPriority() 374

19 Memory Management Code 377

Memory Management Data 378
Internal Code 378
InitMemMgmt 379
FindHiPage 381
FindLoPage 382

MarkPage 383
UnMarkPage 384
LinToPhy 385
FindRun 385
AddRun 387
AddAliasRun 388
AddUserPT 390
AddOSPT 392
Public Memory Management Calls 394
AddGDTCallGate 394
AddIDTGate 395
AllocOSPage 396
AliasMem 397
DeAliasMem 400
DeAllocPage 401
QueryMemPages 403
GetPhyAdd 404

20 Timer Management Code 405

Choosing a Standard Interval 406
Timer Data 407
Timer Code 408
The Timer Interrupt 408
Sleep() 410
Alarm() 411
KillAlarm() 412
MicroDelay() 413
GetCMOSTime() 414
GetCMOSDate() 415
GetTimerTick() 416
Beep_Work() 416
Beep() 417
Tone() 418

21 Initialization Code 419

OS Global Data 420
Operating System Entry Point 422
Initialization Helpers 429

22 Job Management Code 439

Reclamation of Resources 440
Job Management Helpers 440

Job Management Listing 448

23 Debugger Code 471

Debugger Interrupt Procedure 472
Debugger Source Listing 473

24 Selected Device Driver Code 497

IDE Disk Device Driver 498
The RS-232 Asynchronous Device Driver 519
RS232.h 519
The Device Driver, RS-232.C 520

25 Keyboard Code 541

Keyboard Data 542
Keyboard Code 550
Keyboard ISR 550
Keyboard Translation 551
Reading the Final Buffer 557
The Keyboard Service 557
Keyboard Procedural Interface 563
Debugger Keyboard 565
Keyboard Hardware Set Up 566
Keyboard Service Initialization 567
Hardware Helpers 568

26 Video Code 571

Virtual Video Concept 572
VGA Text Video 572
The Video Code 573
Relating MMURTL's Video Code to Your Operating System 589

27 File System Code 591

How MMURTL Handles FAT 592
Physical (Absolute) Sector Numbers 592
Hidden Sectors 592
Logical Block Address (LBA) 592
MS-DOS Boot Sector 593
File System Initialization 593
File System Listing 593

Part V Additional Software

28 DASM: A 32-Bit Intel-Based Assembler 663

DASM Concepts 664

Using DASM 665
Command Line Options 667
DASM Source Files 668
Local (Nonpublic) Declarations 669
Memory Model 669
DASM Program Structure 669
DOT Commands 670
DASM Versus DOS Assemblers 672
DASM Addressing Modes 673
Storage Declarations 675
DASM Peculiar Syntax 676
386 Instruction Set 682
Executable File Format 688
Error Codes From DASM 693
Your Own Assembler? 696

29 CM32: A 32-Bit C Compiler 697

Memory Usage and Addressing 699
Important Internal Differences 699
Variable-Length Argument Lists 699
CM32 Language Specifics 700
Far Calls 700
Interrupt Functions 700
Supported C Language Features 701
Library Header File Access 702
Limitations as Compared to ANSI 702
Far External Functions 703
Type Conversion and Pointers 704
Initialization of Static Variables 704
Other Implementation-Dependent Information 706
Using CM32 707
Command Line Options 707
Library Functions 708
Library Code Examples 709

Index 719

Preface

I'm taking this opportunity to give you some insight into the frame of mind I've been in for the past few years while writing my own operating system. For years I have developed software under the direction of others, using tools designed by others, eventually for others to use. The tools I used were top-quality, the operating systems I used all worked as advertised for the most part, and the jobs got done.

One of the more perplexing things I ran across was the tendency to over-design something. When this tendency was added to the requirements for backward compatibility, the resulting products could become endless projects, or even nightmares, if not carefully watched.

I wanted to break free of the bonds of backward compatibility. I wanted to produce a system that was powerful, yet elegant and simple in its implementation. I also wanted to make use of the more interesting designs and ideas I have run across over the years. When an avenue was presented to me that could reduce the complexity of the code and sacrifice only a small amount of power, it was my obvious path.

Quite often, I refer to Albert Einstein's words on simplicity: "Everything should be as simple as possible, but not simpler." You'll run across this idea or a reference to it in several chapters of this book. Simplicity was also the best way to provide you with a foundation of information and ideas for your own operating system, especially if you've never undertaken such a gargantuan project.

Here's one other important note: Version 1.0 (or the first-released version) of every operating system I've ever used was not to be trusted with valuable information and was not a good basis on which to build a time-critical project. I put MMURTL, the operating system included with this book, in this same class. It's new, untested, and still under construction to a large extent.

Good luck, and have fun!

Richard A. Burgess

Acknowledgments

This has been a long five years. I have had a tremendous amount of help along the way. In Chapter 2, "General Discussion and Background," I mention a friend who initially started with me on this project. He is Reginald B. Carie, code-cutter extraordinaire. Very early in the game, Reggie went on to make more important things than an operating system. He and his lovely wife had a baby. Now that's work. Thanks Reg—I couldn't have gotten this far without you. Not a whole lot of what we did together is left, but your enthusiasm will remain forever.

I was on active duty in the U.S. Coast Guard when I started MMURTL. Many people at the U.S. Coast Guard Information Systems Center (now called the Telecommunications and Information Systems Command) in Alexandria, Virginia, were backboards for my ideas and were sources of information. Tom Clark (the one-man think tank) and Dan Haynes (the wunderkind of programming) lost a few lunch hours to my babbling questions and ideas. Commander Scott Bair laughed at me when I needed it. Thanks guys.

Dave McLarty of Convergent Consultants of Atlanta made some serious contributions while listening to my ideas (and paying the phone bill most of the time). Dave knows real-time multitasking and has taken it places I've never even been. Thanks Dave.

I had learned a lot at the 4 1/2-year mark, and I decided to share what I had learned in the form of a book. I initially had delusions of grandeur for a commercial product, but that's too much work. (Hey! I'm retired....) When I submitted the proposal to a publisher, I immediately got a positive response from Stacy Hiquet of Sams Publishing. Stacy was briefly my acquisitions editor and was promoted off my book on to bigger things. Congratulations Stacy and thanks. Chris Denny took over as acquisitions editor and has been so easy to work with that it's almost scary. Thanks Chris. Brad Jones (famous author and all-around great guy) has been one of my development editors. He has given me some serious insight into the publishing industry. I have been doing my best not to be a thorn in his side, and he hasn't once destroyed my will to live. Thanks Brad. Thanks also to Todd Bumbalough, development editor, who helped guide the project in its later stages. Thanks to the production editor, Deborah Frisby and to the copy editors, Fran Blauw, Johnna VanHoose, and Joe Williams.

I must also apologize to the copy editors and proofreaders. I do know the difference between *there, their*, and *they're*, and also between *its* and *it's*. While typing at 60 WPM, listening to the Crash Test Dummies and Boingo at about 100 decibels, and gulping down coffee and sodas like nobody's business, anything can happen and usually does. Hundreds of pages in six months...never again.

Special thanks to Nita, for some serious scanning of the mental manuscript.

Special thanks to Dave Dunfield of Dunfield Development Systems; Nepean, Ontario; Canada, for his contributions to the CD-ROM and for the beginnings of the CM-32 compiler.

About the Author

Richard A. Burgess is a retired U.S. Coast Guard commissioned officer (1993). His first 10 years in the Coast Guard were in electronics technology and maintenance. His last 10 years of service were all computer-science related; he was the primary software design engineer for the Coast Guard's Communication Station Control System, and he headed the Coast Guard's Software Development Standards Team. He taught computer electronics technology at the Computer Learning Center of Washington, D.C., for two years, and he consulted in and around Washington under the guise of The D-Group. He is now the Chief Information Officer with Optipat, Inc. of Arlington, Virginia, a rapidly growing, high-technology, patent-trademark-and-copyright imaging firm. He would rather be on a beach somewhere in Florida and claims he is working to that end.

I

Introduction

1

Overview

Computer programmers and software engineers work with computer operating systems every day. They use them, they work with them, they even work "around" them if that's what it takes to get their jobs done. If you're an experienced programmer, I'm willing to bet you've pondered changes you would make to the operating systems you use, or you've even thought about what you would build and design into an operating system if you were to write your own.

You don't have to be Albert Einstein to figure out that many years of labor can go into writing a computer operating system. You also don't have to be Einstein to write one, although I'm sure it would have increased my productivity a little while writing my own. Unfortunately, I have only Albert's absent-mindedness, not his IQ.

Late in 1989, I undertook the task of writing a computer operating system. What I discovered was that most of the books about operating-system design were very heavy on general theory and existing designs, but very light on specifics, code, and advice on the seemingly endless tasks facing someone who really wants to sit down and create one. I'm not knocking the authors of these books; I've learned a lot from all that I've read. It just seemed there should be a more "down to earth" book that documents what you need to know to get the job done.

Writing my own operating system has turned into a very personal thing for me. I have been involved in many large software projects where the deliverables were conceived by others and I was to turn them into working code and documentation. In the case of my own operating system, I set the ground rules. I decided what stayed and what went. I determined the specifications for the final product.

When I started out, I thought the "specs" would be the easy part. I would just make a list, do some research on each of the desirables, and begin coding. Not so fast there, bucko!

I may not have to tell you this if you've done some serious "code cutting" yourself, but the simplest items on the wish list for a piece of software can add years to the critical path for research and development, especially if you have a programming team comprising just one person.

My first list of desirables looked like a kid's letter to Santa Claus. It turned out that I hadn't been a good-enough boy to get all those things (and the average life expectancy of a male human being wouldn't allow it either). Realizing this, I whittled away to find a basic set of ingredients that constitute the real make-up of the "guts" of operating systems. The most obvious of these, as viewed by the outside programmer, are the tasking model, the memory model, and the programming interface. As coding began, certain not-so-obvious factors came into play, such as the OS-to-hardware interface, portability issues, size, and speed.

One of the most obvious concerns for commercial operating-system developers is compatibility. If you write something that no one can use, it's unlikely they'll use it. Each of the versions of MS-DOS that were produced had to be able to run software written for previous versions. When OS/2 was first developed, the "DOS-BOX" was a bad joke. It didn't provide the needed

compatibility to run the programs that business depended on. In the early days, UNIX had a somewhat more luxurious life, because source-code compatibility was the main objective; you were pretty much expected to recompile something if you wanted it to run on your particular flavor of the system. You may have noticed that this has changed substantially.

I did not intend to let compatibility issues drive my design decisions. I was not out to compete with the likes of Microsoft, IBM, or Sun. You, however, may want compatibility if you intend to write your own operating system. It's up to you. If you desire compatibility, the existing interfaces are all well documented, so you don't need to design your own. But I wanted a small API. That was part of my wish list.

Along my somewhat twisted journey of operating-system design and implementation, I have documented what it takes to really write one, and I've included the "unfinished" product for you to use with very few limitations or restrictions. Even though I refer to it as unfinished, it's a complete system. I believe that when a piece of software is "finished," it's probably time to replace it because of obsolescence or lack of maintenance.

What This Book Is About

In this book I discuss the major topics of designing your own OS: the tasking model, the memory model, programming interfaces, hardware interface, portability issues, size, and speed. I back up these discussions by showing you what I decided on (for my own system), and then finally I discuss my code in great detail. Your wish list may not be exactly like mine. In fact, I'm sure it won't be. For instance, you'll notice the ever-so-popular term "object oriented" conspicuously missing from any of my discussions. This was completely intentional. This term clouds far too many real issues these days. Don't get me wrong—I'm an avid C++ and OOP user. But that's not what this book is about, so I've removed the mystique. Once again, your goals will be your own.

Throughout this book, I use the operating system I developed (MMURTL) to help explain some of the topics I discuss. You can use pieces of it, ideas from it, or even all of it if you wish to write your own.

I have included the complete source code to my 32-bit, message-based, multitasking, real-time, operating system (MMURTL), which is designed for the Intel 386/486/Pentium processors on the PC Industry Standard Architecture (ISA) platforms.

The source code, written in 32-bit, Intel-based assembly language and C, along with all the tools necessary to build, modify, and use the operating system are included on the accompanying CD-ROM. The hardware requirements section below tells you what you need to run MMURTL and to use the tools I have developed. It's not really required if you decide on a different platform and simply use this book as a reference.

One thing I didn't put on my wish list was documentation. I always thought that many systems lacked adequate documentation (and most of it was poorly written). I wanted to know more about how the system worked internally, which I felt would help me write better application software for it. For this reason, I have completely documented every facet of my computer operating system and included it here. You may or may not want to do this for your system, but it was a "must" for me. I think you will find that this will help you if you intend to write your own. Well-commented code examples are worth ten times their weight in generic theory or simple text-based pseudo-algorithms.

The "Architecture and General Theory" part of this book, along with the sections that are specific to the MMURTL operating system, will provide the necessary information for you to use this system as is, to redesign it to meet your requirements, or to write your own. If you don't like the way something in my system is designed or implemented, change it to suit your needs. The only legal restriction placed on your use of the source code is that you may not sell it or give it away.

Who Should Use This Book

This book is for you if you are a professional programmer or a serious hobbyist, and you have an interest in any of the following topics:

- Writing a 32-bit microcomputer operating system.
- 80386/486/Pentium 32-bit assembly language. (An assembler is included.)
- The C programming language. (A compiler for C is included.)
- Intel 80386/486/Pentium paged memory operation and management using the processor paging hardware.
- Intel 80386/486/Pentium 32-bit hardware and software task management using the processor-hardware task-management facilities.
- Embedded or dedicated systems using the 32-bit PC ISA architecture.
- Real-time, message-based operating systems.
- PC Industry Standard Architecture hardware management—including: DMA controllers, hardware timers, priority interrupt controller units, serial and parallel ports, and hard or floppy disk controllers.
- File systems. (A DOS File Allocation Table-compatible file system in C is included.)

You may note that throughout this book I refer to the 386, 486, and Pentium processors as if they were all the same. If you know a fair amount about the Intel 32-bit processors, you know that a quantum leap was made between the 286 and 386 processors. This was the 16- to 32-bit jump.

The 486 and Pentium series have maintained compatibility with the 386 instruction set. Even though these newer processors have added speed, a few more instructions, and some serious "turbo" modifications, they are effectively super-386 processors. In my opinion, from a programmer's standpoint the next quantum leap really hasn't been made, even though the Pentium provides 64-bit data access.

My Basic Design Goals (Yours May Vary)

A review of my ambitions when I was designing the MMURTL operating system may help you understand what I faced, my end results, and what you may face. If you're going to write your own system, I recommend that you avoid an overly extensive wish list. You may start with my list and modify it, or you may start one from scratch. From my own wish list, you may be able to get a good handle on your own list. Chapter 2, "General Discussion and Background," may also help by explaining in some detail how my list came to be.

Here was my wish list:

True multitasking: Not just task switching between programs, or even sharing the processor between multiple programs loaded in memory, but real multithreading—the ability for a single program to create several threads of execution that could communicate and synchronize with each other while carrying out individual tasks. I'll discuss some of your options, and also what I decided on, in Chapter 3, "The Tasking Model," and Chapter 4, "Interprocess Communications."

Real-time operation: The ability to react to outside events in real time. This design goal mandated that MMURTL be message based. The messaging system would be the heart of the kernel. Synchronization of the messages would be the basis for effective CPU utilization (the tasking model). Real-time operation may not even be on your list of requirements, but I think you'll see that the push in the industry is towards real-time systems, and for a very good reason. We, the humans, can be thought of as "outside events." We want the system to react to us. This is also discussed in Chapters 3 and 4.

Client/server design: The ability to share services with multiple client applications, whether on the same machine or across a network. A message-based operating system seemed the best way to do this. I added the Request and Respond message types to the more common Send and Wait found on most message-based systems. Message agents could be added to any platform. If you don't have this requirement, the Request and Respond primitives could be removed, but I don't recommend it. I discuss this in Chapter 4.

A common, affordable hardware platform with minimal hardware requirements: The most common 32-bit platform in the world is the PC ISA platform. How many do you think are out there? Millions! How much does a 386SX cost these days? A

little more than dirt (or maybe a little less). Of course, you may have a different platform in mind. The hardware interface may radically change, based on your choice. I discuss this in Chapter 6, "The Hardware Interface."

Flat 32-bit virtual memory model: Those of you that program on the Intel processors know all about segmentation and the headaches it can cause (not to mention Expanded, Extended, LIM, UMBs, HIMEM, LOMEM, YOURMOTHERSMEM, and who knows what other memory schemes and definitions out there). Computers using the Intel-compatible 32-bit processors are cheap, and most of them are grossly underused. MMURTL would use the memory paging capabilities of 386/486 processors to provide an *easy* 32-bit flat address space for each application running on the system. Chapter 5, "Memory Management," covers memory management but not all of the options. Many detailed books have been written on schemes for memory management, and I stuck with a very simple paged model that made use of the Intel hardware. You could go overboard here and throw in the kitchen sink, if desired.

Easy programming: The easiest way to interface to an operating system from a procedural language, such as C or Pascal, is with a procedural interface. A procedural interface directly into the operating system (with no intermediate library) is the easiest there is. Simple descriptive names of procedures and structures are used throughout the system. I also wanted to maintain a small, uncluttered Applications Programming Interface (API) specification, adding only the necessary calls to get the job done—power without bloat, the right mix of capability and simplicity. I would shoot for less than 200 basic public functions. I discuss some of your options in Chapter 8, "Programming Interfaces."

Protection from other programs: But not the programmer. I wanted an OS that would prevent another program from taking the system down, yet allow us the power to do what we wanted, if we knew how. This could be a function of the hardware as well as the software, and the techniques may change based on your platform and processor of choice.

Use of the CPU instruction set as designed: Many languages and operating systems ported between processors tend to ignore many of the finer capabilities of the target processor. I didn't want to do this. I use the stack, calling conventions, hardware paging, and task management that is native to the Intel 32-bit x86 series of processors. Even with this goal in mind, I have attempted to isolate the API from the hardware as much as possible (for future source-code portability). You may want your operating system to run on another platform or even another processor. If it is for another processor, many of the items that I use and discuss in this book may not apply. I take a generic look at these issues in Chapters 6 and 8.

Simplicity: Keep the system as simple as possible, yet powerful enough to get the job done. I also wanted to reduce the "jargon" level and minimize the number of terse, archaic names and labels so often used in the computer industry.

A Brief Description of MMURTL

From the wish list, I put together the MMURTL operating system. I will describe it to you here so that you can see what I ended up with, based on the list. This may help you with your own list.

MMURTL (pronounced like the girl's name *Myrtle*) is a 32-bit, message-based, multitasking, real-time operating system designed around the Intel 80386 and 80486 processors on the PC Industry Standard Architecture (ISA) platforms. The name is an acronym for Message-based, MUltitasking, Real-Time kerneL. If you don't like my acronym, make up your own. But I warn you: Acronyms are in short supply in the computer industry.

MMURTL is designed to run on most 32-bit ISA PCs in existence. Yes, this means it will even run on an 80386SX with one megabyte of RAM (although I recommend 2MB). Then again, it runs rather well on a Pentium with 24MB of RAM, too. If you intend to run MMURTL or to use any of the tools I included, see the "Hardware Requirements" section later in this chapter.

MMURTL is not designed to be compatible with today's popular operating systems, nor is it intended to directly replace them. It is radically different, in that simplicity was my most important goal.

If you don't want to start from scratch or you have an embedded application for MMURTL, the tools, the code, and the information you need are all contained here so you can make MMURTL into exactly what you want. Sections of the source code will, no doubt, be of value in your other programming projects as well.

Uses for MMURTL

If you are not interested in writing your own operating system (it *can* be a serious time-sink) and you want to use MMURTL as is (or with minor modifications), here's what MMURTL can be used for:

- MMURTL can be considered a general-purpose operating system, and dedicated vertical applications are an ideal use.
- It is an ideal learning and reference tool for programmers working in 32-bit environments on the Intel 32-bit, x86/Pentium processors, even if they aren't on ISA platforms or using message-based operating systems.
- MMURTL can be the foundation for a powerful, dedicated communications system, or can be used as a dedicated interface between existing systems. The real-time nature of MMURTL makes it ideal for handling large numbers of interrupts and communications tasks very efficiently.
- Dedicated, complex equipment control is not beyond MMURTL's capabilities.

- Vertical applications of any kind can use MMURTL where a character-based color or monochrome VGA text interface is suitable.
- MMURTL would also be suitable for ROM-based embedded systems (with a few minor changes). One of my goals was to keep the basic system under 192K (a maximum of 128K of code and 64K of data), excluding dynamic allocation of system structures after loading. The OS could be moved from ROM to RAM, and the initialization entry point could be jumped to. If you're not worried about memory consumption, expand it to your heart's desire.
- In the educational field, MMURTL can be used to teach multitasking theory. The heavily commented source code and in-depth theory covered in this book make it ideal for a learning/teaching tool or for general reference.
- And of course, MMURTL can be the basis or a good starting point for your very own microcomputer operating system.

Similarities to Other Operating Systems

MMURTL is not really like any other OS that I can think of. Its closest relative is CTOS (Unisys), but that is only a distant cousin, and only so because of the similar kernel-messaging schemes. Your creation also will take on a personality of its own, I'm sure.

The flat memory-model implementation is a little like that of OS/2 2.x (IBM), but not enough alike that you could ever confuse the two.

Some of the public and internal calls may resemble UNIX a little (or even MS-DOS), but still, it is not at all the same.

The file system included with MMURTL uses the MS-DOS disk structures, but only out of convenience. This certainly wasn't because I liked the design or filename limitations. There are some 100 million+ disks out there with MS-DOS FAT formats. This makes it easy to use MMURTL, and more importantly, it eases the development burden. No reformatting or partitioning your disks—you simply run the MMURTL OS loader from DOS, and the system boots. Rebooting back to DOS is a few keystrokes away—if you also want to run DOS, that is. MMURTL has its own loader to boot from a disk, which eliminates the need for MS-DOS altogether. If you don't want to write a completely new operating system, some serious fun can be had by writing a much better file system than the one I have included.

Hardware Requirements

This section describes the hardware required to run MMURTL (as included) and to work with the code and tools I have provided. If you intend to build on a platform other than that described here, you can ignore this section.

The hardware (computer motherboard and bus design) must be PC ISA (Industry Standard Architecture) or EISA. Other 16 or 32-bit bus designs may also work, but minor changes to specialized interface hardware may be required.

The processor must be an Intel 80386, 80486, Pentium, or one of the many clones in existence that execute the full 80386 instruction set. These include Cyrix, AMD, IBM, and other clone processors.

VGA video text (monochrome or color) is required. MMURTL accesses the VGA text memory in color text mode (which is the default mode set up in the boot ROM if you have a VGA adapter).

One megabyte of RAM is required; 2MB (or more) are recommended. MMURTL will handle up to 64MB of RAM. MMURTL itself uses about 300K after complete initialization.

The floppy disk controller must be compatible with the original IBM AT controller specification. (Most are.) Both 5.25 inch and 3.5 inch are supported.

The hard-disk controller must be MFM or IDE (Integrated Drive Electronics). IDE is preferred. Some RLL controllers will not work properly with the current hard-disk device driver.

A standard 101-key AT keyboard and controller (or equivalent) is required.

The A20 address-gate line must be controllable, as documented in the IBM-PC AT hardware manual, via the keyboard serial controller. Most are, but a few are not. If yours isn't, you can change the code as needed, based on your system manufacturer's documentation (if you can get your hands on it).

8250, 16450, or 16550 serial controllers are required for the serial ports (if used).

The parallel port should follow the standard IBM AT parallel port specification. Most I have found exceed this specification and are fully compatible.

MMURTL does *not* use the BIOS code on these machines. Full 32-bit device drivers control all hardware. Hence, BIOS types and versions do not matter.

2

General Discussion and Background

This section discusses the actual chore of writing an operating system, as well as what may influence your design. I wrote an operating system over a period of almost five years. Things change a lot in the computer industry in five years, but I'm happy to find that many of the concepts I was interested in have become a little more popular (small tight kernels, simplified memory management, client/server designs, and so on). Reading this chapter will help you understand why I did things a certain way; my methods may not seem obvious to the casual observer. I also give you an idea of the steps involved in writing an operating system.

Where Does One Begin?

Where does one begin when setting out to write a computer operating system? It's not really an easy question to answer. The history of MMURTL and an explanation of why certain decisions were made, however, can be valuable to you if you intend to dig into the source code, write your own operating system, or even just use MMURTL the way it is.

A friend of mine and I began by daydreaming over brown-bag lunches in a lab environment. We bantered back and forth about what we would build and design into a microcomputer operating system if we had the time and inclination to write one. This lead to a fairly concrete set of design goals that I whittled down to a size I thought I could accomplish in two years or so. That was 1989. As you can see, Fud's Law of Software Design Time Estimates came into play rather heavily here. You know the adage (or a similar one): Take the time estimate, multiply by two, and add 25 percent for the unexpected. That worked out just about right. Five years later, here it is. I hope this book will shave several years off of your development effort.

In Chapter 1, "Overview," I covered what my final design goals were. I have tried to stick with them as closely as possible without scrimping on power or overcomplicating it to the point of it becoming a maintenance headache. Far too often, a piece of software becomes too complex to maintain efficiently without an army of programmers. Usually software also tends to expand to fill all available memory (and more!). You would not let this happen.

You Are Where You Were When

The heading above sounds a little funny (it was the title of a book that was popular a few years back, and it is borrowed with great respect, I might add). It does, however, make a lot of sense. MMURTL's design is influenced by all of my experiences with software and hardware, my schooling, my reading of various publications and periodicals, and what I have been introduced to by friends and coworkers. Even a course titled Structured Programming with FORTRAN 77, which I took online from the University of Minnesota in 1982, has had an effect. I have no doubt that your background will have a major effect on your own design goals for such a system. Borrow from mine, add, take away, improve, and just make it better if you have the desire. What I don't know fills the Library of Congress quite nicely, and "where you were when"

will certainly give you a different perspective on what you would want in your own system. Every little thing you've run across will, no doubt, affect its design.

My first introduction to computers was in 1976, on board a ship in the U.S. Coast Guard. I was trained and tasked to maintain a Honeywell DDP-516 computer system. It had a "huge" 8K of hand-wound core memory and was as big as a refrigerator (including ice maker). It was a 32-bit system, and it was very intriguing (even though punching in hundreds of octal codes on the front panel made my fingers hurt). The term "register" has a whole new meaning when you are looking for a bit that intermittently drops out in a piece of hardware that size. It whet my whistle, and from there I did everything I could to get into the "computer science" end of life. I bought many little computers (remember the Altair 8800 and the TRS-80 Model I? I know, you might not want to). I spent 20 years in the military and took computer science courses at schools when time allowed.

In 1981 I was introduced to a computer system made by a small company named Convergent Technologies (since swallowed by Unisys, a.k.a. Burroughs and Sperry, merged).

That system was an Intel 8086-based system and ran a multitasking, real-time operating system from the very start (called CTOS). Imagine an 8086-based system with an eight-inch, 10MB hard drive costing $20,000. This was a good while before I was ever introduced to PC-DOS or MS-DOS. In fact, the convergent hardware and the CTOS operating system reached the market at about the same time.

After I got involved with the IBM PC, I kept looking for the rich set of kernel primitives I had become accustomed to for interprocess communications and task management, but, as you well know, there was no such thing. This lack was disappointing because I wanted a multitasking OS running on a piece of hardware I could afford to own! In this early introduction to CTOS, I also tasted "real-time" operation in a multitasking environment. The ability for a programmer to determine what was important while the software was running, and also the luxury of changing tasks based on outside events in a timely manner, was a must. There was no turning back. Nothing about my operating system is very new and exciting. The messaging is based on theory that has been around since the early 1970s. It is just now becoming popular. People are just now seeing the advantages of message-based operating systems. Message-based, modular micro kernels seem to be coming of age.

The Development Platform and the Tools

My desire to develop an operating system to run on an inexpensive, popular platform, combined with my experience on the Intel x86 processors, clinched my decision about where to start: It would definitely be ISA 386-based machines (the 486 was in its infancy). The next major decision was the software development environment. Again, access and familiarity played a role in deciding the tools that I would use. MS-DOS was everywhere, like a bad virus. I had two computers at that time (a 386DX and a 386SX) still running MS-DOS and not enough

memory or horsepower to run UNIX or even OS/2, which at the time was still 16 bits anyway. I thought about other processors. I purchased technical documentation for Motorola's 68000 series, National Semiconductor's N80000 series, and a few others. The popularity of the Intel processor and the PC-compatible systems was, however, a lure that could not be suppressed.

As you may be aware, one of the first tools required on any new system is the assembler. It's the place to start. Utilities and compilers are usually the next things to be designed or ported to a system. Not having to start from scratch was a major advantage. Having assemblers available for the processor from the start made my job a lot easier. Of course, in the process of using these other tools, I found some serious bugs in them when used for 32-bit development. The 32-bit capabilities were added to these assemblers to take advantage of the capabilities of the new processors, but not many people were using them for 32-bit development five or six years ago on the Intel-based platforms. Operating system developers, people porting UNIX systems, and those building 32-bit DOS extenders were the bulk of the users.

I began with Microsoft's assembler (version 5.0) and later moved to 5.1. Two years later I ran into problems with the Microsoft linker and even some of the 32-bit instructions with the assembler; these problems pretty much forced the move to all-Borland tools. Then those were the 2.x assembler and 2.x linker. I'm not a Borland salesman, but the tools just worked better.

There were actually several advantages to using MS-DOS in real mode during development. Being in real mode and not having to worry about circumventing protection mechanisms to try out 32-bit processor instructions actually lead to a reduced development cycle in the early stages.

The Chicken and the Egg

The egg is first... maybe. Trying to see the exact development path was not always easy. In fact, at times it was a quite difficult. I wanted the finished product to have its own development environment—including assembler, compiler, and utilities—yet I needed to have them in the MS-DOS environment to build the operating system. I wanted eventually to port them to the finished operating system. How many of you have heard of or remember ISIS or PLM-86? These were some of the very first environments for the early Intel processors—all but ancient history now.

It was definitely a "chicken-and-egg" situation. You need the 32-bit OS to run the 32-bit tools, yet you need the 32-bit tools to develop the 32-bit OS. It can be a little confusing and frustrating. Again, MS-DOS and the capability for the Intel processors to execute some 32-bit instructions in real mode (while still in 16-bit code segments) made life a lot easier. I could actually experiment with 32-bit instructions without having to move the processor into protected mode and without having to define 32-bit segments. Memory access was limited, but the tools worked. I even put a 32-bit C compiler in the public domain for MS-DOS as freeware. It was limited to the small-memory model, but it worked nonetheless.



I really do hate reinventing the wheel; however, in the case of having the source code to an assembler I was comfortable with, I really had no choice. Some assemblers were available in the "public domain" but came with restrictions for use, and they didn't execute all the instructions I needed anyway. I even called and inquired about purchasing limited source rights to some tools from the major software vendors. Needless to say, the cost was prohibitive. (They mentioned dollar figures with five and six digits in them.... Now *that's* a digital nightmare.) This lead to the development of my own 32-bit assembler. Developing this assembler was, in itself, a major task. That set me back four months, at least; but it gave me some serious insight into the operation of the Intel processor. The source code to this assembler (DASM) is included with this book on the CD-ROM. The prohibitive costs also led to the development of a 32-bit C compiler, a disassembler, and a few other minor utilities. The C compiler produces assembly language, which is fed to the assembler. There is no linker. That's right, no linker. It's simply not needed. To fully understand this, you will need to read Chapter 28, "DASM: A 32-Bit Intel-Based Assembler."

Assembly language has its place in the world, but there is nothing like a high-level language to speed development and make maintenance easier. I actually prefer Pascal and Modula over C from a maintenance standpoint. I was into Pascal long before I was into C. I think many people started there. Those who must maintain code or read code produced by others understand the advantages, if they get past the "popularity concept." C still can be made almost unreadable by the macros and some of the language constructs, even though ANSI standards have fixed much of it. C's big strength lies in its capability to be ported easily. Most of the machine-dependent items are left completely external to the language in libraries that you may, or may not, want or need. I don't like the calling conventions used by C compilers—for example, parameters passed from right to left, and the fact that the caller cleans the stack. This is obviously for variable length parameter blocks, but it leads to some serious trouble on occasion. Nothing is harder to troubleshoot than a stack gone haywire. MMURTL uses the Pascal conventions (referred to as PLM calling conventions in the early Intel days). Parameters are passed left to right, and the called function cleans the stack. I mention this now to explain my need to do some serious C compiler work.

I started with a public-domain version of Small-C (which has been around quite a while) and took a good look at Dave Dunfield's excellent Micro-C compiler, but I pretty much built from scratch. Some very important pieces are still missing from the compiler, but another port is in the works. The details of CM-32 (C-Minus 32) are discussed in Chapter 29, "CM32: A 32-Bit C Compiler."

Early Hardware Investigation

I taught computer electronics technology for two years at the Computer Learning Center of Washington, DC. I have worked for years with various forms of hardware, including most of the digital logic families (ECL, TTL, and so on). I dealt with both the electrical and timing

characteristics, as well as the software control of these devices. The thought of controlling all of the hardware (timers, DMA, communications controllers, and so on) in the machine was actually a little exciting. The biggest problem was accumulating all of the technical documentation in adequate detail to ensure it was done right. If you've ever looked though technical manuals that are supposed to give you "all" of the information you need to work with these integrated circuits, you know that it's usually *not* everything you need.

You will need to accumulate a sizable library of technical documentation for your target hardware platform if you intend to write an operating system or even to port MMURTL to it.

I'm not going to tell you I did some disassembly of some BIOS code; I plead the "Fifth" on this. Besides, IBM published theirs. BIOS code brings new meaning to the words *spaghetti code.* It's not that the programmers who put it together aren't top quality; it's the limitations of size (memory constraints) and possibly the need to protect trade secrets that create the spaghetti-code effect. Following superfluous JMP instructions is not my idea of a pleasant afternoon.

I used many different books, documents, articles, and technical manuals throughout the early development of my operating system. The IBM PC-AT and Personal System/2 Technical Reference manuals were a must. They give you the overall picture of how everything in the system ties together. Computer chip manufacturers, such as Chips & Technologies, supply excellent little documents that give even more specific details on their particular implementation of the AT series ISA logic devices. These documents are usually free.

All of the ISA-type computer systems use VLSI (Very Large Scale Integration) integrated circuits that combine and perform most of the functions that were done with smaller, discreet devices in the IBM AT system. I keep referring to the IBM AT system even though it was only an 80286 (16-bit architecture). I do so because its bus, interface design, and ancillary logic design form the basis for the 32-bit ISA internal architecture. In fact, as you may understand, the external bus interface on the ISA machines is still 16 bits. It is tied to the internal bus, which is 32 bits for the most part. SX machines still have 16-bit memory access limitations, but this is transparent to the programmer, even to the operating system writer. Many bus designs now out take care of this problem (PCI, EISA, and so on).

This made the IBM designs the natural place to start. It was proven that PC-style computer manufacturers who deviated from this hardware design to the point of any real low-level software incompatibility usually weren't around very long. Even machines that were technically superior in most every respect died horrible deaths. (Remember the Tandy 2000?)

MS-DOS really didn't have to worry about the specific hardware implementation because the BIOS software/firmware writers and designers took care of those details. As you may know, however, the BIOS code is not 32-bit (not even on 32-bit machines). It is also not designed to be of any real value in most true multitasking environments. A variety of more recent operating systems go directly to much of the hardware just to get things done more efficiently, anyway (for example, OS/2 Version 2.x, Windows Version 3.x, all implementations of UNIX). I knew from the start that this would be a necessity. "Thunking" was out of the question.

NOTE

Thunking is what occurs when you interface 32- and 16-bit code and data. An example would be making a call to a 32-bit section of code *from* a 16-bit code segment. This generally involves manipulating the stack or registers, depending on the calling conventions used in the destination code, as well as storage on the stack. Some processors (such as the Intel 32-bit x86 series) force a specific element size for the stack.

Much of the early development was a series of small test programs to ensure I knew how to control all of the hardware, such as Direct Memory Access (DMA), timers, and the Priority Interrupt Controller Unit (PICU). This easily consumed the first six months. During this time, I was also building the kernel and auxiliary functions that would be required to make the first executable version of an operating system. The second six months was spent moving in and out of protected mode and writing small programs that used 32-bit segments. If you use Intel-based platforms, this is a year you won't have to go through, because of what you get on this book's CD-ROM.

My initial thought on memory management centered on segmentation, but not on a fully segmented model—two segments per program and two for the OS, all based on their own selectors. This would have given me a zero-based addressable memory area for all code and data access for every program. It would have appeared as a flat memory space to all the applications, but it presented several serious complications. The largest complication was that all memory access would be based on 48-bit (far) pointers. The thought was rather sickening, and some initial testing showed that it noticeably slowed down program execution. When the 386/486 loads a segment register, a speed penalty is paid because of the hardware overhead (loading shadow registers, and so on). I was after a "kinder, gentler" environment, anyway. The segmented memory idea was canned, and I went straight to fully paged memory allocation for the operating system. It has turned out very nicely, as you will see.

One other consideration on the memory model is *address aliasing*. If two programs in memory must exchange data and they are based on different selectors, you must create an alias selector to be used by one of the programs. In a paged system, aliasing is also required, but it's not as complicated. In Chapter 5, "Memory Management," I discuss several options you have.

To get a simple model up and running took almost two years. This simple model could allocate and manage memory, and it had only the most rudimentary form of messaging (Send and Wait).

There was no loader, no way to get test programs into the system. There wasn't even a file system! Test programs were actually built as small parts of the operating system code, and the entire system was rebuilt for each test. This was time-consuming but necessary to prove some of my theories and hunches. All of the early test programs were in assembler, of course. If you start from scratch on a different processor, you will go through similar contortions. If you

intend to use the Intel processors and start with MMURTL or the information presented in this book, you'll be ahead of the game.

The Real Task at Hand (Pun Intended)

All of the hardware details aside, my original goal was to create a multitasking operating system that was not a huge pig, but instead a lean, powerful, easy-to-use system with a tremendous amount of documentation. I had to meet my design goals, and the primary emphasis was on the kernel and all that that implies. If you write your own system, don't lose sight of the forest for the trees. I touched on what I wanted in Chapter 1, but putting it on paper and getting it to run are two very different things.

A Starting Point

If you have truly contemplated writing an operating system, you know that it can be a very daunting task. I wish there would have been some concrete information on really where to start, but there wasn't. The books I have purchased on operating system design gave me a vast amount of insight into the major theories and overall concepts concerning operating system design, but they just didn't say, "You start here, at the beginning of the yellow brick road."

In addition, it is difficult for me to advise you because I don't know exactly where you're coming from, how much knowledge you have, or how much time you really have to invest. But I'll try.

The theory behind what you want is the key to a good design. Know your desired application. If you are writing an operating system for the learning experience (and many people do), the actual application may not be so important, and you may find that you really have something when you're done with it—or at least when you get it working; you'll *never* really be done with it. Your target application will help you determine what is important to you. For example, if you are interested in embedded applications for equipment control, you may want to concentrate on the device and hardware interface in more detail. I was really interested in communications. You'll probably be able to see that as you read further in this book.

The easy way out would be to take a working operating system and modify it to suit you. I have provided such an operating system with this book. It is, however, by no means the only operating system source code available. Of course, I'm not going to recommend any others to you. If you choose to modify an existing operating system, study what you have and make sure you completely understand it before making modifications. I say this from experience. Many times I have taken and "gutted" a piece of software that I didn't fully understand, only to find out that it had incorporated many of the pieces I needed—I simply hadn't understood them at the time. If this has never happened to you, I envy you.

The Critical Path

Along your journey, you'll find that there are major items to be accomplished, and there will also be background noise. If you've ever done any large-scale project management, you know that certain items in a project must work or that certain must be proved in order for you to continue along the projected path, also called the *critical path.* I'll give you a list of these items in the approximate order that I found were required while I wrote my system.

1. Decide on your basic models. This includes the tasking model, memory model, and basic programming interface. You may need to choose a hardware platform before making your memory model concrete. Chapter 3, "The Tasking Model," and Chapter 4, "Interprocess Communications," will give you plenty of food-for-thought on the tasking model.
2. Select your hardware platform. It may be RISC, it may be CISC, but whatever you decide on, get to know it very well. Try to be sure that it will still be in production when you think you'll be done with your system. This may sound humorous, but I'm very serious. Things change rapidly in the computer industry.
3. Investigate your tools. Get to know your assemblers, compilers, and utilities. Even though you may think the programmers who wrote those tools know all there is to know about them, you will no doubt find bugs. I blamed myself several times for problems I thought were mine but were not.
4. Play with the hardware to be sure you understand how to control it. This includes understanding some of the more esoteric processor commands. Operating systems use processor instructions and hardware commands simply never used by applications and even device drivers. Document the problems, the discoveries—anything you think you'll need to remember. Chapter 6, "The Hardware Interface," gives you some hints on this. With so many platforms out there, hints are about all I *can* give you, unless you go with the same hardware platform I did.
5. Go for the kernel. If you have to build a prototype running under another operating system, do it. Doing so will be worth the time spent. You can then "port" the prototype.
6. Memory management is next. This can be tricky because you may be playing equally with hardware and software. The level of complexity, of course, will depend on the memory model you choose. In Chapter 5, I give you some options. I tell you the easiest option, and I also dive into some more challenging possibilities.
7. If you've gotten past the fun parts (kernel and memory), the rest is real work. You can take these pieces—everything from device drivers, garbage collection, and overall program management—as you see fit.

Working Attitude

Set your goals, but don't set them too high. I had my wish list well in advance. Creating a realistic wish list may be one of the harder jobs. Don't wish for a Porsche if what you need is a good, dependable pickup truck. The Porsche is nice…but think of the maintenance (and can you haul firewood in it?).

Make sure your tools work. Make certain you understand their capabilities and limitations. I spent a good six months playing with assemblers, compilers, and utilities to be sure I was "armed and dangerous." After I was satisfied, I moved on from there. As I mentioned previously, I actually had to switch assemblers in midstream. Problems with programming tools were something I couldn't see in advance. The tools you should have include documentation for the hardware and plenty of "theory food" for the engine (your brain).

Work with small, easily digestible chunks. Even if you have to write four or five small test programs to be sure that each added piece functions correctly, it's worth the time. Nothing is worse than finding out that the foundation of your pyramid was built on marshmallows. When writing my memory-allocation routines, I must have written 30 small programs (many of which I did not intend to act correctly) to be sure I had what I thought I had. I'll admit that I had to do some major backtracking a time or two.

Document as you go. I lay out my notes for what I want, I write the documentation, and then I write the code. Sounds backward? It is. I wrote major portions of my programming documentation before the system was even usable, but doing so gave me great insight into what a programmer would see from the outside.

Good luck. Really, it won't be luck. It will be blood, sweat, and maybe some tears, all the way.

II

Architecture and General Theory

3

The Tasking Model

This chapter discusses the tasking model and the things that have an effect on it. Most of the discussion centers on resource management and, more specifically, CPU utilization. After all, resource management is supposed to be what a computer operating system provides you.

Terms We Should Agree On

One thing the computer industry lacks is a common language, and I'm not talking about programming languages. One computer term may mean six different things to six different people. Much of the problem is caused by hype and deceptive advertising, but some of it also can be attributed to laziness. (Yes, I admit some guilt here too.) People use terms in everyday conversation without really knowing what they mean or at least without coming to agreement on them. Before I begin a general discussion or analyze your options for a tasking model, I have to define some terms just to make sure we're on common ground. You may not agree with the definitions, but I hope you will accept them while you're reading this book. The terms you already should be familiar with are as follows:

- Task
- Kernel
- Preemptive multitasking

Tasks

A *task* is an independently scheduled thread of execution. You can transform the same 50 processor instructions in memory into 20 independent tasks. When a task is suspended from execution, its hardware and software states are saved while the next task's state is restored and then executed. A single computer program can have one or more tasks. Some operating systems call a task a "process," whereas others call it a "thread." In most instances, I call it a task. Some documentation, however, may use other words that mean the same thing, such as "interprocess communications," which means communications between tasks.

Generally, I have seen the term thread applied to additional tasks all belonging to one program. But this is not cast in stone either. It really depends on the documentation with the operating system you are using. If you write your own, be specific about it. Tell the programmers what you mean.

For *task*, I use the same definition as in the Intel *80386 System Software Writer's Guide* and the *80386* and *80486 Programmer's Reference* manuals. This is convenient if you want to refer to these documents while reading this book. I literally destroyed two copies of each of these books while writing MMURTL (pages falling out, coffee and soda spilled on them, more highlighter and chicken scratching than you can imagine). The Intel-based hardware is by far the most available and least expensive, so I imagine most of you will be working with it.

The Kernel

The term *kernel* is not disputed very much, but it is used a lot in the computer industry, so it is important to understand. Apparently a lot of people use the term and they don't even know what it is. The *kernel* of an operating system is the code that is directly responsible for the tasking model. In other words, it is responsible for the way the CPU's time is allocated. MMURTL has a very small amount of code that is responsible for the tasking model. In its executable form, it probably isn't much more than two or three kilobytes of code. This makes it a very small kernel. This might incline me to use the techno-buzz-word "microkernel" if I were into buzzwords. The operating system functions that directly affect CPU tasking are called kernel primitives. This is not because they're from the days of Neanderthal computing, but because they are the lowest-level entry points into the kernel code of the operating system.

Preemptive Multitasking

Preemptive multitasking is a hotly disputed term (or phrase) in the industry. It shouldn't be, but it is. In computing, the word *preempt* simply means to interrupt something that is currently running, but doesn't expect the interrupt. There are many ways to do this in a computer operating system. One type of preemption that occurs all the time is hardware interrupts. I'm not including them in this discussion because they generally return control of the processor back to the same interrupted task, and non-multitasking systems use them to steal time from programs and simulate true multitasking (for example, MS-DOS TSRs—terminate-and-stay-resident programs). The preemptive multitasking I'm talking about and that most people refer to is the capability for an operating system to equitably share CPU time between several well-defined tasks currently scheduled to run on the system (even tasks that may be a little greedy with the CPU).

My definition of preemptive multitasking is as follows: If an operating system has the capability to stop a task while running before that task was ready to give up the processor, to save its state, and then start another task running, it's preemptive. How and when it does this is driven by its tasking model. An operating system that you use may be preemptive, but its tasking model may not be the correct one for the job you want to accomplish (or you may not be using it correctly)—hence, all the grumbling and heated discussions I see on the online services.

When you determine exactly what you want for a tasking model, you will see the rest of your operating system fit around this decision.

Resource Management

An operating system manages resources for applications and services. This is its only real purpose in life. Consider your resources when planning your operating system. The resources include CPU time sharing (tasking model), memory management, and hardware (timers, DMA, interrupt controller units, and so on). These are the real resources. Everything else uses these

resources to accomplish their missions. Input/output (I/O) devices, including video, keyboard, disk, and communications equipment, make use of these basic resources to do their jobs. These resources must be managed properly for all these things to work effectively in a multitasking environment.

One of my major goals in designing the MMURTL operating system was simplicity. To paraphrase a famous scientist (yes, good old Albert), "Everything should be as simple as possible, but not simpler." (Otherwise it wouldn't work!) I have attempted to live up to that motto while building my system as a resource manager. One of my friends suggested the motto "Simple software for simple minds," but, needless to say, it didn't sit very well with me. Managing these resources in a simple yet effective fashion is paramount to building a system you can maintain and work with.

Task Management

Tasks begin. Tasks end. Tasks crash. All of these things must be taken into consideration when designing a task-management scheme. Obviously, keeping track of tasks is no simple "task." In some operating systems, each task may be considered a completely independent entity. It may have memory assigned to it, and it may even run after the task that created it is long gone. On the other hand, it may be completely dependent on its parent task or program. How tasks are handled and how resources are assigned to these tasks was easy for me to decide, but it may not be so easy for you.

If you decide that a single task may exist on its own two feet, then you must make sure that your task-management structures and any interprocess communications mechanisms take this into account. When you get to Part IV of this book ("The Operating System Source Code"), take a good look at the TSS (Task State Segment). This is where all of the resources for my tasks are managed. You may need to expand on this structure quite a bit if you decide that each and every task is an entity unto itself.

Single User Versus Multiuser

One of the determining factors on how your tasks are managed may be whether or not you have a true *multiuser* system. The operating system included with this book is not multiuser. It is not a UNIX clone by any stretch of the imagination. Nor did I want it to be. In a system that is designed for multiple terminals/users, tasks may be created to monitor serial ports or even network connections for logons, execute programs as new users' desire, and then still be there after all the child processes are long gone (for the next user).

Probably the best way to get this point across is to describe what I decided, and why. From this information you will be able to judge what's right for you.

I have written many commercial applications that span a wide variety of applications. Some are communications programs (fax software, comms programs), some deal with equipment control (interaction with a variety of external hardware), and some are user-oriented software.

In each case, when I look back at the various jobs this software had to handle, each of the functions dealt with a specific well-defined mission. In other words, each program had a set of things that it had to accomplish to be successful (satisfy the user). When I wrote these programs, the ability to spawn more tasks to handle requirements for simultaneous operations came in very handy. But in almost all of these systems, when the mission was accomplished, or the user was done, there seemed to be no purpose in having additional tasks that continued to run. The largest factor was that I had a CPU in front of me, and each of the places these programs executed also had a CPU—in other words, *single user, multitasking.*

My operating system is designed for microcomputers rather than a mini or mainframe environment (although this line is getting really blurry with the unbelievable power found in micro CPUs these days). I deal with one user at a time. You may want to deal with many. Your decision in this matter will have a large effect on how you manage tasks or whole programs that are comprised of one or more tasks.

The Hardware State

When a task switch occurs, the task that is running must have its hardware context saved. The hardware context in its simplest form consists of the values in registers and processor flags at the time the task stopped executing. In the case of hardware interrupts, this is usually a no-brainer. The CPU takes care of it for you. If you must do it yourself, there are instructions on some processors to save all of the necessary registers onto the stack. Then all you have to do is switch stacks. When you restore a task, you switch to its stack and pop the registers off.

The hardware state may also consist of memory context if you are using some form of hardware paging or memory management. This will depend on the processor you are using and possibly on external paging circuitry. You will have to determine this from the technical manuals you accumulate for your system.

The Software State

The software state can be a little more complicated than the hardware state and depends on how you implement your task management. The software state can consist of almost nothing, or it may be dozens of things that have to be saved and restored with each task switch.

An example of an item that would be part of the software state is a simple variable in your operating system that you can check to see which task is running. When you change tasks, you change this variable to tell you which task is currently executing. The number of things you have to save may also depend on whether or not the processor assists you in saving and restoring the software state. Some processors (such as the Intel series) let you define software-state items in a memory structure that is partially managed by the operating system. This is where the hardware state is saved.

It's actually easier than it sounds. You can define a structure to maintain the items you need for each task, then change a pointer to the current state structure for the task that is executing.

CPU Time

CPU time is the single most important resource in a computer system. "It's 11:00 p.m. Do you know where your CPU is?" A funny question, but an important one. What instructions are executing? Where is it spending all its time? And why?

When an application programmer writes a program, he or she generally assumes that it's the only thing running on the computer system. My first programs were written with that attitude. I didn't worry about how long the operating system took to do something, unless it seemed like it wasn't fast enough. Then I asked, "What is that thing doing?" When I used my first multiuser system, there were many times that I thought the system had died and gone to computer heaven. Yet, it would seem to return at its leisure. What *was* that thing doing? It was being shared with a few dozen other people staring at their terminals just like I was. We were each getting a piece of the pie, albeit not a very big piece. Microcomputers spoiled me. I had my very own processor, and I could pretty much figure out what it was doing most of the time. Many of the programs I wrote didn't require multitasking, and a single thread of instructions suited me just fine. It was in 1980 that I wrote my first time slicer, which was built into a small program on an 8-bit processor. It was crude, but effective. I knew exactly how long each of my two tasks were going to run. I also knew exactly how many microseconds it took to switch between them. There were so few factors to be concerned with on that early system, it was a breeze.

In a multitasking operating system, ensuring that the CPU time is properly shared is the job of kernel and scheduling code. This will be the heart and brains of the system.

Single Tasking

What if you don't want a multitasking system? This is entirely possible, and in some cases a single-threaded system may be the best solution. In such a case, you basically have a hardware manager. In a single-tasking operating system, management of CPU time is easy. The only thing for the programmer to worry about is how efficient the operating system is at its other jobs, such as file handling, interrupt servicing, and basic input/output. Many methods have been devised to allow single-tasking operating systems to share the CPU among pseudo-tasks. These have been in the form of special languages (Co-Pascal), and it can also be done with mini-multitasking kernels you include in your programs. These forms of CPU time sharing suffer from the disadvantage that they are not in complete control of all resources. They can, however, do an adequate job in many cases.

You may also be aware that multitasking is simulated quite often in single-tasking systems by stealing time from hardware interrupts. This is done by tacking your code onto other code that is executed as a result of an interrupt.

Multitasking

I categorize forms of multitasking into two main groups: *cooperative* and *preemptive*. An operating system's tasking model can be one of these, or a combination of the two. In fact, many operating-system tasking models are a combination. There are other important factors that apply to these two types of multitasking, which I cover later in this chapter.

The major point that you must understand here is that there are really only two ways a task switch will occur on *any* system: A task that has the processor gives it up, or it is preempted (that is, the processor is taken away from the task).

Cooperative Multitasking

In a solely cooperative environment, each task runs until it decides to give up the processor. This can be for a number of reasons, but usually it is either to wait for a resource that is not immediately available or to pause when it has nothing to do, if the system allows this (a sleep or delay function). An important point here is that in a cooperative system, programs (or, more precisely, the programmer) may have to be aware of things that must be done to allow a task switch to happen. This implies that the programmer does part of the scheduling of task execution. This is in the form of the task saying to the operating system, "Hey—I'm done with the CPU for now." From there, the operating system decides which of the tasks will run next (if any are ready to run at all). It may not always be obvious when it happens, however. For instance, a program that is running makes an operating-system call to get the time of day. The operating system may be written in such a fashion that this (or any call into the operating-system code) will suspend that task and execute another one.

I'm sure you've run across the need to delay the execution of your program for a short period of time, and you have used a library or system call with a name such as `sleep()` or `delay()`. With these calls, you pass a parameter that indicates how long the program should pause before resuming execution. This may be expressed in milliseconds, tenths of seconds, or even full seconds. If the operating system uses this period of time to suspend your task and execute another, you are cooperating with the operating system. It is then making wise use of your request to delay or sleep. In many single-tasking systems, a call of this type may do nothing more than loop through a series of instructions or continually check a timer function until the desired time has passed. And yes, that's a waste of good CPU time if something else can be done with it.

There is a major problem with a fully cooperative system. If a programmer writes code that doesn't give up the CPU (which means he doesn't make an operating-system call that somehow gets to the scheduler), he then monopolizes the CPU. If this isn't the task that the user is involved with, the screen and everything else looks frozen. Meanwhile, there's some task in there calculating pi to a zillion digits, and the computer looks dead for all practical purposes.

Because such "CPU hogs" are not only possible but inevitable, it is a good idea to have some form of a preemptive system, with the ability to cut a task off at the knees, if necessary.

Preemptive Multitasking

At the opposite end of the spectrum from a fully cooperative system is a *preemptive* system. This is an operating system that can stop a task dead in its tracks and start another one. I defined *preemptive* at the beginning of this chapter. To expand on that definition, when a task is preempted, it doesn't expect it. It does nothing to prepare for it. This can occur between any two CPU instructions that it is executing (with some exceptions discussed later).

As I also mentioned earlier, this actually happens every time a hardware interrupt is activated. However, when a hardware interrupt occurs on most hardware platforms, the entire "state" of your task is not saved. Most of the time, it's only the hardware state, which includes registers, and maybe memory context, or a few other hardware-related things. Also, after the interrupt is finished, the CPU is returned to the task that was interrupted. Generally, hardware interrupts occur in order to pass information to a device driver, a program, or the operating system from an external device (hardware timer, disk drive, communications device, and so on).

One of the most important hardware interrupts and one of the most useful to an operating system designer is a *timer interrupt*. On most platforms, there is hardware (programmable support circuitry) that allows you to cause a hardware interrupt either at intervals or after a set period of time. The timer can help determine when to preempt.

Task Scheduling

Given the only two real ways a task switch can occur, you must now decide which task runs next. To refresh your memory, these two reasons are as follows:

1. A task that has the processor decides to give it up.
2. The currently running task is preempted, which means the processor is taken away from the task without its knowledge.

The next section is the tricky part. When do you switch tasks, to which task do you switch, and how long does that task get to run?

Who, When, and How Long?

The procedure or code that is responsible for determining which task is next is usually called the *scheduler*.

In fully cooperative systems, the scheduler simply determines who's next. The programmer (or the task itself) determines how long a task will run. This is usually until it willingly surrenders the processor. In a preemptive system, the scheduler also determines how long a task gets to run.

For the preemptive system, this adds the requirement of timing. How long does the current task run before you suspend it and start the next one?

One thing that you can't lose sight of when considering scheduling is that computer programs are generally performing functions for a human (the user). They are also, quite often, connected to the outside world—communicating, controlling, or monitoring. This means there is often some desire or necessity to have certain jobs performed in a timely fashion so there is synchronization with outside events. This synchronization may be required for human interaction with the computer which is not that critical, or it may be for something extremely critical, such as ensuring your coffee is actually brewed when you walk out for a cup in a maniacal daze looking like Bill the Cat.

This is where the scheduling fun really begins.

Scheduling Techniques

In a multitasking system, only one task is running at a time (excluding the possibility of multiprocessor systems). The rest of the tasks are waiting for their shot at the CPU, or waiting for outside resources such as I/O.

As you can see, there is often a desire, or even a requirement, for some form of communication between several parties involved in the scheduling process. These parties include the programmer, the task that's running, the scheduler, outside events, and maybe (if you're nice) the user.

I'm going to present several techniques as if they were used alone. This way you can picture what each technique has to offer.

Time Slicing

Time slicing means you switch tasks based on set periods of execution time for each task—"a slice of time." This implies the system is capable of preempting tasks without their permission or knowledge.

In the simplest time-sliced systems, each task will get a fixed amount of time—all tasks will get the same slice—and they will be executed in the order they were created. A simple queueing system suffices. For example, each task gets 25 milliseconds. When a task's time is up, its complete hardware and software state is saved, and the next task in line is restored and executed.

Many early multitasking operating systems were like this. They were perhaps not quite that simple, but they used a time-slicing scheduler based on a hardware timer to divide CPU time among the tasks that were ready to run.

In some systems, a simple time-slicing scheme may be adequate or at least acceptable. For example, a multitasking system that is handling simple point-of-sale and ordering transactions at Ferdinand's Fast Food and Burger Emporium may be just fine as an equitably time-sliced system. The customer (and the greasy guy waiting on you behind the counter) would never see the slowdown as the CPU in the back room did a simple round-robin between all the tasks for the terminals. This would even take care of the situation where an employee falls asleep on the terminal and orders 8,000 HuMonGo Burgers by mistake. The others may see a slowdown, but it won't be on their terminals. (The others will be flipping 8,000 burgers.)

Simple Cooperative Queueing

The very simplest of cooperative operating-system schedulers could use a single queue system and execute the tasks on a first-come/first-served basis. When a task indicates it is done by making certain calls to the operating system, it is suspended and goes to the end of the queue. This is usually not very effective, and if a user interface is involved, it can make for some pretty choppy user interaction. But it may be just fine in some circumstances.

Prioritized Cooperative Scheduling

A logical step up from a simple first-in/first-out cooperative queue is to prioritize the tasks. Each task can be assigned a number (from highest to lowest priority) indicating its importance. This gives the scheduler something to work with. It can look at the priorities of those tasks that are ready-to-run and select the best based on what the programmer thinks is important.

In a cooperative system, this means that of all the tasks that are ready to run, the one with the highest priority will be executed. This also implies that some may not be ready at all. They can be waiting for something such as I/O.

Variable Time Slicing

In a preemptive system, if you preempted a task without it first telling you it was OK to do so, you must assume that it is ready to run immediately. This also means you can't just assign simple priorities in a time-sliced system; otherwise, only the very highest would ever get to run.

In a time-sliced preemptive system, it makes more sense to increase the time allocation for more important tasks than to execute the tasks in a sequence of equal time segments based solely on a priority assignment.

The tasks that need more time can get it by asking for it or by the programmer (or user) telling the scheduler it needs it. Time allocation can even be increased by some form of signaling based on other outside events.

Time slicing is a an old technique. When combined with an adequate form of interprocess communications or signaling (as discussed in Chapter 4, "Interprocess Communications"), it can serve admirably. But it has some major drawbacks. Who determines which task gets more

time or which task will run more often? You generally can't ask the user if they need more time. (They will *always* say yes!) And the programmer can't even determine in advance how much time he or she will need, or how much time there is to go around on the system. Because it is still based on a preset period of time, there is always the possibility of being preempted before a certain important function is accomplished.

It's pretty simple to calculate how much time you'll need for a communications program running at full capacity at a known data rate. To know in advance if it will actually be used at that rate is, however, another story entirely.

Other Scheduling Factors

You know as well as I do that things are rarely as simple as they seem. Users may not know how their computers operate, but they know what they want, including dependable, clean communications and an interface that is responsive to their needs. To achieve maximum functionality, the machine must perform countless tasks about which the user may have no concept or interest.

In case you haven't noticed, computers are everywhere. They do so many things for you that you simply accept them in your everyday life. Many of these things depend on *real-time* interaction between the machine and the world to which it is connected.

Some of the best examples of real-time computing are in robotics and equipment control. A computer in charge of climate control for a large building may be a good example, but it's not dramatic enough. Instead, let's look at one that controls a nuclear reactor and the various events at a nuclear power plant. The programs that run on this computer may have some snap decisions to make based on information from outside sensors or the user. If this computer has to wait 200 or more milliseconds before performing a critical safety task, it may spell disaster. Therefore, outside events must be able to force (or at least speed up) a task switch. This implies that some form of communication is required between the sensors (for example, hardware-interrupt service routines) and the scheduler. The programmer on this type of system must also be able to indicate to the operating system that certain programs or tasks are more important than others. (Should turning on the coffee pot for the midnight shift take priority over emergency cooling-water control?)

Not many of us will ever write the software for nuclear power-plant control, but there are everyday tasks that require real-time computing to operate smoothly. One of the most volatile reactions I can think of is when a user loses data from a communications program or an important fax is garbled because buffers went a little bit too long before being emptied or filled. If there's big money involved, this reaction can approximate meltdown conditions. This *is* critical real-time stuff. Thus, when something that is out of the immediate control of the programmer (and the CPU) must be handled within a specific period of time, it requires the ability to juggle tasks on the system in real-time.

In the case of the nuclear-reactor operating system, you definitely want to be able to preempt a task, and prioritization of these tasks will be critical. If I thought my operating system would be used in this fashion, I would have spent more time on the tasking model and also on interrupt service routines. But I opted to "get real" instead.

If you have ever had the pleasure of writing communications-interrupt service routines on a system that has a purely time-sliced model, you know that it can lead to serious stress and maybe even hair loss.

Operating systems that share the CPU among tasks by just dividing up a period of time and giving each task a share are the kind that respond poorly to outside events. This is not to say that time-sliced systems have poor interrupt latency, but they have problems with unbalanced jobs being performed by the tasks. For instance, take two communications programs that handle two identical ports with two independent Interrupt Service Routines (ISRs). If the programs have an equal time slice and equal buffer sizes, but one channel is handling five times the data, the program servicing the busier port may lose data if it can't get back to the buffer in time to empty it. The buffer will overflow. This is a simple example (actually a common problem), but it makes the point. In a message-based system, the ISR can send a message to the program servicing it to tell it that the buffer is almost full. ("Hey, come do something with this data before I lose it!")

I think you can see the advantages of a priority-based system, and maybe you can even see where it would be advantageous to combine a cooperative system (where the programmer can voluntarily give up the CPU) with a preemptive time-sliced system.

Mixed Scheduling Techniques

As you can see, you have many options for scheduling techniques, as follows:

- Cooperative task scheduling
- Preemptive time slicing
- Cooperative, prioritized task scheduling
- Variable/prioritized time slicing
- Any and all combinations of the above

There are as many ways to mix the two basic types of task scheduling as there are copies of this book out there. No doubt, you have a few ideas of your own. I invite you to experiment with your own combinations. See what works for your applications. That's the only way you'll really know for sure.

In the next sections I'll present a few ways to mix them. All of these ways require some form of signaling or interprocess communications, and I discuss all of that in the next chapter. I didn't want to cloud the tasking issue with the required communications methods, even though these

methods are very important and even help define the tasking model. In addition, all of the examples in the next sections have preemptive capabilities. To be honest, I can't picture a system without these capabilities, although such systems exist.

Fully Time Sliced, Some Cooperation

In this type of model, all tasks will be scheduled for execution and will execute for a predetermined amount of time. A task may, however, give up the CPU if it's no longer needed. This would put the task back into the queue to be executed when it surrendered the CPU.

In this type of system, all tasks will get to run. You may opt to give certain tasks a larger execution time based on external factors or the programmer's desires. But even when the task has nothing to do, it would still be scheduled to execute. If such were the case (having nothing to do) the task would simply surrender the CPU again.

In a system that was designed for multiuser application sharing (such as UNIX originally was), time-slicing all the tasks is suitable (even desirable) in order to accomplish what was initially intended by the system builders. Users on a larger, solely time-sliced system will see the system slow down as the time is equitably divided between an increasing number of users or tasks. Writing communications tasks for these types of systems has always been a challenge (and that's being polite about it).

Time Sliced, More Cooperative

Building on the previous mix (some cooperation), additional signaling or messaging could be incorporated in the system to allow a task to indicate that it requires more time to execute. This could be in the form of a system call to indicate to the scheduler that this task should get a larger percentage of the available CPU time.

You can probably picture in your head how this could get complicated as several tasks begin to tell the scheduler they need a larger chunk of time. There would have to be additional mechanisms to prevent poorly written tasks from simply gobbling up the CPU. This could be in the form of a maximum percentage of CPU time allowed, based on the number of tasks scheduled. A very simple, rapid calculation would hold the reins on a CPU hog.

Primarily Cooperative, Limited Time Slicing

In this type of system, tasks are prioritized and generally run until they surrender the CPU. This means that the programmer sets the priority of the task and the scheduler abides by those wishes. Keep in mind that on a prioritized cooperative system, the task that is executing is always the one with the highest priority found on the queue.

Only in circumstances where multiple tasks of the same priority are queued to run would time slicing be invoked. These tasks would always be the highest priority tasks that are queued on the system.

This type of system would also have the capability to preempt a task in order to execute a higher priority task if one became queued (messaging from an Interrupt Service Routine).

Primarily Cooperative, More Time Slicing

This is similar to the previous model except you can slice a full range of priorities instead of just the highest priority that is running. For instance, if you have 256 priorities, 0 through 255 (the highest) may be sliced together, giving the lowest numbers (highest priorities) the greatest time slice.

The Tasking Model I Chose

I'll tell you what I wanted, then you can make your own choice based on my experiences. As I once read on a sugar packet, "Experience is a wonderful thing. It allows you to recognize a mistake when you make it again." (No, it wasn't Ferdinand's Burger Emporium.)

I wanted a priority-based, preemptive, multitasking, real-time operating system. My goal was not just to equitably share the processor among the tasks, but to use the programmer's desires as the primary factor. In such a system, prioritization of tasks is *very* important. I made MMURTL listen to the programmer. Remember the saying: "Be careful what you ask for—you may get it." Let me repeat that my goal was a *real-time* system. I never wanted to miss outside events. To me, the outside events are even more important than placating the user, and in reality, the user *is* an outside event! This put my system in the category of "primarily cooperative, limited time slicing."

As an illustration of my tasking model, MMURTL task switches occur for only two reasons:

1. The currently running task can't continue because it needs more information from the outside world (outside its own data area), such as keystrokes, file access, timer services, or whatever. In such a case, it sends a "request" or non-specific message, goes into a "waiting" state, and the next task with the highest priority executes.
2. An outside event (an interrupt) has sent a message to a task that has an equal or higher priority than the currently running task. This in itself does not immediately cause a task switch.

The timer-interrupt routine, which provides a piece of the scheduling function, monitors the current task priority, as well as checking for the highest priority task that is queued to run. When it detects that a higher priority task is in a ready-to-run state, it will initiate a task switch as soon as it finds this. If it detects that one or more tasks with priorities equal to the current task are in the ready queue, it will initiate a task switch after a set period of time. This is the only "time slicing" that MMURTL does. This has the desired effect of ensuring that the highest priority task is running when it needs to and also that all tasks of equal priority get a shot at the CPU. This is the preemptive nature of MMURTL.

In Chapter 18, "Kernel Code," and Chapter 20, "Timer Management Code," I discuss in detail how I melded these two models.

Interrupt Handling

Hardware interrupt service routines (ISRs) have always been interesting to me. I have learned two very valuable lessons concerning ISRs: Keep them short, and limit the functionality if at all possible. I bring them up in this chapter because how an operating system handles ISRs can affect the tasking model, and they will also be used for execution timing on some preemptive systems.

Certain critical functions in the operating-system kernel and even some functions in device drivers will require you to suspend interrupts for brief periods of time. The less time the better. This will usually be during the allocation of critical shared operating-system resources and certain hardware I/O functions.

With the Intel processors, you even have a choice of whether the ISR executes as an independent task or executes in the context of a task that it interrupted. The task-based ISR is slower. A complete processor task change must occur for the ISR to execute, and another one must occur to return it to the task that was interrupted. Other processors may offer similar options. Go for the fastest option. Period.

I decided that MMURTL ISRs would execute in the context of the task that they interrupted. This is a speed issue. It's faster—actually, much faster. This could have presented some protection problems, and also the possibility of memory-management headaches, but because ISRs all execute code at the OS level of protection and all programs on the system share this common OS-base addressing, the problems didn't surface. This was due to MMURTL's memory management methods. I discuss memory management in Chapter 5, "Memory Management," and MMURTL's specific methods in Chapter 19, "Memory Management Code."

An important thing you should understand is that you don't have a big choice as to when an ISR executes. It is determined by hardware that may be completely out of your control, or even possibly the user's, and they are surely out of your control most of the time.

For instance, who's to say when Mr. or Ms. User is going to press a key on the keyboard? That's right—only the user knows. Who will determine how long it actually takes to transfer several sectors from a disk device? There are too many factors involved to even prepare for it. This means that your operating system must be prepared at all times to be interrupted. Your only option (when you need one) is to suspend the interrupts either through the processor (which stops all of them except those of the non-maskable variety), or to suspend one or more through the Priority Interrupt Controller Unit.

The real effect on the tasking model comes from the fact that some interrupts may convey information that should cause an eventual task switch. On systems that are primarily time sliced, the generic clock interrupt may be the basis of most task switches. After a period of time has elapsed, you switch tasks. In a cooperative environment, it may possibly be some form of intelligent message from an ISR that causes a task switch (if an ISR causes a task switch at all).

When you get to Chapter 4, you'll see how messaging ties into and even helps to define your tasking model.

4

Interprocess Communications

This chapter introduces you to some of your options for Interprocess Communications (IPC). As I mentioned in earlier chapters, you should have a good idea what your operating system will be used for before you determine such important things as its tasking model and what forms of interprocess communications it will use. As you will see in this chapter, the IPC mechanisms play a very important part in the overall makeup of the system. You will also notice that even though I covered task scheduling in the previous chapter, I can't get away from it here because it's tied in very closely with interprocess communications.

Messaging and Tasks

As described earlier, a task is a single thread or series of instructions that can be independently scheduled for execution. How the operating system decides when a task is ready to execute, where it waits, or is suspended when it can't run, and also how it is scheduled will all depend on the messaging facilities provided by the system.

Synchronization

Synchronization of the entire system is what IPC all boils down to. On a single CPU, only one task can execute at a time. Whether a task is ready to run will depend on information it receives from other tasks (or the system) and also on how many other tasks are ready to run (multiple tasks competing for CPU time).

Looking at this synchronization from the application programmer's point of view, he or she doesn't see most of the action occurring in the system, nor should he or she have to. When a programmer makes a simple system call for a resource or I/O, the task may be suspended until that resource is available or until another task is done with it. Some resources may be available to more than one task at a time, while others are only available to a single task at any one point in time.

An example of a resource that may only be available for one task at time is a system memory-allocation routine. Of course, everyone can allocate memory in an operating system, but more than likely it will be managed in such a way that it can be handed out to only one task or program at a time. In other words, portions of the code will not be *reentrant*. I discuss reentrancy in a later section of this chapter.

In Chapter 3, "The Tasking Model," I mentioned that a task switch can occur at just about any time. This means that unless interrupts are suspended while one task is in the middle of a memory-allocation routine, this task could be suspended and another task could be started. If the memory-allocation routine was in the middle of a critical piece of code (such as updating a linked list or table) and the second task called this same routine, it could cause some serious problems. This is one place where the IPC mechanism comes into play and is very important. You can't suspend interrupts for extended periods of time. I discuss this also in detail later in this chapter.

Semaphores

One common IPC mechanism is called a *semaphore.* As the name implies, semaphores are used for signaling and exchanging information. Semaphores are code routines and associated data structures that can be accessed by more than one task.

In operating systems that use semaphores, there are public calls that allocate semaphores, perform operations on the semaphore structures, and finally destroy (deallocate) the semaphore and its associated structures when they are no longer required.

A semaphore is usually assigned a system-wide unique identifier that tasks can use to select the semaphore when they want to perform an operation on it. However, some systems allow private semaphores that only related tasks can access. This identifier is generally associated with the ID of the task that created it, and more than likely it will be used to control access to one or more functions that the owner task provides. This allows you to lock out tasks when necessary to prevent simultaneous execution on non-reentrant code. The semaphores play traffic cop. They are used also to signal the beginning or ending of an event.

Typical semaphore system calls are `semget()`, which allocates a semaphore; `semctl()`, which controls certain semaphore operations; and `semop()`, which performs operations on a semaphore. The arguments (or parameters) to these calls usually include the semaphore ID (or an indication of where to return the ID if it's being allocated), what operation you want to perform, and a set of flags that indicate specific desires or conditions that your task wants to adhere to, such as one that says "Don't suspend me if I can't get by you" or "I'll wait till I can get it."

In most cases, the application programmer won't even know that a semaphore operation is being performed. He or she will make a system call for some resource, and the library code or code contained in the system will perform the semaphore operation that ensures synchronized access to its code (if that's its purpose) or that ensures some other required condition on the system is met before the calling task is scheduled to execute.

When a semaphore operation is performed and the calling task is suspended, it is waiting at that semaphore for a condition to change before proceeding (being restarted to execute). As you can see, in order to work properly, the semaphore mechanism would have to be coupled with all of the other kernel code, such as those procedures that create new tasks, destroy tasks, or schedule tasks for execution.

I opted not to use semaphores in my operating system. I studied how they were used in UNIX systems and in OS/2, and it was my opinion that they added an unnecessary layer of complexity. Furthermore, all of the functions that they perform are available with simpler messaging facilities.

Another reason I avoided semaphores is that they are usually associated specifically with tasks (processes). In my system, tasks are not wholly independent. They rely on a certain data contained in structures that are assigned to a program or job that will consist of one or more tasks.

If you intend to implement semaphores, you should study how they are implemented in the more popular systems that use them (such as UNIX, OS/2, and so on). One thing to note is that portability may be an issue for you. Semaphore functions on some systems are actually implemented as an installable device instead of as an integrated part of the kernel code. In my humble opinion, I think they are best implemented as tightly integrated pieces of the kernel code. This is much more efficient.

Pipes

Another popular IPC mechanism is called a *pipe*. Pipes are pseudo input/output devices that can be used to send or receive data between tasks. They are normally stream-oriented devices, and the system calls will look very much like file-system operations.

Pipes can be implemented as public devices, which means they can be given names so that unrelated tasks can access them. Quite often they are not public and are used to exchange data between tasks that are part of one program.

Common system calls for pipe operations would include `CreatePipe()`, `OpenPipe()`, `ClosePipe()`, `ReadPipe()`, and `WritePipe()`. The parameters to these calls would be very similar to equivalent file operations. In fact, some systems allow standard I/O to be redirected through pipes. OS/2 is one such system.

If you intend to implement pipes as part of your IPC repertoire, you should study how they are used in other systems.

I initially considered public named pipes for MMURTL to allow data transfer between system services and clients, but I used a second form of messaging instead. If you're not worried about size or complexity in your system, public pipes are very handy. They also don't have to be so tightly coupled with the kernel code.

Messages

Messaging methods can be implemented in several different ways (although not in as many different ways as semaphores), as follows:

Send and Wait

The most rudimentary form of messaging on any system is a pair of system calls generally known as `send()` and `wait()`. You will see these commands, or similar ones, on most real-time operating systems. It is even possible to implement messaging routines using semaphores and shared memory. This would provide the needed synchronization for task management on the system. But once again, you must keep the code-maintenance aspects in mind for whatever implementation you choose.

The best method to implement messaging (and I don't think I'll get an argument here) is to make it an integral part of the kernel code in your system. Indeed, the entire kernel should be built around whatever messaging scheme you decide on.

With the `send()` and `wait()` types of messaging, the only other resource required is somewhere to send a message and a place to wait for one (usually the same place).

The best way I can illustrate messaging is to provide you the details of my own messaging implementation. This will afford an overview of messaging for the novice as well as the details for the more-experienced systems programmer.

In the system I've included for you (MMURTL), a program is made up of one or more tasks. I refer to a program as a *job*. The initial task in a program may create additional tasks as needed. These new tasks inherit certain attributes from the initial task. I am describing this because you may decide to implement tasks as wholly independent entities.

In a message-based system such as MMURTL, tasks exchange information and synchronize their execution by sending messages and waiting for them. On many message-based systems for smaller embedded applications, only the basic form of messaging is implemented. I have added a second, more-powerful form of messaging, which adds two more kernel message functions: `request()` and `respond()`. This now gives MMURTL two forms of messaging, one which is a "request" for services, which should receive a response, and a non-specific "message" that doesn't expect a response. The request/respond concept is what I consider the key to the client-server architecture that was one of my original goals. Other operating systems use this type of messaging scheme also.

Sending and receiving messages in any message-based system is not unlike messaging of any kind (even phone messages). You can send one-way messages to a person, such as, "Tell Mr. Zork to forget it. His offer is the pits." This is an example of a one-way message, for which you expect no response. One key element in a message system is *where* you leave your messages. In Bob Zork's case, he will get the message from his secretary. In an operating system's case, it is generally an allocated resource that can be called a mailbox, a bin, or just about any other name you like. It is a place where you send messages or wait for them to exchange information. In MMURTL's case, I called it an *exchange*. In order to send or receive a message, you must have an exchange. In your system, you will provide a function to allocate a mailbox, an exchange point, or whatever you call it. MMURTL provides a call to allocate exchanges for jobs. It is called `AllocExch()` and is defined as follows in C:

```
unsigned long AllocExch(long *pdExchRet);
```

The parameter `pdExchRet` points to a dword (32-bit unsigned variable) where the exchange number is returned. The function return value is the kernel error if there is one (such as when there are no more exchanges). It returns 0 if all goes well.

You also need a way to send the message. Operating-system calls to send a message come in several varieties. The most common is the `send()` function I described earlier. I implemented

this as `SendMsg()`. It does just what it says—it sends a message. You tell it what exchange, give it the message, and away it goes. The task that you want to receive the message may be "waiting" at the exchange by calling the `wait()` function I described earlier. In MMURTL I call this `WaitMsg()`. If there is no task waiting at that exchange, the message will wait there until a task waits at the exchange or checks the exchange with `CheckMsg()`. This is an added messaging function to allow you to check an exchange for a message without actually waiting. The C definitions for these calls and their parameters as I have implemented them are as follows:

```
unsigned long SendMsg(long dExch,
                      long dMsgPart1,
                      long dMsgPart2);
```

The `dExch` is the destination exchange.

The `dMsgPart1` and `dMsgPart2` are the two dwords in the message. Note that my messages are two 32-bit values—not too large to move around, but large enough for a couple of pointers.

```
unsigned long WaitMsg(long dExch,
                      char *pMsgRet);
unsigned long CheckMsg(long dExch,
                       char *pMsgRet);
```

The `dExch` is the exchange where you will wait to check for a message.

The `pMsgRet` points to an eight-byte (two dwords) structure where the message will be placed.

Did you notice (from the preceding definitions) that not only messages wait at exchanges, but tasks can wait there, too? This is an extremely important concept. Consider the example of the telephone again. The task is the human, the answering machine is the exchange. You can leave a message on the machine (at the exchange) if no one (no task) is waiting there. If a human is there waiting (a task is at the exchange, waiting), the message is received right away.

Now consider this: In a single processor system that is executing a multitasking operating system, only one task is actually executing instructions. All of the other tasks are waiting somewhere.

There are only two places where a task can wait in MMURTL: at an exchange or on the Ready queue. The Ready queue is the line-up of tasks that are in a ready-to-run state but are not running because there's a higher-priority task currently executing.

Here's one more quick topic to round off what I've already covered. Tasks are started with the kernel primitives `SpawnTask` or `NewTask`. You point to a piece of code, provide some other pieces of information, and *voila*, a task is born. Yes, it's a little more complicated than that, but you have enough information to understand the basic theory.

Now I have defined some very important terms and functions—not in detail yet, but enough that I can talk about them. `SendMsg()`, `CheckMsg()`, `WaitMsg()`, `SpawnTask()`, and `NewTask()` are five very important kernel primitives. `AllocExch()` is an important auxiliary function. The only

reason this is discussed with the rest of the kernel primitives is because of its importance. I don't consider it part of the kernel, because it has no effect on the tasking model (CPU time allocation). You also know now about exchanges and the Ready queue.

I apologize that none of the items I've introduced here have names that are confusing or based on buzzwords. I'll try to liven it up later on.

You now have enough information under your belt to provide a scenario of message passing that will help enlighten you to what actually occurs in a message-based system. Table 4.1 shows actions and reactions between a program and the operating system.

You start with the execution of a single task. What it's doing isn't important. Imagine that as you move down the table in the example, time is passing. In this example, whenever you call a kernel primitive, you enter The Kernel Zone. (Just kidding. It's not called The Kernel Zone—just the "kernel.")

Table 4.1.Task and kernel interaction.

Task Action	*Kernel Action*
Task1 is running.	
Task1 allocates Exch1.	
Task1 calls `SpawnTask` (to start Task2).	
	Kernel checks priority of new task. Task2 is higher.
	Kernel places Task1 on the Ready queue.
	Kernel makes Task2 run.
Task2 is running.	
Task2 allocates Exch2.	
Task2 sends a message to Exch1.	
	Kernel checks for a task waiting at Exch1. None are waiting.
	Kernel attaches message to Exch1. Kernel evaluates Ready queue to see what runs next. It's still Task2.
Task2 is still running.	
Task2 calls `WaitMsg` at Exch2.	
	Kernel checks for a message waiting at Exch2. None found. Kernel places Task2 on Exch2.

continues

Table 4.1. continued

Task Action	*Kernel Action*
	Kernel evaluates Ready queue.
	Task1 is ready to run.
	Kernel makes Task1 run.
Task1 is running.	
Task1 sends a message to Exch2.	
	Kernel checks for task at Exch2 and finds Task2 there. It gives Task2 the message.
	Kernel places Task1 on Ready queue.
	Kernel makes Task2 run. (It's a higher priority.)
Task2 is running.	

From the simple example in Table 4.1, you can see that the kernel has its job cut out for it. You can also see that it is the messaging and the priority of the tasks that determine the sharing of CPU time (as was described in previous chapters). From the example you can also see that when you send a message, the kernel attaches the message to an exchange. If there is a task at the exchange, the message is associated with that task and it is placed on the Ready queue. The Ready queue is immediately reevaluated, and if the task that just received the message is a higher priority than the task that sent it, the receiving task is made to run.

What would happen if both Task1 and Task2 waited at their exchanges and no one sent a message? You guessed it—the processor suddenly finds itself with nothing to do. Actually it really does absolutely nothing. In some systems, dummy tasks are created with extremely low priorities, so there is always a task to run. This can simplify the scheduler. This low-priority task can even be made to do some chores, such as garbage collection or a system-integrity check. I didn't do this with MMURTL. I actually halt the processor with interrupts enabled. If everyone is waiting at an exchange, they must be waiting for something. More than likely it is something from the outside world, such as a keystroke. Each time an interrupt occurs, the processor is activated and it checks the Ready queue. If the interrupt that caused it to wake up sent a message, there should be a task sitting on the Ready queue. If there is one, it will be made to run. This may not work on all types of processors. You will have to dig into the technical aspects of the processor you are using to ensure that a scheme like this will work.

Request()

I have added two more messaging types in my system. If a very small embedded kernel is your goal, you will more than likely not use these types in your system. They are dedicated types.

`Request()` and `Respond()` form the basis for a client-server system that provides for identification of the destination exchange and also allows you to send and receive much more than the eight-byte message used with the `SendMsg()` primitive.

The `Request()` and `Respond()` messaging primitives are designed so you can install a program called a *system service,* which provides shared processing for all applications on the system. The processing is carried out by the service, and a response (with the results and possibly data) is returned to the requester.

Message-based services are used to provide shared processing functions that are not time critical (where a few-hundred-microsecond delay would not make a difference), and also need to be shared with multiple applications. They are ideal for things such as file systems, keyboard input, printing services, queued file management, e-mail services, BBS systems, fax services, the list could go on and on. The MMURTL FAT-file system and keyboard are both handled with system services. In fact, you could implement named pipes (discussed earlier in this chapter) as a system service.

With MMURTL, each service that is installed is given a name. The name must be unique on the machine. When the service is first installed, it registers its name with the OS Name Registry and tells the OS what exchange it will be serving. This way, the user of the system service doesn't need to know the exchange number, only the service name. The exchange number may be different every time it's installed, but the name will always be the same. A fully loaded system may have 5, 10, or even 15 system services. Each service can provide up to 65,533 different functions as identified by the service code. The functions and what they do are defined by the service. The OS knows nothing about the service codes (except for one discussed later). It doesn't need to, because the `Request()` interface is identical for all services.

The interface to the `Request()` primitive is procedural, but it has quite a few more parameters than `SendMsg()`. Look at it prototyped in C:

```
unsigned long Request(char *pSvcName,
                      unsigned int wSvcCode,
                      long dRespExch,
                      long *pdRqHndlRet,
                      long dnpSend
                      char *pData1SR,
                      long dcbData1SR,
                      char *pData2SR,
                      long dcbData2SR,
                      long dData0,
                      long dData1,
                      long dData2 );
```

At first glance, you'll see that the call for making a request is a little bit more complicated than `SendMsg()`. But, as the plot unfolds you will see just how powerful this messaging mechanism can be (and how simple it really is). Consider each of the parameters:

`pSvcName` Pointer to the service name you are requesting. The service name is eight characters, all capitals, and space-padded on the right.

`wSvcCode` Number of the specific function you are requesting. These are documented independently by each service.

`dRespExch` Exchange that the service will respond to with the results (an exchange you have allocated).

`pdRqHndlRet` Pointer where the OS will return a handle to you that is used to identify this request when you receive the response at your exchange. This is needed because you can make multiple requests and direct all the responses to the same exchange. If you made more than one request, you'll need to know which one is responding.

`dnpSend` The number (0, 1, or 2) of the two data pointers that are moving data from you to the service. The service already knows this, but network transport mechanisms do not. If `pSend1` was a pointer to some data the service was getting from your memory area and `pSend2` was not used or was pointing to your memory area for the service to fill, `dnpSend` would be 1. If both data items were being sent to you from the service, this would be 0.

`pData1` Pointer to memory in your address space that the service must access (either to read or write as defined by the service code). For instance, in the file-system `OpenFile()` function, `pData1` points to the file name (data being sent to the service). This may even point to a structure or an array, depending on how much information is needed for this particular function (service code).

`dcbData1` How many bytes the `pDataSend` points to. Using the `OpenFile()` example, this would be the size (length) of the filename.

`pData2` This is a second pointer, exactly like `pData1` described previously.

`dcbData2` This is the same as `dcbData1`.

`dData0`, `dData1`, `dData2` These are three dwords that provide additional information for the service. In many functions you will not even use `pData1` or `pData2` to send data to the service, but will simply fill in a value in one or more of these three dwords. These can never be pointers to data. This will be explained later, in the discussion of memory management.

Respond()

The `Respond()` primitive is much less complicated than `Request()`. This doesn't mean the system service has it easy. There's still a little work to do. Here is the C prototype:

```
Respond(long dRqHndl, long dStatRet);
```

The parameters are also a little less intimidating than those for `Request()`. They are as follows:

`dRqHndl` The handle to the request block that the service is responding to.

`dStatRet` The status or error code returned to the requester.

A job (your program) calls `Request()` and asks a service to do something, then it calls `WaitMsg()` and sits at an exchange waiting for a reply. If you remember, the message that comes in to an exchange is an eight-byte (two dwords) message. Two questions arise:

1. How do you know if this is a response or just a simple message sent here by another task?
2. Where is the response data and status code?

First, the content of the eight-byte message tells you if it is a message or a response. The convention is the value of the first dword in the message. If it is 80000000h or above, it is *not* a response. Second, this dword should match the request handle you were provided when you made the `Request()` call. (Remember `pRqHndlRet`?) If this is the response, the second dword is the status code or error from the service. Zero (0) usually indicates no error, although its exact meaning is left up to the service.

Second, if the request was serviced (with no errors), the data has already been placed into your memory space where `pData1` or `pData2` was pointing. This is possible because the kernel provides alias pointers to the service into your data area to read or write data. Also, if you were sending data into the service via `pData1` or `pData2`, the kernel has aliased this pointer for the service as well, and the service has already read your data and used it as needed.

Not as difficult as you expected, right? But let me guess—this aliasing thing with memory addresses is still a bit muddy to you. A little further into this chapter I'll cover memory management as it applies to messaging, which should clear things up considerably.

Link Blocks

I keep referring to how a message or a task just "sits" or "waits" at an exchange. An exchange is a small structure in memory. You will need some way (a method) to attach messages to your mailbox. It will no doubt be a linked list of sorts.

I opted to use something called a *link block* (LB). A link block is a little structure (smaller than an exchange) that becomes a link in a linked list of items that are connected to an exchange. Link blocks are not very big, but still very important. (You will find out how important they are if you run out of them!) There is one link block in use for every outstanding (unanswered) request and one for every message waiting at an exchange. This can add up to hundreds in a real hurry.

Reentrancy Issues

A multitasking system that lets you simply point to a set of instructions and execute them means that you can actually be executing the same code at almost the same time. I say "almost" because really only one task is running at a time.

This can lead to serious problems if the code being executed is not designed to be *reentrant.* Reentrant means that if a task is executing a series of instructions and gets preempted, or if it pauses in the middle of them, it's OK for another task to enter and execute the same code before the first task is done with it.

To really understand this issue, let's look at an example. I'll use a pseudo-text algorithm to define a series of instructions that are *non*-reentrant and show you what can happen. The example is a function that allocates buffers and returns a pointer to it. It's called `GetMeSomeMem()`. In this example, the code accesses two variables to carry out its mission.

```
Variable nBuffersLeft
Variable pNextBuffer

GetMeSomeMem()
If (nBuffersLeft > 0)

    (A place between instructions called PointX)

    {   (Allocate a Buffer)
        Decrement nBuffersLeft
        Increment pNextBuffer
        Return pNextBuffer
    }
else
    Return Error
```

Let's suppose two tasks are designed to call this function. As you know, only one task can really be running at a time, even in a multitasking system. So, Task1 calls `GetMeSomeMem()`. But when it reaches `PointX`, it is preempted. For the sake of understanding this, there was only one buffer left. Now, Task2 is running and calls `GetMeSomeMem()`. It successfully makes it all the way through and gets to that last buffer. Now, Task1 gets to run again. He is restarted by the scheduler and starts up at `PointX`. The code has already checked and there was one buffer left. So the code goes in and whammo!—it's crash time. The buffer is gone; pointers are off the ends of arrays; it's a mess.

As you can see, some method is required to prevent two tasks from being inside this code at the same time. Semaphores are used on many systems. Each task that enters the code checks the semaphore and will be suspended until the last task is done with it. The call to manage and check the semaphore would actually be part of the allocation code itself. Messages can also provide exactly the same functionality. Allocate a mailbox or exchange and send it one message. Each task will `wait()` at the exchange until it has a message. When each task is done with the allocation routine, it sends back the message so the next task can get in. It's really quite simple.

What's usually not so simple for the operating-system writer is ensuring that you identify and protect all of the non-reentrant code.

If the instruction sequence is short enough, you can simply disable interrupts for the critical period of time in the code. You should always know how long this will be, and it shouldn't be

too long. If you disable interrupts for too long, you will suffer interrupt latency problems. You will lose data, as discussed forthwith.

Interrupt Latency

Interrupt latency is when interrupts don't get serviced fast enough, and maybe even not often enough. I mentioned in the preceding section that you shouldn't disable interrupts for too long a period of time. Just to give you an idea of what periods of time I'm talking about, look at a real-world example. A nonbuffered communication UART (Universal Asynchronous Receiver Transmitter) operating at 38,400 bits per second will interrupt every 208 microseconds. This is $1/38{,}400 \times 8$ because they will interrupt for every byte (8 bits). The typical processor running at 25MHz executes most of its instructions in 2 or 3 system-clock periods. That would be an average of 120 nanoseconds ($1/25{,}000{,}000 \times 3$). In theory, this means you could execute as many as 1,730 instructions in the interrupt interval.

Whoa there—that was only in theory! Now you have to do a reality check. You must take into consideration that there are more interrupts than just that communications channel. The timer interrupt will be firing off every so often. The communications interrupt itself will have interrupts disabled for a good period of time, and there is also "the biggie": task switches.

Why are task switches considered "the biggie"? A task switch can consume as many as 30 or 40 microseconds. The actual amount of time will depend on whether you use hardware task switching or do it yourself, but you will still have interrupts disabled for a rather scary length of time. Also, if you remember the `GetMeSomeMem()` example, you know you locked out multiple users with the kernel messaging (or semaphores). You didn't disable interrupts. In the kernel code, you can't use the kernel code to lock things out of the kernel. You *must* disable interrupts in many sections. It will be the only way.

Good, tight code in the kernel is a must. I could probably tighten mine a whole lot, but my testing shows I did OK. You will have to do some serious testing on your system to ensure interrupt latency is not going to be a problem. Have you ever run a communications program for an extended period of time at 19,200 baud in Windows 3.x? You will lose data. Don't blame Windows, though. It's doing some things I personally consider impossible.

Memory Management Issues

The last hurdle in interprocess communications is memory. It doesn't sound like it would be a problem, but it can be, depending on how you implement your memory model.

If you use a completely flat model for the entire system, it will simplify IPC greatly—but it will increase the headaches elsewhere. A huge linear array of memory in your system that is shared by all the programs and tasks means they all understand a pointer to any memory address. This

in turn means you can freely pass pointers in messages back and forth (even between different programs) and no one gets confused.

If you implement independent linear-memory arrays through paging hardware (as I did with MMURTL), then you have to find a way to translate (alias) addresses that are contained in messages that will be passed between different programs.

Segmented memory models will suffer from this aliasing requirement to an even greater extent. I recommend you avoid a fully segmented model. I started with one but abandoned it early on. Fully segmented models allow any memory allocation to be zero-based. This means that if paging is used along with the segmentation, it will be even a greater chore to translate addresses that are passed around on the system.

You will learn about memory-management options in Chapter 5, "Memory Management."

5

Memory Management

This chapter provides you with some fairly detailed ideas for memory management from an operating system's "perspective." An operating system's view of memory management is quite different than that of a single application.

If you've ever written memory-allocation routines for compiler libraries, or other complicated applications, you know you are basically asking the operating system for chunks of memory that you will break down even further as required for your application.

Just as your memory management routines keep track of what they hand out, so does an operating system. There are some major differences though. These differences vary considerably based on how you choose to handle the memory on the system.

Memory-management code in an operating system is affected by the following things (this list is not all inclusive, but touches on the important things):

1. How the processor handles addressing. This greatly affects the next three items.
2. The memory model you choose.
3. Whether or not you use paging hardware, if available.
4. Whether you allow variable-sized chunks of memory to be allocated.

I do make some assumptions about your processor. I can do this because most of the processors afford roughly the same memory-management schemes. Some do it with internal paging hardware; some do it with an external *page memory management unit* (PMMU). Either way, it's all basically the same.

Basic Terms

Once again, I need to define some terms to ensure we are talking about the same thing.

Physical memory is the memory chips and their addresses as accessed by the hardware. If I put address 00001 on the address bus of the processor, I am addressing the second byte of physical memory. Address 0 is the first.

Linear memory is the address space a program or the operating system sees and works with. These addresses may or may not be the same as the physical addresses. This will depend on whether or not some form of hardware translation takes place. For instance, Intel uses this term to describe memory addresses that have been translated by paging hardware.

Logical memory is more an Intel term than a generic term that can be applied across different manufacturers' processors. This is the memory that programs deal with and is based around a *selector* (a 16-bit number that serves as an index into a table). With the Intel processors, a protected-mode program's memory is always referenced to a selector that is mapped in a table to linear memory by the operating system and subsequently translated by the processor. I will discuss segmentation in greater detail in this chapter for those of you who may want to use it.

Memory Model

Do not confuse my term "memory model" for the MS-DOS/Intel term. It does not refer specifically to size or to any form of segmentation. Instead, it refers to how and where programs are loaded, and how the operating system allocates and manages processor memory space.

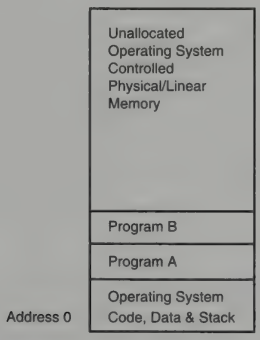
There are several basic memory models to choose from and countless variations. You may be restricted from using some of these options, depending on what your processor or external memory management hardware allows. I'll start with the easiest to understand and work from there. Keep in mind that the easiest to understand may not necessarily be the easiest to implement.

Simple Flat Memory

Simple flat memory means that the operating system and all programs share one single range of linear memory space, and that physical addresses and linear addresses are the same. No address translations take place in hardware.

In this type of system, the operating system is usually loaded at the very bottom or very top of memory. As programs are loaded, they are given a section of memory (a range of address) to use as their own. Figure 5.1 shows this very simple shared-memory arrangement.

FIGURE 5.1.
Simple flat memory.



Unless some form of protection is used, any program can reach any other program's memory space (intentionally or unintentionally). This can be good and bad. The good part is that no address translations are required for any kind of interprocess communications. The bad part is that an invalid pointer will more than likely cause some serious problems.

Hardware memory paging would possibly provide a solution to the bad pointer problem. Paging hardware usually affords some form of protection, at least for the operating system. If the

paging hardware provides only two levels of protection, this still means that program A can destroy program B as long as they both share the same linear space and are at the same protection level.

Considerations and variations of this simple model include paging, different memory allocation schemes (top down, bottom up, and so on), and possible partitioning for each program. Partitioning means you would allocate additional memory for a program it would use as its unallocated heap. This would keep all of the program's memory in one range of linear addresses. As programs are loaded, they are partitioned off into their own space with a lower and upper bound.

Paged Flat Memory

Paged flat memory is a variation on simple flat memory. With this scheme, chunks of physical memory, called *pages*, are allocated and assigned to be used as a range of linear memory. This is still a flat scheme because all the programs will use the same linear addresses. They can still share and get to each other's address space if no protection is applied. This scheme requires the use of paging hardware.

Figure 5.2 shows how the blocks of physical memory are turned into addressable chunks for the system. With paged memory, the physical memory can come from anywhere in the physical range to fit into a section of linear address space. With this scheme, the linear address range can also be much larger than the amount of physical memory, but you can't actively allocate more linear address space than you have physical memory. For example, your linear address space may run from 0 to 1 gigabytes, but only 16 megabytes of this linear address range may be active if that's all the physical memory your system has. This gives the operating system some options for managing memory allocation and cleanup. Figure 5.2 also implies that the physical pages for the program were allocated from the "top down" to become its addressable area from the "bottom up" which is perfectly legitimate.

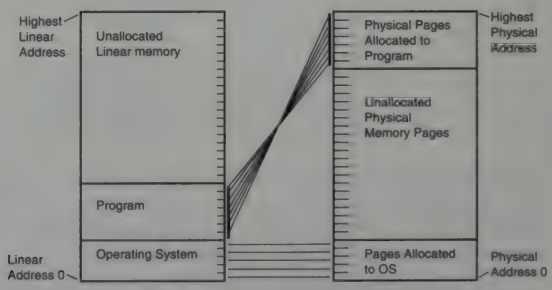


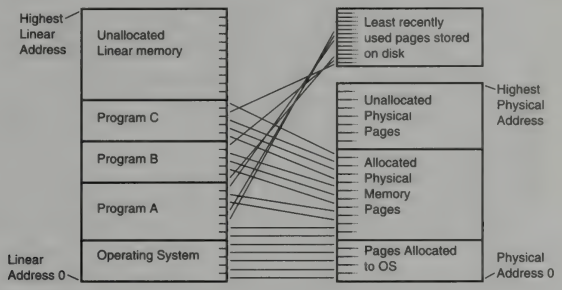
FIGURE 5.2.
Paged flat memory.

Demand-Paged Flat Memory

Demand-paged flat memory is a major variation on paged flat memory. With demand-paged memory, you extend the active size of your linear memory beyond your physical memory's limits. For instance, if you have 16MB of physical memory, your system's actively-allocated linear address range may actually be extended to a much greater size (say 32, or even 64MB). The additional pages of physical memory will actually be contained on an external, direct-access storage device, more than likely a hard disk drive. The word *demand* means that they are sent out to the disk when not needed, and brought back into real physical address space on demand or when required.

Figure 5.3 shows an example of the page translations that can occur.

FIGURE 5.3.
Demand-paged flat memory.



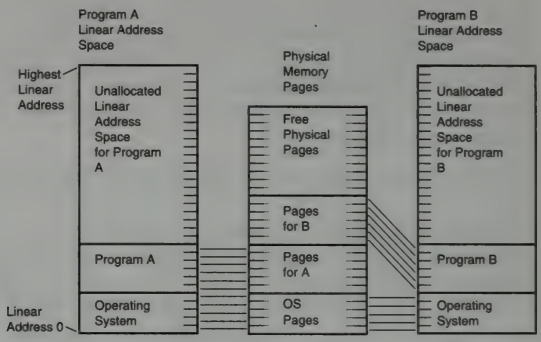
In this scheme, all the programs still use the same linear space. Address 30000h is the same to all programs and the operating system. Something else to note is that the actual values of these linear addresses may extend to the highest values allowed by the paging hardware. You can start your application addressing at the 3GB mark if you desire, and the hardware allows it.

Virtual Paged Memory

The term *virtual* usually means it's not real, but it sure looks like it is, as in virtual reality. The minute you apply this term to memory management, it means that you're lying to your programs, and indeed you are. With paging hardware, you're already lying to them when you map in a physical page and it's translated to a linear page address they can use. So you can really call any form of memory management where address translations take place "virtual" memory. Some of the older systems referred to any form of demand paging as virtual memory. My definition of virtual is telling a *big* lie (not a little one).

This means you can tell two programs they are both using linear address ranges that are the same! In other words, they are operating in "parallel" linear spaces. Look at Figure 5.4 for an example that will help you understand this technique.

FIGURE 5.4.
Virtual paged memory.



Like the paged flat memory, physically-allocated pages can be placed anywhere in a program's linear address space. Unlike paged flat memory, virtual paged memory is *not* flat. To a single program with its own address space, it may appear flat, which is the idea, but linear address 30000h to one program may not be the same physical memory addressed as linear address 30000h to another program (same address, different data or code). Notice I said *may not*, and not *will not*. This is because you can make the physical addresses match the linear addresses for two programs if you desire. The options here are almost mind boggling. You can have all programs share a section of the same physical memory in a portion of their linear space, but make all the rest of their linear space completely independent and inaccessible by any other program. In Figure 5.4 you can see that both programs see the operating system pages at the same addresses in their linear spaces, but the same linear range of addresses where program code and data are stored are completely independent.

Note the advantages here. This means you can share operating system memory between all your programs, yet any memory they allocate can be theirs alone.

Note also the disadvantages. The largest will be when you implement interprocess communications. You will need some form of translation between addresses that are passed from one linear space to another. A way around this to use a shared memory area. Once again, there are many possible variations. No doubt you will think of, and implement, some of your own.

In order to allow multiple, parallel linear spaces, the memory management hardware must support it. You will have to do some homework for your processor of choice.

Demand-Paged Virtual Memory

Just like demand-paged flat memory, demand-paged *virtual* memory is when you can allocate more pages than you physically have on your system. The same type of on-demand external storage is used as with demand-paged flat memory. The major difference is that these pages can be for any one of several independent linear address spaces.

Demand-Paged Memory Management

Management of demand-paged memory is no trivial task. The most common method is with a least recently used (LRU) algorithm. It's just what it sounds like. If you need to move someone's physical pages back into their linear space, the pages that are moved out to make room are the ones that haven't been used recently.

The LRU schemes have many variations. The most common, and generally the most efficient, is to use a separate thread or task that keeps track of how often pages are used. The paged memory management hardware will tell you which ones have been used and which ones have been sitting idle since they were paged in or created. In fact, paging hardware will tell you whether a page has been accessed only for reading or has actually been modified. When a page has been written to, it is usually termed as *dirty*. The idea of a page being dirty has some significance. If you had read in a page and it was accessed for reading, and now you want to move it out to disk because it hasn't been used lately, you may already have a *clean* copy on the disk. This means you can save the time rewriting it. This can be a big time-saver when you're talking about thousands of pages being swapped in and out.

This additional task will try to keep a certain percentage of physical space cleared for near term page-in or page creation (memory allocation) requirements.

Segmented Memory

I will discuss segmented memory because the Intel processors allow this type of memory, and you need to understand the implications. You may even want to use it to make your system compatible with some of the most popular operating systems in use (OS/2 or Windows for example). This compatibility may be in the form of the ability to use code or object modules from other systems that use segmentation, and maybe to use the Intel Object Module format.

Segmentation is really a holdover from the past. When this was a 16-bit world, or even an 8-bit one, we had some fairly limited address spaces. A multitude of software was written for 16-bit addressing schemes.

When Intel built the first of its 32-bit processors (80386), Intel made sure that the 32-bit processor was downward-compatible with code written for the 16-bit processors. Likewise, when Intel built the 16-bit processors (8086, 8088, 80186, and 80286), the 16-bit processors were also compatible with object code for the 8-bit processors (8080 and 8085, and the popular Zilog Z80).

The 8-bit processors had 16 address lines. This meant you could have a whopping 64K of RAM. It also meant that when they built processors with more address lines, 20 to be exact, they needed a way to use all of the 8-bit processor code that used 16-bit addressing.

What Intel came up with is called *segmentation.* They allowed multiple sequential, or overlapping 64-K sections of memory called segments. To make this work, they needed to identify and manage those segments. They used something called a *segment register.* This was actually several registers, one for code and several for data access, that would identify which particular 64K-addressable area you were working with. What they needed was a transparent way to use the extra four bits to give you access to a full 1MB of RAM. The segment registers gave you an addressable granularity of 16 bytes.

All of the Intel products are very well documented, so I won't go into further detail. However, this introduction leads me to another very important point about segmentation that you need to understand. The segment registers in the 16-bit processors were a simple extension to the address lines. When the 80286 came on the scene, they added a very important concept called Protected mode. This made it possible to manage these segments with a *selector.* A selector is a number that is an index into a table that selects which segment you are working with. It is really a form of virtual memory itself. Protected mode operation with selectors carried over into the 32-bit processors.

Another major change in the 32-bit processors that deal with segments is the fact that they can now be as large as all available memory. They are not limited to 64K.

The selectors are managed in tables called the *Global Descriptor Table* (GDT) or *Local Descriptor Tables* (LDT). When you use a selector, the processor uses the table entry indicated by the selector and performs an address translation. The descriptor tables allow you to set up a zero-based address space that really isn't at linear address zero. This means you can locate code or data anywhere you want in the 4GB of linear memory, and still reference it as if it were at offset zero. This is a very handy feature, but it can become cumbersome to manage because it can force you to use something called *far* pointers to access almost everything. The far pointer is a pointer that includes the selector as part of the address. With a 32-bit system, this makes a far pointer 48 bits. This can be a real hassle. It also slows things down because the processor must do some housekeeping (loading shadow registers) each time the selector is changed.

Memory Management

After you've chosen a memory model, you'll face many decisions on how to manage it. Management of memory involves the following things:

- Initialization of memory management
- Allocating memory to load applications and services
- Allocating memory for applications

- Tracking client linear memory allocations
- Tracking physical memory, if paging is used
- Handling protection violations, if your hardware supports protection

I'll discuss each of these things, but not in the order I listed them. This is because you have to figure out how you'll manage memory before you know how to initialize or track it.

Tracking Linear Memory Allocation

Two basic methods exist to track allocation of linear or physical memory. They are *linked lists* and *tables*. Before you decide which method to use, you should decide on your basic allocation unit.

Basic Memory Allocation Unit

The basic allocation unit defines the smallest amount of memory you will allow a client to allocate. Remember, *you* are the operating system. You make the rules. You can let applications allocate as little as one byte, or you can force them to take 16 kilobytes at a time. You'll have to make a decision, and it will be based on many factors. One of the important ones will be whether or not you use hardware paging. If you do, you may want to make the allocation unit the same size as a page, which will be defined by the paging hardware. This will simplify many things that your memory management routines must track. Other factors include how much memory you have, the type of applications you cater to, and how much time and effort you can put into to the algorithms.

I don't know of any theoretical books that take into account how much time it takes to actually implement pieces of an operating system, but it's a very realistic part of the equation, especially if you have a very small programming team—one person, for instance. I learned this the hard way, and my advice to you is not to bite off more than you can chew.

Linked List Management

Using linked lists is probably the most common and one of the easiest ways to manage memory. This type of algorithm is most common when a large linear address space is shared by one or more applications, such as I described earlier with the simple flat memory model. With this scheme, you have one or more linked lists that define allocated and/or free memory. Hundreds of algorithms have been designed for linked list memory management. I will give you a good idea what it's like, and then it's up to you to do some serious research to find the best method for your system.

The linked lists are set up when the operating system initializes memory. How you store the links in this list can vary. The easiest and most efficient way is to store the link at the beginning of the allocated block. This way, each block holds its own link and can be located almost instantly with the memory reference. The links are small structures that contain pointers to other

links, along with the allocated size and the memory block owner. I recommend you track which client owns each link for deallocation and cleanup purposes. In operating systems that don't provide any type of protection, when one application crashes, it usually takes the whole system down. In many systems, however, when one application crashes, it is simply terminated, and the resources it used are recovered.

The word *cleanup* also implies that the memory space becomes fragmented as callers allocate and deallocate memory. This is another topic entirely, but I'll touch on it some.

MS-DOS appears to use links in the allocated space. I haven't disassembled their code; I can tell this from the pointer values I received when I allocate memory from the MS-DOS operating system heap. Ignoring the fact that MS-DOS uses segmented pointers, you can see when you do the arithmetic that there are a few conspicuous bytes missing from two consecutive memory allocations. You now have an idea where they went.

In the simple linked list scheme in Listing 5.1, you have two structure variables defined in your memory management data. These are the first links in your list. One points to the first linked block of free memory (`pFree`), and the other points to the first linked block of allocated memory (`pMemInUse`).

Listing 5.1. Linked list memory management.

```
/* Memory Management Variables */

struct MemLinkType    /* 16 bytes per link */
{
   char *pNext;
   char *pPrev;  /* a backlink so we can find previous link */
   long  size;
   long  owner;
};
struct MemLinkType pFree;
struct MemLinkType pMemInUse;
struct MemLinkType *pLink;       /* to work with links in memory */
struct MemLinkType *pNewLink;    /* "  " */
struct MemLinkType *pNextLink;   /* "  " */
struct MemLinkType *pPrevLink;   /* "  " */

long Client = 0;
```

In the process of memory management initialization, you would set up the two base link structures, `pFree` and `pMemInUse`.

For my description, let's assume you have 15MB of memory to manage beginning at the 1MB address mark (100000h), and all of it is free (unallocated). You would set up the base structures as follows:

```
void InitMemMgmt(void)
{
    pFree.pNext = 0x100000;  /* first link in 15 megabyte array */
```

```
    pFree.pPrev = 0;          /* No backlink */
    pFree.size  =  0;         /* No size for base link */
    pFree.owner =  0;         /* OS owns this */
    pMemInUse.pNext = 0;      /* No mem blocks allocated yet */
    pMemInUse.pPrev = 0;      /* No backlink */
    pMemInUse.size  = 0;      /* No size for base link */
    pMemInUse.owner = 0;      /* OS owns this */

    /* Set up first link in free memeory */

    pLink = 0x100000;              /* Set structure pointer to new link */
    pLink->pNext = 0;              /* NULL. No next link yet.  */
    pLink->pPrev = &pFree;         /* Backlink to pFree */
    pLink->size = 0xf00000 - 16;   /* 15 Megs-16 for the link */
    pLink->owner = 0;              /* OS owns it now */
}
```

When a client asks for memory, you follow the links beginning at `pFree` to find the first link large enough to satisfy the request. This is done by following `pLink.pNext` until `pLink.size` is large enough, or you reach a link where `pNext` is null, which means you can't honor the request. Of course, you must remember that `pLink.size` must be at least as large as the requested size *plus* 16 bytes for the new link you'll build there.

The following case of the very first client shows how all allocations will be. Let's assume the first request is for 1MB. `WantSize` is the size the client has requested, and `Client` is the owner of the new block. The function returns the address of the new memory, or a `NULL` pointer if you can't honor the request.

The example code in Listing 5.2 is expanded to show each operation. You could make this type of operation with pointers almost "unreadable" in the C programming language. If you use this code, I suggest that you leave it as is for maintenance purposes and take the hit in memory usage. I've gone back to some code I did five or six years ago and spent 30 minutes just trying to figure out what I did. Granted, changes in pointer usage were in an effort to save memory and make the code tight and fast, but it didn't seem like it was worth the headaches.

Listing 5.2. Linked list memory allocation routine.

```
char *AllocMem(long WantSize)
{

pLink = &pFree;          /* Start at the base link */
                         /* Keep going till we find one */
while (pLink->pNext)     /* As long as we have a valid link...*/
{
    pLink = pLink->pNext;    /* Next link please */

    if (pLink->size >= (WantSize + 16))    /* This one will do! */
    {
        /* Build a new link for the rest of the free block */
        /* then add it to the free list. This divides the free */
        /* block into two pieces (one of which we'll allocate) */
```

continues

Listing 5.2. continued

```
        /* Set up new link pointer, fix size and owner */

        pNewLink = &pLink + (WantSize + 16);
        pNewLink->size = pLink->size - WantSize - 16;
        pNewLink->owner = 0;              /* OS owns this for now */

        /* Hook it into free links after pLink */

        pNewLink->pPrev = &pLink;         /* Backlink */
        pNewLink->pNext = pLink->pNext;
        pLink->pNext = &pNewLink;

        pLink->size = WantSize;           /* Fix size of pLink */

        /* Remove pLink from the pFree list and put it in the */
        /* allocated links! */

        pPrevLink = pLink->pPrev;         /* get previous link */
        pPrevLink->pNext = &pNewLink;     /* Unhook pLink */

        pLink->pNext = pMemInUse.pNext;   /* Put pLink in USE */
        pMemInUse.pNext = &pLink;        /* Point to the new link */
        pLink->size = WantSize;           /* How much we allocated */
        pLink->owner = Client;            /* This is who owns it now */

        return(&pLink+16);  /* Return address of the NEW memory */
    }
}
return(0);  /* Sorry - no mem available */
}
```

You will need a routine to free up the memory and also to clean up the space as it gets fragmented. Believe me, it *will* get fragmented. The deallocation routine is even easier than the allocation routine. When they hand you a pointer to deallocate, you go to that address (minus 16 bytes) validate the link, change its owner, then move it to the `pFree` list. This is where the fragmentation comes it.

The cleanup of fragmented space in this type of system is not easy. Even though the links themselves are next to each other in memory, they can end up anywhere in the `pFree` list, depending on the order of deallocation.

There are several thoughts on cleanup. You can add code in the allocation and deallocation routines to make it easier, or you can do it all after the fact. I personally have not done any research on the fastest methods, but much depends on the patterns of allocation by your clients. If they allocate a series of blocks and then free them in the same order, the links may end up next to each other in the free list anyway. A simple scan of the list can be used to combine links that are next to each other into a single link. But you can't depend on this.

Another method is to keep the `pFree` list sorted as you add to it. In other words, do a simple insertion at the proper point in the list. This will add time to deallocation routines, but cleanup

will be a snap. You have the linear address, so you simply walk the list until you find the links that the newly freed memory belongs between, and you insert it there. Combining blocks can be done at the same time. This makes the most sense to me.

Here is a deallocation routine that defragments the space as programs free up memory. This starts at the first link `pFree` is pointing to, "walks" up the links, and finds its place in linear memory order. You try to combine the newly-freed link with the one before it or after it, if at all possible. This provides instant cleanup.

You should notice that the allocation algorithm ensured that there was always a link left at a higher address than the link you allocated. This will be useful in the routine in Listing 5.3, which returns memory to the free pool, because you can assume there will always will be a link at a higher address.

Listing 5.3. A routine to return memory to the free pool.

```
int FreeMem(char *MemAddress)
{

/* We will use pNewLink to point to the link to free up. */
/* It's actually a MemInUse link right now. */

pNewLink = MemAddress - 16;  /* point to the link to free up */

/* Some form of error checking should go here to ensure they */
/* are giving you a valid memory address. This would be the link */
/* validation I talked about. You can do things like checking to */
/* ensure the pointer isn't null, and that they really own the */
/* memory by checking the owner in the link block. */

pNextLink = pFree.pNext;    /* Start at pFree */

/* scan till we find out where we go. We will stop when */
/* we find the link we belong between. */

while ((pNextLink->pNext) && (&pNextLink < &pNewLink))
{
    pNextLink = pNextLink->pNext;
}
pPrevLink = pNextLink->pPrev;  /* this will be handy later */

/* If memory isn't corrupted, we should be there! */
/* We could just stick the link in order right here, */
/* but that doesn't help us cleanup if we can */

/* First let's see if we can combine this newly freed link */
/*  with the one before it. This means we would simply add our */
/* size + 16 to the previous link's size and this link would */
/* disappear. But we can't add our size to pFree. It's a dummy */

if (pNextLink->pPrev != &pFree)   /* Can't be pFree */
{
    if (&pNewLink == (&pPrevLink + (pPrevLink->size + 16)))
```

continues

Listing 5.3. continued

```
    {
        /* add our size and link size (16), then go away! */

        pPrevLink->size += pNewLink->size + 16;
        return(0);
     }
}
/* If we got here, we couldn't be combined with the previous. */
/* In this case, we must insert ourselves in the free list */

pPrevLink->pNext = pNewLink;
pNextLink->pPrev = pNewLink;
pNewLink->pNext  = pNextLink;
pNewLink->pPrev  = pPrevLink;
pNewLink->owner = 0; /* operating system owns this now!*/

/* Now we'll try to combine pNext with us! */

if ((&pNewLink + pNewLink->size + 16) == &pNextLink)
{
    /* We can combine them. pNext will go away! */
    pNewLink->size += pNextLink->size + 16;
    pNewLink->pNext = pNextLink->pNext;
    pLink = pNextLink->pNext;
    if (pLink)                  /* the next could have been the last! */
      pLink->pPrev = &pNewLink; /* fix backlink */
}

/* If we didn't combine the new link with the last one, */
/* we just leave it were we inserted it and report no error */

return(0);
}
```

Memory Management with Tables

If you use paging hardware—or you decide to make your operating system's allocation unit a fixed size and fairly large—you can use tables for memory management. In the case of the paging hardware, you will be forced to use tables of some kind anyway.

Paging hardware will define how the tables are organized. If you don't use the paging hardware, you can decide how you want them set up. Tables are not as elegant as linked lists. Quite often you resort to brute processor speed to get things done quickly. I go into great detail on paging hardware and tables when I describe MMURTL memory management. It combines the use of managing physical and linear memory with tables. It should give you a great number of ideas, and all the information you need to use paging hardware on the Intel and work-alike systems. If you want to use tables, and you're not using paging hardware, you can still look at the setup of the tables for paging. You can expand on them if needed.

Tracking Physical Memory

If you use some form of paging, you will have two memory spaces to track. The two must be synchronized. In the memory-management routines earlier, you assumed you were working with a flat linear space. You can still use paging under the system described earlier; you will simply have to ensure that when you hand out a linear address, there are physical pages assigned to it. This means your operating system will be tracking linear and physical memory at the same time.

When you work with paging hardware, you will deal with a fixed size allocation unit. It will be the page's size. It will be greater if you keep a certain number of pages grouped together in a cluster.

This will almost certainly lead to some form of table use. Paging hardware uses tables to track the relationship between physical and linear memory. Your operating system is responsible for ensuring they are updated properly.

You could also use linked lists to manage the physical memory underneath the linear address space if you like. Something to remember about working with physical memory when paging is used, is that you can't address it until it has been identified in the hardware page tables that you manage.

Initialization of Memory Management

I won't discuss loading the operating system here because it's covered in Chapter 7, "OS Initialization," but once the operating system is loaded, you need to ensure that you initialize memory so that it takes into account all memory used by the OS code and data. This is one of the first things you will do during initialization of memory management.

You must consider where the operating system loads. You may want your operating system code and data at the very top of your linear memory address space. Wherever you load it, it becomes allocated memory right away.

From there, you may also have to take into account any hardware addresses that may not be allocated to applications or the operating system. Some processors use memory addresses from your linear space for device I/O. Intel has a different scheme called port I/O. I'll discuss that in Chapter 6, "The Hardware Interface." If you use a processor that uses hardware I/O from the memory space, it will generally be in a block of addresses. You will allocate these up front.

If you use paging, you have a lot of work to do during the initialization process. You must set up tables, add the translations you need and are already using, and turn on the paging hardware. The hardware paging is usually not active when the processor is reset. Most paging hardware also requires that the non-paged addresses you are executing during initialization match the paged linear addresses when you turn on paging. This means that if you want your operating system code and data in a linear address beyond the physical range, you will have to move

it all, or load the rest of it *after* you turn on paging. This can get complicated. I recommend you leave the initial operating system memory block in an address range that matches physical memory. But it's up to you.

Another chore you'll have is to find out just how much memory you have on the system. Some hardware platforms have initialization code stored in ROM (executed at processor reset) that will find the total and place it somewhere you can read, such as battery backed-up CMOS memory space. You may not always be able to depend on this to be accurate. Batteries fail, and you should take that into consideration.

Memory Protection

Protected memory implies that the processor is capable of signaling you when a problem is encountered with memory manipulation or invalid memory address usage.

This signaling will be in the form of a hardware trap or interrupt. The trap may not actually be signaling a problem, but may be part of the system to help you manage memory. For instance, demand-paged systems must have a way to tell you that some application is trying to address something in a page that isn't in memory right now.

Other forms of traps may be used to indicate that an application has tried to access memory space that doesn't belong to it. It generally means the application has done some bad pointer math and probably would be a candidate for shut-down. If your processor gives you these types of options, it will generally be associated with page tables that you are using to manage your linear address space. In some cases, you will have independent tables for each application.

You will have to investigate each type of interrupt, trap, fault or other signaling method employed by the processor or paging hardware and determine how you will handle all those that apply to the type of system you design.

An Intel-Based Memory-Management Implementation

I chose the Intel processors to use for MMURTL, so the best thing I can do is provide you with a very detailed description of the memory model and my implementation.

As described previously, I use a virtual paged memory model. I may add demand paging to it another time, but it serves my purposes quite well without demand paging.

I depend on the paging hardware very heavily. It can be very complicated to use on any system, but how I use it may give you some ideas even I haven't thought of.

A Few More Words on Segmentation

Even though I continue to refer to Intel as the segmented processors, the concept of program segments is an old one. For instance, programs are quite often broken down into the code segment, the initialized data segment, uninitialized data segment, and many others. I do use a very small amount of segmentation, and I even use the Intel segment registers, but in a very limited fashion. If you have no desire to use them, set them all to one value and forget them.

If you are familiar with segmented programming, you know that with MS-DOS, programs generally have one data segment (which is usually shared with the stack) and one or more code segments. This is commonly referred to as a Medium Memory Model program. In the 80x86 Intel world, there are Tiny, Small, Compact, Medium, Large, and Huge models to accommodate the variety of segmented programming needs. The variety of models is too complicated for me and is no longer necessary on these powerful processors. I want only *one* program memory model. The program memory model I chose is most analogous to the small memory model that has two segments. One is for code and the other is for data and stack. This may sound like a restriction until you consider that a single segment can be as large as all physical memory and even larger with demand page memory.

I use almost no segmentation. The operating system and all applications use only three defined segments: the operating system code segment, the application code segment, and one data segment for all programs and applications. The fact that the operating system has its own code segment selector is really to make things easier for the operating system programmer and to permit protection within the operating system pages of memory. Making the operating system code zero-based from its own selector is not a necessity, but nice to have. This could change in future versions, but the change will have no affect on applications.

The "selectors" (segment numbers for those coming from real mode programming) are fixed. The selector values must be a multiple of eight, and I chose them to reside at the low end of the Global Descriptor Table. These will never change in MMURTL as long as they are legal on Intel and work-alike processors.

The operating system code segment is 08h.

The user code segment is 18h.

The common data segment is 10h.

MMURTL's memory management scheme allows you to use 32-bit data pointers exclusively. This greatly simplifies every program you write. It also speeds up the code by maintaining the same selectors throughout most of the program's execution. The only selector that will change is the code selector as it goes through a call gate into the operating system and back again. This means the *only* 48-bit pointers you will ever use in MMURTL are for an operating system call address (16-bit selector, 32-bit offset).

How MMURTL Uses Paging

MMURTL uses the Intel hardware-based paging for memory allocation and management. The concept of hardware paging is not as complicated as it first seems. Getting it straight took only half my natural life.

MMURTL really doesn't provide memory management in the sense that compilers and language systems provide a heap or an area that is managed and cleaned up for the caller. I figured that an operating system should be a "wholesale dealer" in memory. If you want just a few bytes, go to your language libraries for this trivial amount of memory. My thought was for simplicity and efficiency. I hand out (allocate) only whole pages of memory as they are requested, and I return them to the pool of free pages when they are deallocated. I manage *all* memory in the processor's address space as pages.

A *page* is four kilobytes (4K) of contiguous memory. It is always on a 4K boundary of physical as well as linear addressing.

Paging allows you to manage physical and linear memory address with simple table entries. These table entries are used by the hardware to translate, or map, *physical* memory to what is called *linear* memory. Linear memory is what applications see as their own address space. For instance, you can take the very highest 4K page in physical memory and map it into the application's linear space as the second page of its memory. This 4K page of memory becomes addresses 4096 through 8191, even though it's really sitting up at a physical 16MB address, if you had 16MB of RAM. No, it's not magic, but it's close.

Page Tables (PTs)

The tables that hold these translations are called *page tables* (PTs). Each entry in a PT is called a *page table entry* (PTE). There are 1024 PTEs in every PT. Each PTE is four bytes long. (Aren't acronyms fun? Right up there with CTS—Carpal Tunnel Syndrome).

With 1024 entries (PTEs) each representing four kilobytes, one 4K page table can manage 4MB of linear/physical memory. That's not too much overhead for what you get out of it.

Here's the tricky part. The operating system itself is technically not a *job*. Sure, it has code and data and a task or two; but most of the operating system code—specifically, the kernel—runs in the task of the job that called it. The kernel itself is never scheduled for execution. Because of this, the operating system really doesn't own any of its memory. The operating system is *shared* by all the other jobs running on the system. The page tables that show where the operating system code and data are located get mapped into *every* job's memory space.

Page Directories (PDs)

The paging hardware needs a way to find the page tables. This is done with something called a *page directory* (PD). Every job gets its own PD. You could design your system with only one page directory if you desire.

Each entry in a PD is called a *page directory entry* (PDE). Each PDE holds the *physical* address of a page table. Each PDE is also four bytes long. This means you can have 1024 PDEs in the PD. Each of the PDEs points to a PT, which can have 1024 entries, each representing 4K of physical memory. If you get your calculator out, you'll see that this allows you to map the entire 4GB linear address space.

1024×1024×4K (4096) = 4,294,967,296 (4GB)

You won't need this capability any time soon, right? Wrong. What you don't see, because I haven't explained it yet, is that you really *do* need most of this capability, but you need it in pieces.

The Memory Map

The operating system code and data are mapped into the bottom of every job's address space. A job's memory space actually begins at the 1GB linear memory mark. Why so high? This gives the operating system one gigabyte of linear memory space, and the same to each application. Besides, if I add demand paging to MMURTL's virtual memory, an application of a hundred megabytes or more is conceivable.

The map of a single job and the operating system is shown in Table 5.1. The map is identical for every job and service that is installed.

Table 5.1. MMURTL memory map.

Description	*Address Range*
Linear Top	4GB – 1 byte
Dead address space	2GB – LinearTop
Linear Max	2GB (artificial maximum limit)
Job Allocated Memory	1GB + job memory
Data	1GB + stack page(s) + code pages
Code	1GB + stack page(s)
Initial Stack	1GB
Job Memory	1GB (initial stack, code, data)
DLLs (loadable shared code)	
Device Drivers	
OS Allocated memory	0GB + operating system memory
OS Memory	0GB
Linear Base	0GB

Now the pundits are screaming: "What about the *upper* two gigabytes—it's wasted!" Well, in a word, *yes*. But it was for a good cause. (No, I didn't give it to the Red Cross.)

In the scheme of things, the operating system has to know where to find all these tables that are allocated for memory management. It needs a *fast* way to get to them for memory-management functions. Sure, I could have built a separate table and managed it, but it wasn't needed. Besides, I wanted to keep the overhead down. Read on and see what I did.

The processor translates these linear (false) addresses into real (physical) addresses by first finding the current page directory. It does this by looking at the value you (the OS) put into Control Register CR3. CR3 is the *physical address* of the current PD. Now that it knows where the PD is, it uses the upper 10 bits of the linear address being translated as an index into the PD. The entry it finds is the physical address of the page table (PT). The processor then uses the next lower 10 bits in the linear address as an index into the PT. Now it's got the PTE. The PTE *is* the physical address of the page it's after. Sounds like a lot of work, but it does this with very little overhead, certainly less overhead than this explanation.

The operating system has no special privileges as far as addressing physical memory goes. The operating system uses linear addresses (false ones) just like the applications do. This is fine until you have to update or change a PDE or PTE. You can't just get the value out of CR3 and use it to find the PT because it's the physical address (*crash—page fault*). Likewise, you can't just take a physical address out of a PDE and find the PT it points to.

Finding the PD for an application is no problem. When I started the application, I built the PD and stored the *physical* address in the Task State Segment field for CR3, then I put the linear address of the PD in the Job Control Block. This is fine for *one* address per job. However, now you're talking dozens or even hundreds of linear addresses for all the page tables that you can have, possibly with several for each application.

NOTE

The Task State Segment (TSS) is a processor-defined structure used to manage a task. It may be expanded by the operating system to define other items needed for task management.

The Job Control Block (JCB) is defined by the operating system and used to manage programs that are running on the system.

This is how I use the upper 2K of the page directories. I keep the linear address of all the PTs there. Two kilobytes doesn't sound like a lot to save, but when you start talking 10, 20, or even 30 jobs running, it starts to add up.

I make this upper 2K a *shadow* of the lower 2K. If you remember, each PDE has the physical address of each PT. MMURTL needs to know the physical address of a PT for aliasing addresses, and it needs it fast.

Exactly 2048 bytes above each real entry in the PD is MMURTL's secret entry with the linear address of the PT. Well, the secret is out. Of course, these entries are marked "*not used*" so the operating system doesn't take a bad pointer and try to do something with it.

Page Directory Entries (PDEs)

I know you're trying to picture this in your mind. What does this make the page directory look like? Below, in Table 5.2 is a list of the entries in the PD for the memory map shown in Table 5.1. The table assumes the operating system consumes only a few PTEs (one-page table maximum).

Table 5.2. Page directory example.

Entry Number	*Description*
0	Physical address of operating system PT (PDE 0)
1	Empty PDE
...	
256	Physical address of job PT
257	Empty PDE
...	
512	Linear address of operating system PT (Shadow—marked not present)
513	Empty Shadow PDE
...	
768	Linear address of job PT (Shadow—marked not present)
769	Empty Shadow PDE
...	
1023	Last Empty Shadow PDE

This table doesn't show that each entry only has 20 bits for each address and the rest of the bits are for management purposes, but you get the idea. It's 20 bits because the last 12 bits of the 32-bit address are below the granularity of a page (4096 bytes). The low-order 12 bits for a linear address are the same as the last 12 bits for a physical address. As shown, all the shadow entries are marked *not present*, in fact, all of the entries with nothing in them are marked not present. They simply don't exist as far as the processor is concerned. If I desired, I could move the shadow information into separate tables and expand the operating system to address and

handle 4GB of memory, but I'm more interested in conservation at this point in time. If I decided to do it, it would be transparent to applications anyway.

Something else the example doesn't show is that the entry for the physical address of the operating system PT (0) is actually an alias (*copy*) of the page tables set up when memory management was initialized. I don't keep duplicate operating system page tables for each job. That would *really* be a waste.

Allocation of Linear Memory

You now know the mechanics of paging on the Intel processors, and how I use the processor's paging hardware. Now you need to know how MMURTL actually allocates the linear space in each job or for the OS. This is accomplished with three different calls, depending what type of memory you want. `AllocPage()`, `AllocOSPage()`, and `AllocDMAPage()` are the only calls to allocate memory in MMURTL.

`AllocPage()` allocates contiguous linear pages in the job's address range. This is 1GB–2GB. The pages are all initially marked with the user protection level Read/Write.

`AllocOSPage()` allocates contiguous linear pages in the operating system address range. This is 0–1GB. The pages are all initially marked with the system protection level Read/Write, and the entries automatically show up in all jobs' memory space because all the operating system page tables are listed in every job's page directory.

`AllocDMAPage()` allocates contiguous linear pages in the operating system address range, but it ensures that these pages are below the 16MB physical address boundary. Direct Memory Access hardware on ISA machines can't access physical memory above 16MB. `AllocDMAPage()` also returns the physical address needed by the users of DMA. The pages are all initially marked with the system protection level Read/Write.

All `AllocPage()` calls first check to see if there are enough physical pages to satisfy the request. If the physical memory exists, then they must find that number of pages as contiguous, free entries in one of the PTs. If enough free PTEs don't exist in a row, it will create a new PT. All `AllocPage()` calls return an address to contiguous linear memory, or will return an error if it's not available. With a 1GB address space, it's unlikely that the allocation routine won't find a contiguous section of PTEs. You are likely to run out of physical memory (the story of my life).

Deallocation of Linear Memory

When pages are deallocated (returned to the operating system), the caller passes in a linear address, from a previous `AllocPage()` call, along with the number of pages to deallocate. The caller is responsible for ensuring that the number of pages in the `DeAllocMem()` call does not exceed what was allocated. If it does, the operating system will attempt to deallocate as many pages as requested which may run into memory that was allocated in another request, but only from this caller's memory space. If so, there will be no error, but the memory will not be

available for later use. If fewer pages than were allocated are passed in, only that number will be deallocated. The caller will never know, nor should it try to find out, where the physical memory is located, with the exception of DMA users (device drivers).

I've discussed how MMURTL handles linear addresses. Now comes that easy part—managing physical memory.

Allocation of Physical Memory

The fact that the processor handles translation of linear to physical memory takes a great deal of work away from the OS. It is not important, nor do you even care, if pages of memory in a particular job are physically next to each other (with the exception of DMA). The main goal of physical memory management is simply to ensure you keep track of how much physical memory there is, and whether or not it's currently in use.

Physical memory allocation is tracked by pages with a single array. The array is called the Page Allocation Map (PAM).

The PAM is similar to a bit allocation map for a disk. Each byte of the array represents eight 4K pages (one bit per page). This means the PAM would be 512 bytes long for 16MB of physical memory. The current version of MMURTL is designed to handle 64MB of physical memory which makes the PAM 2048 bytes long. Now if I could only afford 64MB of RAM. The PAM is an array of bytes from 0 to 2047, with the least significant bit of byte 0 representing the first physical 4K page in memory (physical addresses 0 to 4095).

For `AllocPage()` and `AllocOSPage()`, you allocate physical memory from the top down. For `AllocDMAPage()` you allocate physical memory from the bottom up. This ensures that even if you install a device driver that uses DMA after your applications are up and running, there will be physical memory below 16MB available (if any is left at all).

The PAM only shows you which pages of memory are in use. It does not tell you to whom they belong. To get this information, you must go to the PDs and PTs.

Loading Things Into Memory

Applications, system services, device drivers, and DLLs must all be loaded into memory somewhere.

Each application (job) gets its own PD. This is allocated along with the new job control block (JCB). It also gets as many pages as it needs for its code, initial stack, and data. It's loaded into these initial pages. Message-based system services are exactly like applications from a memory standpoint. They are simply new jobs.

Device drivers have code and data but no stack. They become part of the operating system and are reached through the standard entry points in the call gates. They are actually loaded into the operating system's memory space with freshly allocated operating system pages. They become part of the operating system in the operating-system address space accessible to everyone.

Dynamic link libraries are the weird ones. They have only code, no data and no stack. Some systems allow DLLs to contain data. This can introduce re-entrancy problems. I wanted them to be fully re-entrant with no excuses. They are there to be used as shared code *only*.

DLLs are also loaded into operating system address space, but the pages they occupy are marked executable by user level code. This means they can be accessed from the user's code with near calls. This also means that the loader must keep track of the PUBLIC names of each call in a DLL and resolve references to them after you load the applications that call them, but this gets away from memory management.

Operating system page tables are aliased as the first tables in each job's PD and are marked as supervisor. PDs and PTs are always resident (no page swapper yet, how could they be swapped?). This is 8K of initial memory management overhead for each job. It will still only be 8K for a 4MB application. Memory overhead per application is one thing you will have to consider in your design.

Messaging and Address Aliases

If you design your memory management so that each application has its own independent range of linear addresses, you'll have to tackle address aliasing. This means you will have to translate addresses so one program can reach another's data.

With a page-based system such as MMURTL, an alias address is actually just a shared PTE. If two programs need to share memory (as they do when using the interprocess communications, such as request/respond messaging), the kernel copies a PTE from one job's PT to another job's PT. Instantly, the second job has access to the other job's data. They share physical memory. Of course they probably won't be at the same linear address, which means you have to fix up a pointer or two in the request block, but that's trivial. The fixup involves just a few instructions.

There is no new allocation of physical memory, and the service doesn't even know that the pages don't actually belong to it, as they "magically" appear inside its linear address space. Of course, if it tries to deallocate them, an error will occur. Paging makes messaging faster and easier. An aliased PTE is marked as such and can't be deallocated or swapped until the alias is dissolved. Aliasing will only occur for certain operating system structures and messaging other than the aliased page tables in the user's page directory for the operating system code and data.

If you decide you want to use the full segmentation capabilities of the Intel processors, then you can also alias addresses using selectors. The Intel documentation has a very good description of this process.

Memory and Pointer Management for Messaging

If you haven't read Chapter 4, "Interprocess Communications," you should do it before you read this section. If you read that chapter first, this section will make more sense.

When an application *requests* a service, the kernel allocates a request block from the operating system and returns a handle identifying this request block to the caller. This request block is allocated in operating system memory space, but at the user's protection level so the service can access it.

The user's two pointers, `pData1` and `pData2`, are aliased into the services memory area and are placed in the request block.

The memory management aspects of the request/respond messaging process work like this:

1. The caller makes a request.
2. The `Request()` primitive (on the caller's side) does the following:
   - Allocates a request block.
   - Returns a request handle to the caller.
   - Places the following into the `RqBlk`:

     linear address of `pData1` (if not 0)

     linear address of `pData2` (if not 0)

     sizes of the `pData1` and `pData2`

     pointer to caller's Job Control Block

     service code

     response exchange

     `dData0` and `dData1`
   - Places a message on the service's exchange with the Request handle in it.
   - Schedules the service for execution.
   - Reevaluates the ready queue, switching tasks if needed.
3. The `Wait()` primitive (on the service's side) does the following:
   - Adds aliases to the service's Page Table(s) for `pData1` and `pData1`.
   - Places aliased linear addresses into `RqBlk`.
   - Returns the message to the service.
4. The service does its thing using the aliased pointers, reading and writing data to the caller's memory areas. When it's done its work, it responds.
5. The `Respond()` primitive (on the service's side) does the following:
   - Removes the aliased memory from the service's PTs.
   - Places the message on the caller's response exchange.
   - Reevaluates the ready queue (switch tasks if needed).
6. The `Wait()` primitive (back on the caller's side) passes the response message to the caller.

Summary of MMURTL Memory Management

The key points to remember and think about when pondering my memory-management scheme (and how you can learn from it, use it, or use pieces of it) are these:

- One page directory (PD) for each job.
- Linear address of the PD is kept in the job control block.
- One or more page tables (PT) for each job.
- One or more PTs for the OS.
- OS PTs are mapped into every job's page directory by making entries in the job's PD.
- OS uses upper 2K of each PD for linear addresses of PTs.
- Physical memory is tracked with a bit map.

What does all this mean to applications programmers? Not much I'm afraid. They don't need to know any of this at all to write a program for MMURTL. Only those of you who will brave the waters and write your own operating system will have to consider documentation directed at the programmers. How they see the interface is very important. For MMURTL, the application programmer needs to know the following:

- The minimum amount of allocated memory is 4K (one page).
- Memory is allocated in 4K increments (pages).
- Jobs can allocate one or more pages at a time.
- Jobs can deallocate one or more pages at a time.
- The programmer tracks his own memory usage.

6

The Hardware Interface

As a resource manager, the operating system provides access to hardware for the programs on the system. This chapter discusses your approach to hardware interfaces and highlights some of the pitfalls to avoid.

Hardware Isolation

The concept of isolating the operating system (and the programmer) from directly accessing or having to control the hardware is a good idea. This concept is called *hardware abstraction* or *hardware isolation*; it implies an additional programmatic interface layer between the hardware and the parts of the operating system that control it. You provide a common, well-defined interface to perform tasks such as moving data to and from a device, or for specific timing requirements.

This well-defined interface does not require hardware-specific information for its operation from the point of view of the operating system. In theory, you therefore can port the operating system to almost any platform, as long as the target platform has similar hardware to accomplish the required functions.

You need to define logical device nomenclatures from the operating system's point of view, and physical device nomenclatures to the code that will be below this abstraction layer (which actually is controlling the hardware). The interface can be designed above or below the device driver interface. The most elegant method (and the most difficult) is when it is done below the device driver code. Not even device driver writers have to know about the hardware aspects.

As good as it sounds, there are drawbacks to this idea. The two obvious drawbacks are increased code size and reduced speed. Any additional layers between the users of the hardware and the hardware itself adds to size and reduces the speed of the interface. The most obvious place this is evident is in the video arena. Adding a somewhat bulky layer between a program and the video hardware definitely slows down operation.

The implementation of this hardware isolation layer also means that you must thoroughly investigate all possible platforms you intend to target. It is not a good idea to try to implement a complete hardware isolation layer without having all the documentation for the target platforms you intend to support.

You can keep your device interfaces as hardware non-specific as possible, however. Don't use any fancy nonstandard functions that might not be supported on some platforms—for example, memory-to-memory DMA transfers.

The CPU

The interface to the central processor unit (CPU) seems almost transparent. It executes your instructions, jumps where you tell it, and it simply runs. But there's always more than meets the eye.

The CPU has well-defined methods of communicating with the operating system. The most common method, aside from the instructions you give it, is the interrupt or trap. Other methods may be in the form of registers that you can read to gather information on the state of the processor, or tasks that are running (or running poorly).

An important consideration is how the processor stores and manipulates its data—how it is accessed in memory. Some processors store the least significant byte of a 4-byte integer at the highest memory address of the 4 bytes, and the most significant byte at the lowest address. The Intel and compatible CPUs have been referred to as "backword" in this respect because they store least-significant bytes at the lowest address, which makes more sense to me. This affects everything from the programmatic interfaces to interoperability with other systems. This often is referred to as the *big-endian, little-endian* problem when dealing with the exchange of information across networks or in data files. Your assembler and compilers will take care of all the details of this for you, and if you have worked with the processor for any length of time, it will be second nature. One other thing this may affect is how you manipulate the stack or access data that is on it.

Timing issues also are involved with the CPU. Most 32-bit CPUs have relatively large internal instruction and data caches. These types of caches can affect the timing of certain OS-critical functions. I have learned how to enable and disable caching on the systems I work with for the purposes of debugging. I haven't run into any timing problems I can attribute to caching, but that doesn't mean that you will not experience these types of problems.

An important point concerning the CPU interface is how you handle interrupts. With some CPUs, you actually can switch tasks on a hardware interrupt instead of just calling a procedure. This capability is handy, but it can consume a good deal of valuable bandwidth. This gets into the issue of *interrupt latency*, which means not being able to handle all your interrupts in a timely fashion. Switching the complete hardware and software context of a task for every interrupt simply slows you down. I've tried both methods, and I recommend interrupt procedures rather than interrupt tasks if at all possible.

You should be sure to familiarize yourself with all aspects of the CPU by reading the manufacturer's documentation. I read many books about my target processor, but I found the most valuable, and most useful, were purchased directly from the processor manufacturer. Your interpretation of their documentation may be superior to that of those who write secondary books on these topics.

The Bus Structure

A *bus* is a collection of electrical signals that are routed through a computer system between components. Bus designs usually are based on standards and given names or numbers such as IEEE-696, ISA, EISA, PCI, or Multibus, just to name a few.

The bus structure on any hardware platform usually is comprised of several buses. The most obvious to note is the bus from the CPU to the rest of the computer system. This is called the main bus, internal bus, or CPU bus.

The CPU bus usually is comprised of three distinct sets of signals for most processors. The first set is the *data bus*, which carries the values that are read from and written to memory or I/O devices. The second is the *address bus*, which determines where in memory data is being read from or written to or which I/O device is being accessed. Finally, the *control bus* usually carries a wide variety of signals, including read and write access lines for memory and I/O, along with things like processor interrupt lines and clock signals.

The CPU bus usually is connected to the main bus. On smaller systems, the main bus may be composed solely of the CPU signals; in that case, the CPU bus and the main bus are the same. The connections between buses usually are through gated devices that are turned on and off depending on who is accessing what bus. These actions are under hardware control, so you generally don't have to worry about them.

The CPU bus usually is not the bus that is connected directly to the external interface devices that you must control. I refer to this bus as the *interface bus*. On the PC-ISA platforms, the external interface bus is called the *Industry Standard Architecture*, or *ISA bus*. There are many extensions to this bus—some proposed, and some that are available now.

The interface bus may not carry all the data, address, or control lines that are found on the main or CPU bus. The PC-ISA bus, for example, is really quite crippled compared to the CPU bus. Only half the data lines make their way out to this bus (it's really a 16-bit bus), and not all the I/O addresses or interrupt lines can be accessed on this bus.

You really don't need to know the exact hardware layouts for these buses and signals, but some of the processor documentation is fairly explicit about the effect of certain instructions concerning some of these signal lines, such as those dealing with interrupts. It is a good idea to familiarize yourself with the basic bus structures on your target hardware platform. Before you plan some grand interface scheme, make sure that you understand the capabilities of the platform you're working with, and especially the interface bus. Quite honestly, I've had to "drop back 10 yards and punt" a time or two.

Serial I/O

Many forms of serial input/output exist. The most common is the relatively low-speed, asynchronous serial communications (RS-232). The electronic devices that handle asynchronous communications are called *UARTs (Universal Asynchronous Receiver Transmitters)*. UARTs actually are very forgiving devices, and they take much of the drudgery away from the programmer that is implementing serial communications. Most varieties of these devices are very similar. The RS-232 device driver included with the MMURTL operating system should prove valuable to you, no matter which device you must code.

Other devices beside communications channels also may be controlled through UARTs. Such devices could include the keyboard or a mouse port, to name a few.

Less commonly found, but still very important, are synchronous communications such as those used with X.25 (an international link and network layer standard), SDLC, or HDLC. All these synchronous communications standards have one thing in common from the operating system writer's point of view: *critical timing.*

Unlike UART devices, USART (the added *S* is for *synchronous*) devices are very timing critical. They work with *frames* or *packets* of data. USARTs have no way, except for the hardware clock timing, to determine when one byte ends and the next byte begins. USARTs generally expect the device driver to provide these packets with very little delay. In fact, in many cases, a delay causes USARTs to abort the frame they're working on and send an error instead. You therefore must ensure that you can get out the entire frame (byte by byte) without a large delay factor involved between each byte. Buffered USARTs assist with this issue, but how your operating system implements its tasking model and its interrupt service routines (ISRs) greatly affects its capability to handle these types of devices.

The issue of *polling* versus *interrupt-driven* control of communications devices crops up in documentation you read about UARTs and USARTs. The concept of polling a device means that your device driver or program always is there to continually check on the device to see when it needs attention. This procedure is performed by repeatedly reading the status registers on the UART/USART in a loop. If you intend to design a true multitasking system, I recommend that you forget about polling and stick with interrupts. Concentrate on the efficiency of all the ISRs on your system.

Parallel I/O

Parallel input/output can include devices for printer interfaces (the infamous Centronics interface); the IEEE-488 General Purpose Interface Bus (GPIB); and, on some systems, even the main bus may be accessible as a parallel interface of sorts (this is most common on laptops and notebook computers).

Parallel buses can move data faster, simply because they are transferring information one byte or one word at a time instead of one single bit at a time in a serial fashion. These devices provide interrupt-driven capabilities just like the serial devices. Quite often, parallel buses are much easier to control than serial devices because they have fewer communications options.

Block-Oriented Devices

Block-oriented devices can include disk drives, network frame handlers, and tape drives, to name a few. The interfaces to these devices enable large volumes (blocks) of data to be moved between the devices and memory in a rapid fashion.

This block movement of data is accomplished in one of several ways. Direct memory access (DMA) is one method, which is discussed in detail later in this chapter. Another method is shared memory blocks—a process in which a hardware device actually takes up a portion of your physical address space, and all you do to transfer the block is write or read those addresses and then tell the device you're done. Another way in which block movement of data is accomplished is through the use of programmed I/O, which is available on some processors such as the Intel series. Programmed I/O uses special instructions to send data to what appears to be an address, but is really a device connected to the bus. The processor may even have some form of high-speed string instructions to enable you to send blocks of data to this array of pseudo addresses. This is how the Integrated Drive Electronics (IDE) interface works on the PC-ISA systems.

The major difference between these forms of block-data transfer is the consumption of CPU time (bandwidth). DMA uses the least CPU bandwidth because it is a hardware device designed to transfer data between memory and devices, or between memory and memory, during unused portions of the CPU clock cycle. You usually do not have a choice as to which method you use; the method is determined by the interface hardware (the disk or tape controller hardware, or the network card).

The Keyboard

The keyboard may or may not be a part of the hardware to control on your system. If you have a platform that communicates with its users through external terminals, you simply may have to deal with a serial device driver.

On many systems, the keyboard is an integral part of the hardware. This hardware usually is a small processor or a UART/CPU combination. If this is the case, the keyboard hardware is accessed through the I/O port (or memory address), and you need to know how to set it up and operate the device.

The keyboard on many systems is preprogrammed to provide series of bytes based on some translation table within the keyboard microprocessor. Keyboard users prefer a familiar set of codes to work with, such as ASCII (American Standard Code for Information Interchange) or Unicode, which is a relatively new multibyte international standard. You are responsible for the translations required from the keyboard device if they aren't already in the desired format.

Interrupts are the normal method for the keyboard device to tell you that a key has been struck; this requires an interrupt service routine. The keyboard usually can be treated as just another device on your system as far as a standardized interface, but you must realize the importance of this device. Nothing infuriates an experienced user more than an undersized type-ahead buffer or losing keystrokes. You probably can imagine the veins popping out on my forehead when I reach the huge 15-keystroke limit of MS-DOS.

Keyboard handling in a multitasking system can present some pretty interesting problems. One of these problems is how to assign and distinguish between multiple programs asking for keystrokes. Your tasking model and your programmatic interfaces determine how this is accomplished.

On the PC-ISA platform, the keyboard hardware is integrated tightly into the system. In fact, the keyboard serial device allows more than just keyboard interaction. You even can perform a CPU hardware reset through these same control ports. Quite often, devices perform more than one function on a particular platform. It's worth repeating that having the manufacturer's documentation for the platform, and even the device manufacturer's manuals (builders of the integrated circuits), is worth more than I can indicate to you.

Video

Like the keyboard, you may not have any video hardware at all if your user I/O is through external terminals. All the output would be via a serial port. However, most platforms have the video hardware built in or directly accessible through the interface bus.

Many books have been written on the control of video hardware. This is true especially for the IBM PC-AT compatible platforms. You need to understand certain aspects of video hardware, however, because the direct hardware interaction on your platform affects thing like memory-management techniques that you must design or implement.

The two basic forms of video display are character-based and bit-mapped graphics. Each of these displays requires a completely different approach to how you interface video with your operating system. In most cases, you have direct access to video memory somewhere in the array of physical addresses that your CPU can reach.

In character-based systems, this video memory contains character codes (ASCII, for example) that the video hardware translates into individual scan lines on-screen to form the characters. This method consumes the least amount of your address space. One byte per character and maybe one byte for the character's color or attribute are required. On an 80-column-by-25-line display, this method consumes only 4000 bytes.

Bit-mapped graphics-based systems are much more complicated to deal with, and can consume a much larger area of your address space. Graphics systems must have an address for each bit or *pixel* (picture element) of information you want to manipulate or display. In a monochrome system with a decent resolution, this area can be 600 or more bits across and as many as 500 bits vertically:

$600 \times 500 \div 8$ (size of a byte in bits) = 37,500 bytes of memory space

A very common resolution on many systems is 1280×1024.

If you want 16 colors for each bit, this would require three additional arrays to provide the independent color information. These arrays could consume a vast amount of memory. Video

hardware manufacturers realize this problem. Manufacturers provide methods to allow these arrays to occupy the same physical address space, and they give you a way to switch between the arrays. This capability is known as *planar pixel* access. Each bit array is on its own plane. Another method is to make each nibble of a byte represent the four color bits for each displayed bit on-screen. Then video memory is broken into four or more sections, and you can switch between the sections. This capability is known as *packed pixel* access. Even though you can access each of these memory sections in parallel, it is still a vast amount of memory to access. A 256-color system, for example, contains 16 planes at 37,500 bytes (using the 600×500 size sample). That equates to 600,000 bytes. Only one plane of the total memory consumes your address space, because the video interface cards have RAM that they switch in and out of your address space. The efficiency of your video routines therefore has a large effect on the apparent system speed as seen by the user.

These methods of displaying video information require you to allocate an array of OS-managed memory that is for the video display subsystem. The video hardware usually must be programmed to tell it where this memory is located. How much memory you require is determined by what methods your video hardware uses.

Other aspects of the video subsystem also require your attention. Two such aspects to consider are cursor positioning, which is hardware-controlled on character-based systems, and timing, or when you write to the video display area.

In this age of graphics-based systems and graphical user interfaces, you need to take graphics into consideration even if you build a character-based system first. Your graphics implementation might have an effect on your memory management design or even your tasking model. I considered future video and graphics requirements with MMURTL. I even cheated a little and left the video subsystem set up the way the boot ROM left it because it suited my purposes. I did, however, have to research hardware cursor control, and I also allocated memory for large graphics arrays during initialization. See Chapter 26, "Video Code," for descriptions of each of the calls I designed for the video interface. I was interested in character-based, multiple, non-overlapping video users. Your requirements probably are different.

The concept of a message-based operating system plays well into event-driven program control (such as a windowing system). I took this into consideration also.

I recommend that you purchase a book that details the video hardware that you intend to control long before you implement your memory management, interprocess communications, and tasking model.

One thing I should mention is that many video subsystems may provide video code in ROM to control their hardware. In many cases, this code is of little use to you after the system is booted. In some cases, the video ROM's only purpose is to test and set up the video hardware during the testing and initialization phases of the boot process. Video-handling code even may be provided in the form of a BIOS ROM (Basic Input/Output System). This code can be useless if it does not execute properly in your system (16-bit code in a 32-bit system, for example).

Make certain that the video documentation you acquire documents *hardware* access and not simply access to the code found in ROM.

Direct Memory Access (DMA)

You find DMA hardware on almost all platforms you work with, except maybe an embedded system board. DMA devices have several control lines that connect to the CPU and to the devices that use it. These control lines provide the necessary "handshakes" between the device and the DMA hardware. Programmers that write device drivers have often faced the chore of learning all the intricacies of DMA hardware.

In a multitasking system that may have several DMA users, the operating system must ensure harmony between these users. If the programmers follow the rules for programming the DMA hardware to the letter, you will have no problems, but this means that the programmers must be well versed on DMA hardware-control requirements. A better plan is to provide users with a programmatic interface to use the DMA hardware, which takes care of the synchronization and details of DMA programming.

DMA hardware also requires that the programmer know the physical addresses with which he or she is working. Direct memory access is a hardware device that operates outside the virtual address translations that may be taking place within the paging hardware of the processor or external paged memory management unit.

I'll show you what I provided for users of DMA. With minimal changes, you can use this code on almost any system, taking into account the differences in control registers and commands on the target system's DMA hardware. I provided a method to allocate memory that returns the physical address programmers require, which I touched on in Chapter 5, "Memory Management." I also gave users high level calls to set up the DMA hardware for a move, and to query the status of the move. The DMA hardware calls are `SetUpDMA()` and `GetDMACount()`. The code for these calls follows:

```
.DATA
.INCLUDE MOSEDF.INC
.CODE
```

The code is larger than it could be, because I have expanded out each channel instead of using a simple table for the registers. Optimize it as you please.

There really is no data segment for DMA, but the standard include file MOSEDF.INC is used for error codes that may be returned to the caller.

With 8-bit DMA, the lower word (bits 15 through 0) is placed into the address registers of the DMA, and the page register is the next most significant byte (bits 23 through 16). With word, DMA moves (channels 5 through 7), address bits 16 through 1 are placed in the address registers, and 23 through 17 are placed in the page register. Bit 16 is ignored by the page register. The page registers determine which 64K or 128K section of memory is being accessed.

There are two 4-channel DMA devices. One of the channels from the second device is fed into a channel on the first device. This setup is called a *cascaded* device. The following equates are the register addresses for these two devices on ISA hardware:

```
;========== DMA Equates for DMA and PAGE registers =========
; DMA 1 Port addresses and page registers
DMA10Add     EQU  00h     ;Ch 0 Address
DMA10Cnt     EQU  01h     ;Ch 0 Word Count
DMA11Add     EQU  02h     ;Ch 1 Address
DMA11Cnt     EQU  03h     ;Ch 1 Word Count
DMA12Add     EQU  04h     ;Ch 2 Address
DMA12Cnt     EQU  05h     ;Ch 2 Word Count
DMA13Add     EQU  06h     ;Ch 3 Address
DMA13Cnt     EQU  07h     ;Ch 3 Word Count
DMA1StatCmd  EQU  08h     ;Read Status/Write Command
DMA1RqReg    EQU  09h     ;Read/Write DMA Rq Register
DMA1RCmdWbm  EQU  0Ah     ;Read Command/Write Single bit mask
DMA1Mode     EQU  0Bh     ;Read/Write Mode register
DMA1FF       EQU  0Ch     ;Writing this address clears byte ptr flip flop
DMA1Clear    EQU  0Dh     ;Write causes MASTER Clear (Read from Temp Reg)
DMA1ClrMode  EQU  0Eh     ;Rd clears mode reg count/Wr Clr ALL mask bits
DMA1MskBts   EQU  0Fh     ;Read/Write DMA Rq Mask Register

; DMA 2 Port addresses

DMA20Add     EQU  0C0h    ;Ch 0 Address
DMA20Cnt     EQU  0C2h    ;Ch 0 Word Count
DMA21Add     EQU  0C4h    ;Ch 1 Address
DMA21Cnt     EQU  0C6h    ;Ch 1 Word Count
DMA22Add     EQU  0C8h    ;Ch 2 Address
DMA22Cnt     EQU  0CAh    ;Ch 2 Word Count
DMA23Add     EQU  0CCh    ;Ch 3 Address
DMA23Cnt     EQU  0CEh    ;Ch 3 Word Count
DMA2StatCmd  EQU  0D0h    ;Read Status/Write Command
DMA2RqReg    EQU  0D2h    ;Read/Write DMA Rq Register
DMA2RCmdWbm  EQU  0D4h    ;Read Command/Write Single bit mask
DMA2Mode     EQU  0D6h    ;Read/Write Mode register
DMA2FF       EQU  0D8h    ;Writing this address clears byte ptr flip flop
DMA2Clear    EQU  0DAh    ;Write causes MASTER Clear (Read from Temp Reg)
DMA2ClrMode  EQU  0DCh    ;Rd clears mode reg count/Wr Clr ALL mask bits
DMA2MskBts   EQU  0DEh    ;Read/Write DMA Rq Mask Register

;DMA Page register by DRQ/DACK number

DMAPage0     EQU 87h      ;DMA DACK0 Page register
DMAPage1     EQU 83h      ;    DACK1 (etc. etc. etc.)
DMAPage2     EQU 81h
DMAPage3     EQU 82h
DMAPage5     EQU 8Bh
DMAPage6     EQU 89h
DMAPage7     EQU 8Ah
```

The following internal call sets up the initial DMA channel values for both chips to most probable use. This includes the cascade mode for generic channel 4, which is DMA channel 0 on DMA chip 2. This is called one time during initialization of the operating system:

```
PUBLIC InitDMA:
    MOV AL, 04              ;Master disable
    OUT DMA1StatCmd, AL
    OUT DMA2StatCmd, AL

    XOR AL, AL              ;MASTER CLEAR (same as hardware reset)
    OUT DMA1Clear, AL
    OUT DMA2Clear, AL

    XOR AL, AL              ;All commands set to default (0)
    OUT DMA1StatCmd, AL
    OUT DMA2StatCmd, AL

    MOV AL, 40h             ;CH 0 DMA 1
    OUT DMA1Mode, AL        ;
    MOV AL, 0C0h            ;CH 0 DMA 2  (Cascade Mode)
    OUT DMA2Mode, AL        ;

    MOV AL, 41h             ;CH 1 DMA 1 & 2
    OUT DMA1Mode, AL        ;
    OUT DMA2Mode, AL        ;

    MOV AL, 42h             ;CH 2 DMA 1 & 2
    OUT DMA1Mode, AL        ;
    OUT DMA2Mode, AL        ;

    MOV AL, 43h             ;CH 3 DMA 1 & 2
    OUT DMA1Mode, AL        ;
    OUT DMA2Mode, AL        ;

    XOR AL,AL
    OUT DMA1ClrMode, AL     ;Enable ALL DMA 1 Channels
    OUT DMA2ClrMode, AL     ;Enable ALL DMA 2 Channels
    RETN
```

The `PUBLIC` call `DMASetUp()` sets up a single DMA channel for the caller. DMA is crippled because it cannot move data across 64K physical boundaries for a byte-oriented move, or 128K boundaries for a word move. I left it up to the caller to know whether their physical addresses violate this rule. If they do, the segment is wrapped around and data is moved into the lower part of the current 64K segment (not into the next segment, as you might assume).

The caller sets the type of DMA operation (In, Out, or Verify). For channels 5, 6, and 7, the address and count must be divided by 2 for the DMA hardware. I do this for the callers, so they always specify byte count for the setup call, even on word moves.

The `PUBLIC` call `DMASetUp()` follows:

```
;   DmaSetUp(dPhyMem, sdMem, dChannel, dType, dMode)
;      EBP+  28       24     20        16     12
;
;   dPhyMem is physical memory address
;   sdMem is nBytes to move (STILL bytes for word xfers, we do math)
;   dChannel (0,1,2,3,5,6,7)
;   dType - 0 = Verify, 1 = In (Write Mem), 2 = Out (Read memory)
;   dMode - 0 Demand Mode, 1 Single Cycle, (Floppy Disk I/O uses 1)
;           2 Block, 3 Cascade
```

```
;
;
PUBLIC __DMASetUp:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EAX, [EBP+12]       ; dMode
        CMP EAX, 04             ; Mode must be < 4
        JB DMAModeOK
        MOV EAX, ErcDMAMode
        JMP DMAEnd
DMAModeOK:
        SHL EAX, 6              ; Move Mode bits to 6 and 7
        MOV BL, AL              ; Put it in BL
        MOV EAX, [EBP+16]       ; get dType
        AND BL, 0C0h            ; Set to Verify by default (low bits 0)
        CMP EAX, 0              ; Check fType (Verify?)
        JE DMASelect            ; Yes
        CMP EAX, 1              ; In? (Write)
        JE DMAWrite             ; Yes (if no then fall through to Read)
DMARead:
        OR BL,  00001000b       ; OR read command to BL (OUT)
        JMP DMASelect
DMAWrite:
        OR BL, 00000100b        ; OR write command to BL (IN)

DMASelect:
        MOV EAX, [EBP+20]       ;Jump table for channel selection
        CMP EAX, 0              ;
        JE DMA0
        CMP EAX, 1
        JE DMA1
        CMP EAX, 2
        JE DMA2
        CMP EAX, 3
        JE DMA3
        CMP EAX, 5
        JE DMA5
        CMP EAX, 6
        JE DMA6
        CMP EAX, 7
        JE DMA7
        MOV EAX, ErcDMAChannel
        JMP DMAEnd

DMA0:
        CLI
        MOV AL, 00000100b       ; channel 0 Set Mask for DRQ
        OUT DMA1RCmdWbm, AL     ;
        MOV AL, 00000000b       ; CH0 (CH is last 2 bits)
        OR  AL, BL              ; OR with MODE
        OUT DMA1Mode, AL
        OUT DMA1FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        OUT DMA10Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA10Add, AL        ; Hi Byte
        SHR EAX, 8
        OUT DMAPage0, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]
```

```
        DEC EAX
        OUT DMA10Cnt, AL
        SHR EAX, 8
        OUT DMA10Cnt, AL
        MOV AL, 00000000b       ; channel 0 Clear Mask for DRQ
        OUT DMA1RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd

DMA1:
        CLI
        MOV AL, 00000101b       ; channel 1 Set Mask for DRQ
        OUT DMA1RCmdWbm, AL
        MOV AL, 00000001b       ; CH1
        OR  AL, BL              ; OR with MODE/TYPE
        OUT DMA1Mode, AL
        OUT DMA1FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        OUT DMA11Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA11Add, AL        ; Hi Byte
        SHR EAX, 8
        OUT DMAPage1, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        DEC EAX
        OUT DMA11Cnt, AL
        SHR EAX, 8
        OUT DMA11Cnt, AL
        MOV AL, 00000001b       ; channel 1 Clear Mask for DRQ
        OUT DMA1RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd

DMA2:
        CLI
        MOV AL, 00000110b       ; channel 2 Set Mask for DRQ
        OUT DMA1RCmdWbm, AL
        MOV AL, 00000010b       ; CH2
        OR  AL, BL              ; OR with MODE
        OUT DMA1Mode, AL
        OUT DMA1FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        OUT DMA12Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA12Add, AL        ; Hi Byte
        SHR EAX, 8
        OUT DMAPage2, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        DEC EAX
        OUT DMA12Cnt, AL
        SHR EAX, 8
        OUT DMA12Cnt, AL
        MOV AL, 00000010b       ; channel 2 Clear Mask for DRQ
        OUT DMA1RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd
```

```
DMA3:
        CLI
        MOV AL, 00000111b       ; channel 3 Set Mask for DRQ
        OUT DMA1RCmdWbm, AL
        MOV AL, 00000011b       ; CH3
        OR  AL, BL              ; OR with MODE
        OUT DMA1Mode, AL
        OUT DMA1FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        OUT DMA13Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA13Add, AL        ; Hi Byte
        SHR EAX, 8
        OUT DMAPage3, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        DEC EAX
        OUT DMA13Cnt, AL
        SHR EAX, 8
        OUT DMA13Cnt, AL
        MOV AL, 00000011b       ; channel 3 Clear Mask for DRQ
        OUT DMA1RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd

;NOTE: DMA channels 5-7 are on DMA2 and numbered 1-3 for chip select purposes

DMA5:
        CLI
        MOV AL, 00000101b       ; channel 1 DMA2 Set Mask for DRQ
        OUT DMA2RCmdWbm, AL
        MOV AL, 00000001b       ; CH1 on DMA 2
        OR  AL, BL              ; OR with MODE
        OUT DMA2Mode, AL
        OUT DMA2FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        MOV EBX, EAX            ; Save EBX for page
        AND EAX, 0FFFFh         ; Rid of all but lower 16
        SHR EAX, 1              ; DIV by 2 for WORD Xfer (bits 16-1)
        OUT DMA21Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA21Add, AL        ; Hi Byte
        MOV EAX, EBX
        SHR EAX, 15             ; We only need 23-17 for the page
        OUT DMAPage5, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        SHR EAX, 1              ; DIV by 2 for WORD Xfer
        DEC EAX                 ; One less word
        OUT DMA21Cnt, AL
        SHR EAX, 8
        OUT DMA21Cnt, AL
        MOV AL, 00000001b       ; channel 1 Clear Mask for DRQ
        OUT DMA2RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd
```

```
;
DMA6:
        CLI
        MOV AL, 00000110b       ; channel 2 Set Mask for DRQ
        OUT DMA2RCmdWbm, AL
        MOV AL, 00000010b       ; CH2 on DMA 2
        OR  AL, BL              ; OR with MODE
        OUT DMA2Mode, AL
        OUT DMA2FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        MOV EBX, EAX
        AND EAX, 0FFFFh         ; Rid of all but lower 16
        SHR EAX, 1              ; DIV by 2 for WORD Xfer
        OUT DMA22Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA22Add, AL        ; Hi Byte
        MOV EAX, EBX
        SHR EAX, 15
        OUT DMAPage6, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        SHR EAX, 1              ; DIV by 2 for WORD Xfer
        DEC EAX
        OUT DMA22Cnt, AL
        SHR EAX, 8
        OUT DMA22Cnt, AL
        MOV AL, 00000010b       ; channel 2 Clear Mask for DRQ
        OUT DMA2RcmdWbm, AL
        STI
        XOR EAX, EAX
        JMP DMAEnd
;
DMA7:
        CLI
        MOV AL, 00000111b       ; channel 3 Set Mask for DRQ
        OUT DMA2RCmdWbm, AL
        MOV AL, 00000011b       ; CH3 on DMA 2
        OR  AL, BL              ; OR with MODE
        OUT DMA2Mode, AL
        OUT DMA2FF, AL          ; Clear FlipFLop (Val in AL irrelevent)
        MOV EAX, [EBP+28]       ; dPhyMem
        MOV EBX, EAX
        AND EAX, 0FFFFh         ; Rid of all but lower 16
        SHR EAX, 1              ; DIV by 2 for WORD Xfer
        OUT DMA23Add, AL        ; Lo byte address
        SHR EAX, 8
        OUT DMA23Add, AL        ; Hi Byte
        MOV EAX, EBX
        SHR EAX, 15
        OUT DMAPage6, AL        ; Highest byte (to page register)
        MOV EAX, [EBP+24]       ; sdMem
        SHR EAX, 1              ; DIV by 2 for WORD Xfer
        DEC EAX
        OUT DMA23Cnt, AL
        SHR EAX, 8
        OUT DMA23Cnt, AL
        MOV AL, 00000011b       ; channel 3 Clear Mask for DRQ
        OUT DMA2RcmdWbm, AL
        STI
        XOR EAX, EAX
```

```
DMAEnd:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 20                 ; 20 bytes of junk to dump
```

After the DMA call has been made, the device you are transferring data to or from usually interrupts you to tell you it's done. In cases where you always move a fixed size block such as a floppy disk, you can assume that the block was moved if the device status says everything went smoothly (there was no error from the floppy controller, for example). On some devices, you may not always be moving a fixed-size block. In this case, you have to check the DMA controller to see how many bytes or words were moved when you received the interrupt. The call `GetDMACount()` returns the number of bytes or words left in the DMA count register for the channel specified. For channels 5 through 7, this is the number of words. For channels 0 through 3, this is bytes.

You should note that this value reads one less byte or word than is really left in the channel. This occurs because 0 = 1 for setup purposes. To move 64K, you actually set the channel byte count to 65,535:

```
;
;   GetDMACount(dChannel, pwCountRet)
;      EBP+     16        12
;
;   dChannel (0,1,2,3,5,6,7)
;   pwCountRet is a pointer to a Word (2 byte unsigned value) where
;       the count will be returned. The count is number of WORDS-1
;       for channels 5-7 and BYTES-1 for channels 0-3.
PUBLIC __GetDMACount:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EAX, [EBP+16]       ;Channel
        MOV ESI, [EBP+12]       ;Return address for count
DMACSelect:
        CMP EAX, 0              ;
        JNE SHORT DMAC1
        MOV DX, DMA10Cnt        ;
        JMP SHORT DMACDoIt
DMAC1:
        CMP EAX, 1
        JNE SHORT DMAC2
        MOV DX, DMA11Cnt        ;
        JMP SHORT DMACDoIt
DMAC2:
        CMP EAX, 2
        JNE SHORT DMAC3
        MOV DX, DMA12Cnt        ;
        JMP SHORT DMACDoIt
DMAC3:
        CMP EAX, 3
        JNE SHORT DMAC5
        MOV DX, DMA13Cnt        ;
        JMP SHORT DMACDoIt
```

```
DMAC5:
        CMP EAX, 5
        JNE SHORT DMAC6
        MOV DX, DMA21Cnt        ;
        JMP SHORT DMACDoIt
DMAC6:
        CMP EAX, 6
        JNE SHORT DMAC7
        MOV DX, DMA22Cnt        ;
        JMP SHORT DMACDoIt
DMAC7:
        MOV DX, DMA23Cnt        ;
        CMP EAX, 7
        JE SHORT DMACDoIt
        MOV EAX, ErcDMAChannel  ;No such channel!
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ; 20 bytes of junk to dump

DMACDoIt:
        CLI
        IN  AL,DX               ;
        MOV CL,AL               ;
        IN  AL,DX               ;
        MOV CH,AL               ;CX has words/bytes left in DMA
        STI                     ;
        MOV WORD PTR [ESI], CX  ;
        XOR EAX, EAX            ; No Error
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ; 8 bytes to dump from the stack
```

Timers

A hardware timer is a very important piece of any operating system. Almost everything you do is based around an interrupt from a hardware timer.

A *hardware timer* is basically a clock or stopwatch-like device that you can program to interrupt you once or at set intervals. This interval or time period can range from microseconds to seconds, or even hours in some cases. Internally, hardware timers have a counting device (a register) that is usually decremented based on a system-wide clock signal. You generally can set a divisor value to regulate how fast this register is decremented to zero, which determines the interrupt interval. Quite often, these devices enable you to read this register to obtain the countdown value at any time.

Some platforms may provide multiple hardware timers that you can access, program, and read as required. One or more of these timers may be dedicated to hardware functions on the system board, and not available to the operating system interrupt mechanisms directly. The operating system may be responsible for programming these dedicated timers, however, or at least for making sure that the system doesn't interfere with the timers after they are set up by ROM initialization prior to boot time.

If you have timers that perform system functions for the hardware, they probably are initialized at boot time and you can forget them. These timers may be for things like DMA refresh or some form of external bus synchronization.

The Priority Interrupt Controller Unit (PICU)

The PICU is a very common device found on most platforms. Usually, the processor has one or two electrical signal lines that external hardware uses to indicate an interrupt. A *hardware interrupt* is a signal from a device that tells the CPU (the operating system) that it needs servicing of some sort.

As you can imagine (or have realized), many devices interrupt the processor. Therefore, some form of multiplexing must take place on the CPU interrupt signal line. The device that handles all the devices that may interrupt, and then eventually feeds one signal to the CPU, is the PICU. A PICU has several lines that look like the CPU interrupt line to the devices, and also has the logic to select one of these incoming signals to actually interrupt the CPU.

In the process of an interrupt, the CPU must know which device is interrupting it so that it can dispatch the proper interrupt service routine (ISR). This ISR usually is performed by the CPU, which places a value on the data lines when the CPU acknowledges the interrupt. The PICU takes care of this for you, but you generally must set values to indicate what will be placed on the bus. Your job is to know how to program the PICU. This is not very complicated, but it is critical that it's done correctly. There are instructions to tell the PICU when your ISR is finished and interrupts can begin again, ways to tell the PICU to block a single interrupt, and ways tell the PICU how to prioritize all the interrupt lines that it services.

All this information is very specific to the PICU that your platform uses. You will need the hardware manual for code examples.

7

OS Initialization

Initializing an operating system is much more complicated than initializing a single program.

In this chapter, we start with loading the operating system, then go through each section of the basic initialization, including hardware, memory, tasks, and important structures.

Your operating system's tasking and memory models may differ from MMURTL's, but the basic steps to get your operating system up and running will be very similar.

Getting Booted

If you have ever done any research on how an operating system is loaded from disk, you know there is no definitive publication you can turn to for all the answers; it's not every day that someone wants to write an operating system, and it's not exactly easy.

The only hardware platform I've researched for the purpose of learning how to boot an operating system is the IBM-PC ISA-compatible system. If you're using another platform, you will have to do some research. I'm going to give you a good idea of what you need to know, which you can apply to most systems in a generic fashion. This information gives you a very good head start on your studies. If you're using a PC ISA-compatible platform, I've done all your homework.

Boot ROM

When a processor is powered up, it executes from a reset state. Processor designers are very specific about what state they leave the processor in after they accomplish their internal hardware reset. You can find this information in the processor hardware manual or in the processor programming manual from the manufacturer.

The processor's purpose is to execute instructions. The question is, what instruction does it execute first? The *instruction pointer register* or *instruction counter* (whatever they call it for your particular processor) tells the processor which instruction to execute first. (The initial address that is executed is documented in the processor manual.) The operating system writer may not really need to know the processor's initial execution address, because this address is executed by a boot ROM. For this reason, the first executed address usually is somewhere at the high end of physical memory, where boot ROMs usually reside.

The *boot ROM* is read-only memory that contains just enough code to perform a basic system initialization and load (boot) the operating system. When I say "just enough code," that's exactly what I mean. The operating system is located on a disk or other block-oriented device. This device is, no doubt, organized within a file system.

Consider that the boot ROM code knows little or nothing about the file system. If this is true, how does it load a file (the operating system)? It generally doesn't—at least not as a file from the file system.

The Boot Sector

The boot ROM code knows only enough to retrieve a sector or two from the disk to some location in RAM, and then execute it. This process amounts to reading a sector into a fixed address in memory, and then jumping to the first address of the sector. In the PC world, this first address is known as the *boot sector* of the disk. It usually is the first logical sector of the disk.

This small amount of boot sector code has to know how to get the operating system loaded and executed. This process can be completed in more than one step if required. In other words, the boot sector code may load another, more complicated, loader program that in turn loads the operating system. This process is known as a *multistage boot.* To be honest, 512 bytes (a standard hardware sector size on many systems) is not a lot of code.

If the operating system you are loading is located in a special place on the disk, such as a fixed track or cylinder, then life is easy. You only need enough hardware and file-system knowledge to read the sectors from disk to the location you want them in RAM. But it gets more complicated. When the boot ROM loaded your boot sector, it may have loaded it right where you want to put your operating system. This means that the boot sector code has to relocate itself before it can even load the operating system. This dynamic relocation can eat up more of the precious 512 bytes of code space in the boot sector (unless your boot ROM enables you to load more than one sector, such as a whole track).

You should consider these factors when planning how your operating system boots:

1. Where the boot ROM expects your boot sector or sectors to be on disk.
2. Where this boot sector is loaded and executed.
3. Whether you can leave the boot sector code where the boot ROM loaded it (dynamic relocation may be needed if the boot sector is where your operating system will load).
4. How to find your operating system or first-stage boot program on disk from the boot sector code after it is executed.
5. How the boot sector code will read the operating system or first-stage boot program from disk (the hardware commands or BIOS calls needed to load it into memory, for example).
6. Where your operating system will reside in memory, and whether you have to move it after you load it.
7. Whether there are any additional set-up items to perform before executing the operating system.
8. Where your operating system's first instruction will be, so that you can begin execution.

I don't pretend to know how to boot your system if you're not using the platform I chose. But I'm sure you can gain some insight on what you will have to do if I give a detailed description of what it's like on an IBM PC-AT-compatible system. I will go over each of the eight items on the list to explain how it's handled, and then I'll show you some boot sector code in grueling detail. Explanations of each step follow:

1. In an ISA system, the boot ROM expects your boot sector at what it considers is *logical sector number zero* for the first drive it finds it can read. Some systems enable you to change the boot order for drives, but the standard order on ISA systems is drive A, followed by drive C. Keep in mind that the PC ISA systems have a BIOS (Basic Input/Output System) in ROM, and they have some knowledge of the drive geometry from their initialization routines, which they store in a small work area in RAM.
2. A single boot sector (512 bytes) is loaded to address 7C00 hex, which is in the first 64K of RAM.
3. The 7C00 hex address may be fine for your operating system if you intend to load it into high memory. I wanted mine at the very bottom, and it would have overwritten this memory address, so I relocated my boot sector code (I moved it and executed the next instruction at the new address).
4. How you find your operating system on disk depends on the type of file system and where you stored it. If you intend to load from a wide variety of disk types, your OS may not always be in the same place if it's located after variable-sized structures on the first portion of the disk. This was the case with my system, and also with MS-DOS boot schemes.
5. I was lucky with the PC ISA systems, because they provide the BIOS. Each boot sector provides a small table with detailed disk and file system parameters that enable you to update what the BIOS knows about the disk; then you can use the BIOS calls to load the operating system or first-stage boot program.
6. My operating system data and code must start at address 0 (zero), and runs contiguously for about 160K. This was a double problem. The boot sector was in the way (at 7C00h), as were the active interrupt vector table and RAM work area for the BIOS. This problem lead to two relocation efforts. First, I moved the boot sector code way up in memory, and then I executed the next instruction at the new address. Second, I read in the operating system above the actively used memory and then relocated it when I didn't need the BIOS anymore.
7. Usually, you should do a few things before you actually execute the operating system. Just what these things are depends on the hardware platform. In my case, I have to turn on a hardware gate to access all the available memory, and I also go into 32-bit protected mode (Intel-specific) before executing the first operating system instruction.
8. I know the address of the first operating system instruction because my design has only two areas (data and code), and the code segment loads at a fixed address. If your

operating system's first execution address varies each time you build it, you may need to store this offset at a fixed data location and allow your boot sector code to read it so that it knows where to jump.

The following code is a boot sector to load an operating system from a floppy on a PC ISA-compatible system. It must be assembled with the Borland assembler (TASM) because the assembler I have provided (DASM) doesn't handle 16-bit code or addressing. One thing you will notice right away is that I take over memory like I own it. I stick the stack anywhere I want in the first megabyte of RAM. Why not? I do own it. There's no operating system here now. Examine the following code:

```
.386P
;This boot sector is STUFFed to the gills to do a single
;stage boot of the MMURTL OS which is about 160K stored as
;a loadable image beginning at cluster 2 on the disk. The OS must
;be stored contiguously in each of the following logical sectors.
;The actual number of sectors is stored in the data param nOSSectors.
;The .386P directive is required because I use protected
;instructions.

CSEG    SEGMENT WORD 'Code' USE16
        ASSUME CS:CSEG, DS:CSEG, ES:Nothing

ORG     0h

    JMP SHORT Bootup
    NOP                 ;Padding to make it 3 bytes

;This 59-byte structure follows the 3 jump bytes above
;and is found on all MS-DOS FAT-compatible
;disks. This contains additional information about the file system
;on this disk that the operating needs to know. The boot sector code also
;needs some of this information if this is a bootable disk.
Herald          DB   'MMURTLv1'
nBytesPerSect   DW   0200h       ;nBytes/Sector
nSectPerClstr   DB   01h         ;Sect/Cluster
nRsvdSect       DW   0001h       ;Resvd sectors
nFATS           DB   02          ;nFATs
nRootDirEnts    DW   00E0h       ;Root Dir entries max
nTotalSectors   DW   0B40h       ;nTotal Sectors (0 = <32Mb)
bMedia          DB   0F0h        ;media desc. (worthless)
nSectPerFAT     DW   0009h       ;nSect in FAT
nSectPerTrack   DW   0012h       ;nSectors/track
nHeads          DW   0002h       ;nHeads
nHidden         DD   00000000h   ;nHidden Sectors (first whole track on HD)
nTotalSect32    DD   00000000h   ;nTotalSectors if > 32Mb
bBootDrive      DB   00h         ;Drive boot sector came from
ResvdByte       DB   00h         ;Used for temp storage of Sector to Read
ExtBootSig      DB   29h         ; Ext Boot Signature (always 29h)
nOSSectors      DW   0140h       ; (140 sectors max) Was Volume ID number
ResvdWord       DW   0000h
Volname         DB   'RICH       '   ;11 bytes for volume name
FatType         DB   'FAT12   '      ;8 bytes for FAT name (Type)
```

```
;The following are pointers to my IDT and GDT after my OS loads
; and are not part of the above boot sector data structure.

IDTptr          DW 7FFh                 ;LIMIT 256 IDT Slots
                DD 0000h                ;BASE (Linear)

GDTptr          DW 17FFh                ;LIMIT 768 slots
                DD 0800h                ;BASE (Linear)

;This is where we jump from those first 3 bytes

BootUp:

;This is the Boot block's first instruction from the initial jump
; at the beginning of this code

    CLI                 ;Clear interrupts

    ;Stick the stack at linear 98000h (an arbitrary location)

    MOV  AX,9000h
    MOV  SS, AX
    MOV  SP, 8000h

;Move this boot sector UP to 90000h Linear (dynamic relocation)

    MOV  AX, 09000h     ;Destination segment
    MOV  ES, AX
    XOR  DI, DI
    MOV  AX, 7C0h       ;Source segment
    MOV  DS, AX
    XOR  SI, SI
    MOV  CX, 512
    REP  MOVSB

    ; Now we "jump" UP to where we moved it. It's pseudo jump
    ; that I calculated after assembling it once. Push the
    ; return address, then "return" to it. It works.
    MOV AX, 09000h      ;Segment for new location
    PUSH AX
    MOV AX, 6Fh         ;Offset to new location (Next PUSH)
    PUSH AX
    RETF

; Now set DS equal to ES which is 9000h
    PUSH ES
    POP  DS

;Now we must update the BIOS drive parameter
; table so it can read the disk for us.
;This is done by finding out where it is and writing
; some of the parameters to it from the boot sector.
;The BIOS left a pointer to it at the 1E hex interrupt
; vector location.

    MOV CX, nSectPerTrack  ;Get this while DS is correct
    XOR AX, AX
    MOV DS, AX
```

```
MOV  BX, 0078h      ;Int 1E FDC Params!
LDS  SI, DS:[BX]

MOV  BYTE PTR [SI+4], CL
MOV  BYTE PTR [SI+9], 0Fh

PUSH ES             ;Make DS correct again
POP DS
PUSH DS             ;Save DS through the reset

STI                 ;Must set interrupts for BIOS to work

MOV DL, bBootDrive      ;Required for Disk System Reset
XOR AX, AX

INT  13h            ;Reset Disk Controller (DL has drive num)
JC   SHORT BadBoot  ;Reset failed...

POP DS

;The controller is reset. Now let's read some stuff!!
;We are gonna skip checking to see if the first file
;really IS the OS. We need the space for other code.

MOV  SI, OFFSET MsgLoad
CALL PutChars

;What we do now is calculate our way to the third cluster
;on the disk and read in the total number of OS sectors in
;logical sector order. (3rd cluster is really the first allocated
; cluster because the first 2 are unused)
;The layout of the Disk is:
;   Hidden Sectors (optional)
;   Boot Sector (at logical sector 0)
;   FATS (1 or more)
;   Additional Reserved sectors (optional)
;   Root Directory (n Sectors long)

XOR AX, AX
MOV  AL, nFATS
MUL  WORD PTR nSectPerFAT
ADD  AX, WORD PTR nHidden   ;
ADC  DX, WORD PTR nHidden+2
ADD  AX, nRsvdSect
MOV  CX, AX             ;Save in CX

;CX now has a Word that contains the sector of the Root

;Calculate the size of the root directory and skip past it
;to the first allocated sectors (this is where the OS or
;stage 1 of a 2 stage loader should be).

MOV  AX,0020h           ;Size of Dir Entry
MUL  WORD PTR nRootDirEnts
MOV  BX, nBytesPerSect
DIV  BX
ADD  AX, CX
```

```
    ;AX is at sector for cluster 0, but cluster 0 and 1 don't exist
    ;so we are really at cluster 2 like we want to be.

    MOV  CX, nOSSectors ;Number of OS sectors to read
    JMP SHORT ContinueBoot

;Bad boot goes here and displays a message then
;waits for a key to reboot (or tries to) via int 19h

BadBoot:
    MOV  SI, OFFSET MsgBadDisk
    CALL PutChars

    XOR  AX,AX
    INT  16h        ;Wait for keystroke
    INT  19h        ;Sys Reboot

PutChars:
    LODSB
    OR   AL,AL
    JZ   SHORT Done
    MOV  AH, 0Eh
    MOV  BX,0007
    INT  10h
    JMP  SHORT PutChars
Done:
    RETN

ContinueBoot:
    MOV  BX, 06000h    ;This is segment where we load the OS
    MOV  ES, BX        ; before we move it down to 0000h

NextSector:            ;This is a single sector read loop
    PUSH AX            ; It's slow, but works well.
    PUSH CX
    PUSH DX
    PUSH ES

    XOR BX, BX         ; ES:BX points to sector read address

; Read a logical sector to ES:BX
; AX has Logical Sector Number
;
    MOV  SI, nSectPerTrack
    DIV  SI                 ;Divide LogicalSect by nSectPerTrack
    INC  DL                 ;Sector numbering begins at 1 (not 0)
    MOV  ResvdByte, DL          ;Sector to read
    XOR  DX, DX             ;Logical Track left in AX
    DIV  WORD PTR nHeads    ;Leaves Head in DL, Cyl in AX
    MOV  DH, bBootDrive
    XCHG DL, DH             ;Head to DH, Drive to DL
    MOV  CX, AX             ;Cyl into CX
    XCHG CL, CH             ;Low 8 bits of Cyl to CH, Hi 2 bits to CL
    SHL  CL, 6              ;  shifted to bits 6 and 7
    OR   CL, BYTE PTR ResvdByte ;OR with Sector number
    MOV  AL, 1              ;Number of sectors
    MOV  AH, 2              ;Read
    INT  13h                ;Tell BIOS to do it!
    JC   SHORT BadBoot
```

```
    MOV  SI, OFFSET MsgDot
    CALL PutChars

    POP  ES
    POP  DX
    POP  CX
    POP  AX

    MOV  BX, ES     ;I increment ES and leave BX 0
    ADD  BX, 20h    ;512 bytes for segment
    MOV  ES, BX     ;BX will be zeroed at top of loop
    INC  AX         ;Next Sector
    LOOP NextSector

    ;At this point we have the OS loaded in a contigous section
    ;from 60000h linear up to about 80000h linear.
    ;Now we disable interrupts, turn on the A20 line, move
    ;the OS down to address 0, set protected mode, and JUMP!

    CLI
    XOR CX,CX
IBEmm0:
    IN AL,64h
    TEST AL,02h
    LOOPNZ IBEmm0
    MOV AL,0D1h
    OUT 64h,AL
    XOR CX,CX
IBEmm1:
    IN AL,64h
    TEST AL,02h
    LOOPNZ IBEmm1
    MOV AL,0DFh
    OUT 60h,AL
    XOR CX,CX
IBEmm2:
    IN AL,64h
    TEST AL,02h
    LOOPNZ IBEmm2

    ;A20 line should be ON Now
    ;So move the OS

    MOV DX, 8000h

    ; Move 64K data chunk from linear 60000h to linear 0

    MOV BX, 06000h
    MOV DS, BX
    XOR SI, SI
    XOR AX, AX
    MOV ES,AX
    XOR DI,DI
    MOV CX, DX
    CLD
    REP MOVSW               ;WORD move
```

```
        ; Move first 64K code chunk from linear 70000h to 10000h

        MOV BX, 07000h
        MOV DS, BX
        XOR SI, SI
        MOV AX,1000h
        MOV ES,AX
        XOR DI,DI
        MOV CX, DX
        REP MOVSW               ;WORD move

        ; Move last code (32K) from linear 80000h to 18000h

        MOV DS, DX              ;DX is 8000h anyway
        XOR SI, SI
        MOV AX,2000h
        MOV ES,AX
        XOR DI,DI
        MOV CX, DX
        REP MOVSB               ;BYTE move

        MOV BX, 9000h
        MOV DS, BX

        XOR EDX, EDX
        MOV DL, bBootDrive      ;OS can find bootdrive in DL on entry

        LIDT FWORD PTR IDTptr   ;Load Processor ITD Pointer
        LGDT FWORD PTR GDTptr   ;Load Processor GDT Pointer

        MOV EAX,CR0             ;Control Register
        OR AL,1                 ;Set protected mode bit
        MOV CR0,EAX
        JMP $+2                 ;Clear CPU prefetch queue with JMP
        NOP
        NOP

        MOV BX, 10h             ;Set up segment registers
        MOV DS,BX
        MOV ES,BX
        MOV FS,BX
        MOV GS,BX
        MOV SS,BX

        ;We define a far jump with 48-bit pointer manually
        ;by defining the bytes that would make the FAR jump
        ;call JMP FAR 08h:10000h. The segment and code size escape
        ;bytes (66h and 67h) are required to indicate we are
        ;referencing a 32-bit segment with a 32-bit instruction.

        DB 66h
        DB 67h
        DB 0EAh
        DD 10000h
        DW 8h

MsgNone     DB '  This is actually padding!     '
MsgBadDisk  DB 0Dh, 0Ah, 'Bad Boot Disk!', 00h
```

```
MsgLoad     DB 0Dh, 0Ah, 'Loading MMURTL', 00h
MsgDot      DB '.', 00h

BootSig   DW 0AA5Fh   ;MS-DOS used this. I left it in.

CSEG    ENDS
        END
```

To assemble and link this into a usable boot sector, use Borland's Turbo Assembler:

```
TASM Bootblok.asm <Enter>
```

This will make an EXE file exactly 1024 bytes in length. The first 512 bytes are the EXE header. It's worthless. The last 512 bytes are the boot sector binary code and data in a ready-to-use format. As you change and experiment with the boot sector, you need to inspect the actual size of the sector and location of some of the code, such as the dynamic relocation address (if you needed to relocate). To generate a list file from the assembly (Bootblok.lst) and a map file from the link (Bootblok.map), use TASM:

```
TASM Bootblok Bootblok Bootblok <Enter>
```

You need some additional information in order to experiment with this boot sector if you are on the PC ISA platforms, such as how to write the boot sector to the disk or read a boot sector from the disk and inspect it.

Most operating systems provide utilities to make a disk bootable. The systems may do this as they format the disk, or after the fact. Some systems force you to make a decision as to whether the disk will be bootable before you format it if they pre-allocate space on the disk for the operating system.

I used the MS-DOS Debug utility to read and write the disk boot sector while experimenting. It's quite easy to do with the `Load`, `Write`, and `Name` commands. To read the boot sector from a disk, find out where Debug's work buffer is by using an `Assemble` command (just press A while in Debug). This command shows you a segmented address. Press Enter without entering any instructions and use the address shown for the `Load` command with the drive, logical sector number, and number of sectors following it. The following code loads the boot sector from the disk to address 219F:0000 (or whatever address you were shown):

```
A <Enter>
219F:0000 <Enter>
-L 219F:0000 0 0 1 <Enter>
```

Now you can write this sector to another disk as a file with the `Name`, `Register`, and `Write` commands:

```
N C:\Bootblok.bin
R BX <Enter>
BX:0 <Enter>
R CX <Enter>
CX:200 <Enter>
W 219F:0000
```

The boot sector from drive 0 is now in a file named Bootblok.bin in the root directory of drive C. If you want the boot sector from your hard disk, you must use the Load command:

```
-L 219Fc0000 2 0 1 <Enter>
```

The only difference was the drive number that immediately follows the segmented address. Debug refers to the drives numerically from 0 to *n*.

After you assemble your boot sector, you can use Debug to load it as an EXE file and write the single 512 sector directly to the disk. Keep in mind that you should *not* do this to your active development drive. The results can be disastrous. From the MS-DOS prompt, type `Debug`, followed by the name of your new executable boot sector. Then write it to the disk. To find where Debug loaded the boot sector, you can use the `Dump` command to display it:

```
C:\>Debug BootBlok.exe <Enter>
-D <Enter>
219F:0000 EB 3E 90 (etc.)
```

The following code dumps the first 128 bytes and provides the address you need to the `Write` command:

```
-W 219F:0000 0 0 1 <Enter>
```

After the boot sector is on the disk, you only need to put the operating system where it expects to find it. The boot sector code expects to find the operating system in a ready-to-run format at the first cluster on the disk and running contiguously in logical sectors until it ends. This is easy to do. Format a disk with MS-DOS 5.0 or higher using the `/U` (unconditional) option, and *without* the `/S` (system) option. Then copy the operating system image to the disk. It will be the first file on the disk and will occupy the correct sectors. After that, you can copy anything you want to the disk. The commands follow:

```
C:\> Format A: /U  <Enter>
C:\> Copy OS.IMG A:\ <Enter>
```

The new boot sector and the operating system are on the disk and ready to go. Good luck. I only had to do it 30 or 40 times to get it right. Hopefully, I've saved you some serious hours of experimentation. If so, I'm happy.

Operating System Boot Image

Depending on the tools you use to build your operating system, it may not be in a form you can load directly from disk. If your operating system is in an executable format from a linker, there will be a header and possibly fix-up instructions contained in it. You therefore must provide a utility to turn the operating system executable into a directly executable image that can be booted and run by the small amount of code in a boot sector.

With MMURTL, I have provided a utility called `MakeImg`. This reads the MMURTL executable and turns it into a single contiguous executable image to be placed on the disk. The data and code are aligned where they would have been had they been loaded with a loader that

understood my executable format. My `MakeImg` utility is included on the CD-ROM, and must be built with the Borland C compiler. Instructions are included in the single source file (MakeImg.c) to properly build it.



If no address fix-ups are required in your operating system executable, you might be able to avoid a need for a utility like this. I couldn't.

Other Boot Options

If the operating system you write will be run from another environment, you may face the chore of booting it from another operating system. This is how I run MMURTL most of the time. I have a program (also provided on CD-ROM) that reads, understands, and loads the image into RAM and then executes it. This program is called MMLoader. It has some additional debugging display code that you easily can eliminate if you want.



This single source file program (MMLoader.C) also should be built with the Borland C compiler, because it has imbedded assembly language. This program basically turns MS-DOS into a $79 loader. It does everything that the boot sector code does, along with reading the operating system from its native executable format (a RUN file).

Basic Hardware Initialization

One of the first things the operating system does after it is loaded and the boot code jumps to the first address is to set up a simple environment. This process includes setting up the OS stack. In a segmented processor (such as the Intel series), the boot code already may have set up the segment registers. In a non-segmented processor (such as the 680x0 series) this wouldn't even be necessary.

At this point, interrupts usually are disabled because there are no addresses in the interrupt vector table. Your operating system already may have some of the addresses hard-coded in advance, but it is not necessary. Disabling interrupts is one of the chores that my initialization code handles.



All the hardware that is resident in the system—such as the programmable interrupt controller unit, the direct memory access hardware, and basic timers—must be initialized before you can set up the rest of the system (the software state). These elements of hardware simply must be initialized to their default states. Chapter 21, "Initialization Code," contains all the code from MMURTL that is used to set up the operating system from the very first instruction executed, up to the point you jump the monitor program. All this code is contained in the source file Main.asm on the CD-ROM. It should help give you a picture of what's required.

Additionally, some of the hardware-related tasks you may have to accomplish are discussed in Chapter 6, "The Hardware Interface."

Static Tables and Structures

How your operating system handles interrupts and memory, and how you designed your basic operating system API and calling conventions determine how many tables and structures you must set up before you can do things like task and memory management.

I had to set up the basic interrupt vector table, the array of call gates (how users get to OS calls), and some basic variables. Chapter 21 shows how I go about this. These first tables are all static tables and do not require any memory allocation. You will have to think about which tables are static and which are dynamically allocated, because this determines whether their initialization occurs before or after memory-management initialization.

After the basic hardware tables are set up, you may realize that most of this code may never be executed again. This leaves you with a few options—the first of which may be to eliminate the code from the operating system after it is executed so that it doesn't take up memory. Another option is just to forget the code if it doesn't take up too much space. A third option is to make the code reusable by breaking up the code into small segments so that the code can be more generic and used for other things. What you do simply may depend on how much time you want to spend. I just left the code alone. It's not that big. If you want to eliminate the code, you can position it at the end of your code section and simply deallocate the memory for it, or not even initially allocate it with your memory-initialization code.

Initialization of Task Management

Initialization of task management, setting up for multitasking, is not trivial. The processor you're working with may provide the task management in the form of structures that the processor works with, but you also may be performing your own task management. Either way, you've got your job cut out for you.

Chapter 3, "The Tasking Model," discusses some of your options for a tasking model. In all the cases discussed, you must understand that the code you are executing on startup can be considered your first task. Its job is to set up the hardware and to get things ready to create additional tasks. If you prefer, you can let this first thread of execution just die after you set up a real first task. The option usually is up to you, but also may depend on hardware task management if you use it. With Intel processors, for example, you need to set up a task state segment (TSS) to switch from, as well as one to switch to. Because I used the Intel task-management hardware, I really had no choice. Being a little conservative, I didn't want to waste a perfectly good TSS by just ignoring it. I could have gone back and added code to reuse it, but why not just let it become a real task that can be switched back to?

Chapter 21 shows what I have to do using the Intel hardware task management. Additional task-management accomplishments may include setting up linked lists for the execution queue (scheduler) and setting variables to keep track of statistical information.

Initialization of Memory

During the basic hardware initialization, you may be required to send a few commands to various pieces of hardware to set up physical addressing the way you need it. This depends on the platform. It may be to turn on or off a range of hardware addresses, or even to set up the physical range of video memory if your system has internal video hardware.

One particular item that falls into this category on the PC ISA hardware platform is the A20 address line gate. This is one example to help you understand that there may be intricacies on your platform that will drive you nuts until you find the little piece of obscure documentation that explains it in enough detail to get a handle on it.

What is the A20 line gate? It all stems back to IBM's initial PC-AT design. As most of you are aware (if you're programmers), the real mode addresses are 20 bits wide. This is enough to cover a full one megabyte. These address lines are numbered A0 through A19. If you use one more bit, you can address above 1MB. While in real mode, the A20 address line serves no purpose. I suppose that this line even caused some problems because programmers provided a way in hardware to turn it off (always 0). It seems that memory address calculations wrapped around in real mode (from 1MB to 0), and the programmers wanted the same problem to exist in the AT series of processors. I think it's kind of like inheriting baldness from your parents. You learn to live with it.

My goal was not to discover why the A20 address line gate existed, but to ensure that it was turned on and left that way. I tried several methods, some of which didn't work on all machines, and ended up sticking with the way it was documented in the original IBM PC-AT documentation. This method seems to work on most machines, although some people still may have problems with the gate. It is controlled through a spare port on the 8042 keyboard controller (actually a dedicated microprocessor). If your machine doesn't support this method, you may have to do some tweaking of the code. Meanwhile, back to the initialization (already in progress).

At this stage of the initialization, you have basic tables set up and tasks ready to go, but you aren't cognizant of memory yet. You don't know how much you have, and you cannot manage or allocate it.

This is where you call the routine that initializes memory management. This piece of code may have a lot to do. The following list is not all-inclusive, but it gives you a good idea of what you may need to do. Most of the list depends on what memory model you chose for your design. If you went with a simple flat model, you will have a pretty easy job. If you went with a full-blown demand paged system, this could be a fairly large chunk of code. Chapter 19, "Memory Management Code," shows what I did using the Intel paging hardware. It is explained in a fair amount of detail. You may need to consider the following points:

- Find out how much physical memory you have.
- Know where your OS loaded and allocate the memory that it is occupying.

- Allocate other areas of memory for things such as video or I/O arrays if they are used on your system.
- Set up the basic linked list structures if you will be using this type of memory management; otherwise, set up the basic tables that will manage memory.
- Make a transition to a paged memory environment, if you used paging hardware and your processor doesn't start out that way (Intel processors don't).

Something of importance to note about the Intel platform and paged memory-management hardware is that your physical addresses must match your linear addresses when you make the transition into and out of paged memory mode. This is important because it forces you to have a section of code that remains, or is at least loaded, where the address match will occur. I took the easy way out (totally by accident) when I left the entire static portion of my operating system in the low end of RAM. If you chose not to do this, it may mean forced dynamic relocation of your entire operating system. It's not that much of a headache, but one I'm glad I didn't face.

Dynamic Tables and Structures

After memory management has been initialized, you can begin to allocate and set up any dynamic tables and structures required for your operating system.

These structures or tables may be allocated linked lists for resources needed for interprocess communications (IPC), memory areas for future tables that cannot be allocated on the fly, or any number of things. The point is, you couldn't allocate memory until memory-management functions were available.

After all the dynamic structures are allocated and set up, you're ready to execute a real program (of sorts). This program generally is part of the operating system, and does things like set up the program environments and load device drivers or system services.

At this point, you usually load a command-line interpreter, or maybe even load up your windowing environment.

8

Programming Interfaces

For an operating system to be useful, the resources it manages must be accessible to the programmers who will implement applications for it. The programmers must be able to understand what the system is capable of and how to make the best use of the system resources.

Entire books have been written to describe "undocumented system calls" for some operating systems. This can happen when operating system developers add a call that they think may not be useful to most programmers—or even confusing, because the call they add, and leave undocumented, may be similar to another documented call on the system.

In the following sections, I describe both the application programming interface and the systems programming interface, including the mechanics behind some of the possible ways to implement them.

Application Programming Interface

The Application Programming Interface (API) is how a programmer sees your system. The functions you provide are a foundation for all applications that run on your system. The design and implementation of an API may seem easy at first, but there are actually many decisions to be made before you begin work.

Documentation, the consistency of the interface design, error handling, and even the mechanical details of the call itself, all have an effect on how the application programmer views the operating system. These are covered in detail in the following sections.

Documenting Your API

Your operating system may be for a single application, an embedded system, public distribution, or your own use and enjoyment. In any of these cases, you need to properly document the correct use of each call. This may sound like a no-brainer, but it turned out to be a time-consuming and very important task for me. Even though I designed each of the calls for the operating system, I found that, on occasion, I had questions about some of the very parameters I selected when I went back to use them. If this happens to me, I can only imagine what will happen to someone walking into it from a cold start.

Chapter 15, "API Specification," shows you how I documented my calls. Providing examples of the calls is very desirable.

Procedural Interfaces

Procedural interfaces are the easiest and most common type of interface for any API. You place the name of the call and its parameters in a parenthetical list and it's done. The call is made, the function does its job, and returns. What could be easier? I can't imagine. But there are so many underlying complications for the system designer it can give you a serious headache. This

is really a layered problem that application programmers shouldn't have to worry about. Quite often though, they do. I'll go over the complications that have crossed my path. The calling conventions of the interface were the largest complications.

Calling Conventions

As a programmer, you're aware that not every language and operating system handles the mechanics of the procedural call in the same way. These are some of the considerations:

- Where are the parameters placed (on the stack or in registers)?
- If the parameters are on the stack, what order are they in?
- If the stack is used, are there alignment restrictions imposed by hardware?
- If registers are used, what do you do with the excess parameters if you run out of register?
- Who cleans the stack (resets the stack pointer) when the call is completed?

These considerations can get complicated. No kidding! With UNIX, life was easy in this respect. UNIX was written with C; it used the standard defined C calling conventions. With newer operating systems always looking for more speed (for us power junkies), faster, more complicated calling conventions have been designed and put into place. It leads to a lot of confusion if you're not prepared for it when you go use a mixed-language programming model. OS/2, for example, has three or four calling conventions. Some use registers, some use the stack, and some use both. Even the IBM C/Set2 compiler has its own unique calling conventions that only work with CSet/2 (no one else supports it). I even have run across commercial development libraries (DLLs) that used this convention, which forced a specific compiler, even though I preferred another (Borland's, as if you couldn't guess).

When a high-level language has its own calling convention, it will not be a problem if this language also provides a method to reach the operating system if its conventions are different.

The following are the two most popular procedural calling conventions:

- Standard C. Parameters are pushed onto the stack from right to left, and the caller (the one that made the OS call) is responsible for cleaning the stack.
- Pascal. (Also known as PLM conventions in some circles.) Parameters are pushed left to right, and the called function is responsible for making the stack correct.

There is a plethora of not-so-standard conventions out there. I'm sure that you know of several. Execution speed, code size, and hardware requirements are the considerations that drive the need to deviate from a standardized calling convention. I'm fairly opinionated on this issue, especially when it comes to speed. In other words, I don't think the few microseconds saved (that's all it really is) by redesigning the mechanics of a standardized interface are really worth it when it comes to an operating system. Not many calls are made recursively or in a tightly looped repetitive fashion to many operating system functions. Your opinion may vary on this. It's up to you.

The hardware requirements also may be one of the deciding factors. I used an Intel-specific processor hardware function for my calls (call gates), and it required a fixed number of parameters. I had no need for variable-length parameter blocks into the operating system, so the C calling conventions had no appeal to me. I decided on the Pascal (PLM) conventions. This makes porting a C compiler a little hectic, but I wasn't going to let that drive my design.

You may not want to go to the lengths that I did to support a specific calling convention on your system. You may be using your favorite compiler and it will, no doubt, support several conventions. If you want to use code generated by several compilers or languages, you surely will want to do some research to ensure that there is full compatibility in the convention that you choose.

Mechanical Details

Aside from the not-so-simple differences of the calling conventions, there are many additional details to think about when designing the API.

Will your call go directly into the operating system? In other words, will the call actually cause the processor to change the instruction pointer to the code you wrote for that particular call? Your answer will more than likely be no.

Many systems use an intermediate library, which is linked into your code that performs some translation before the actual operating system code is executed. There are two reasons this library may be required. First, your code may expect all the data to be in registers, and you may want to maintain a standard procedural calling convention for the sake of the high-level language interface. Second, the actual execution of the call may depend on a hardware mechanism to actually execute the code.

An example of a hardware-calling mechanism is the software interrupt. A *software interrupt* is a method provided by some processors that enables a pseudo-interrupt instruction to trigger a form of a call. This feature causes the instruction pointer to be loaded with a value from a table (the interrupt vector table, for example). The stack is used to save the return address just like a `CALL` instruction, but the stack is not used for parameters. A special instruction is provided to return from the software interrupt. MS-DOS uses such a device on the Intel-compatible processors. This device is extremely hardware dependent, and forces an intermediate library to provide a procedural interface.

The advantages of using the software-interrupt function include speed (from assembly language only), and also the fact that operating system code relocation doesn't affect the address of the call because it is a value in the interrupt vector table. The address of the operating system calls may change every time you load it, or at least every time it is rebuilt. This address change is not sufficient reason to use this type of interface, however. A simple table fixed in memory to use for indirect call addresses, or some other processor hardware capability can provide this functionality without all the registers directly involved.

When the software-interrupt mechanism is used, you may need only one interrupt for all the calls in the operating system. This interrupt is accomplished by using a single register to identify the call. The intermediate library would fill in this value, and then execute the interrupt instruction.

Portability Considerations

Looking at portability might not seem so important as you write your system for one platform. Murphy's law, however, says that as soon as you get the system completed, the hardware will be obsolete. I really hope not, but this *is* the computer industry of the '90s. I can remember reading the phrase "30 or 40 megahertz will be the physical limit of CPU clock speeds." Yeah, right.

The portability I speak of is not the source-code portability of the applications between systems, but the portability of the operating system source code itself. For application source code portability, the procedural interface is the key. For the operating system source, how you design the internal mechanisms of the system calls may save you a lot of time if you want to move your system to another platform.

I thought about this design consideration before I used the Intel-specific call gates, and the call gates are really not a problem. They aid in the interface but don't detract from portability because they were designed to support a high-level calling convention in the first place (procedural interfaces). Things like software interrupts and register usage, however, leave you with subtle problems to solve on a different platform. If portability is near the top of your wish list, I recommend that you avoid a register-driven interface if at all possible.

Error Handling and Reporting

So many systems out there make you go on some kind of wild-goose chase to find out what happened when the function you called didn't work correctly. The systems make you go find a public variable that you're not sure is named correctly for the language you're using, or they make you call a second function to find out why the first function failed. This may be a personal issue for me, but if I could have any influence in this industry at all, it would be to have all systems return a status or error code directly from the function call that would give me some intelligent information about why my call didn't work.

I recommend that you consider providing a status or error code as the returned value from your system functions that really tells the caller what happened. When you need to return data from the call, have the caller provide a pointer as one of the parameters.

Error reporting has special implications in a multitasking environment, especially with multiple threads that may share common data areas or a single data segment. A single public variable linked into your data to return an error code doesn't work well anymore. Give this some serious thought before you begin coding.

Systems Programming Interface

I divide the systems programming interface into two distinct categories. The first category is the interface seen by programmers that write device drivers or any kind of extension to the operating system. The second category is how internal calls are made from inside the operating system to other operating system procedures (public or internal). I never considered the second aspect until I wrote my own operating system.

Internal Cooperation

Understanding how internal calls are made from inside the operating system to other operating system procedures is not as easy as it sounds. This is true especially if you intend to use assembler and high-level languages at the same time. You more than likely will. Many operating systems now are written almost entirely with high-level languages, but underneath them somewhere is some assembly language coding for processor control and status.

Suddenly, naming conventions and register usage can become a nightmare. Select a register-usage plan and try to stick with it. I had to deviate from my plan quite often as you will see, but there was at least a plan to go back to when all else failed.

You also need to investigate exactly how public names are modified by the compiler(s) you intend to use. Is it one underscore before the name? Maybe it was one before and one after, or maybe it was none if a particular compiler switch was active. Do you see what I mean? Internally, I ended up with two underscores for the public names that would be accessed by outside callers, and this wasn't my original plan.

Device Driver Interfaces

For all practical purposes, device drivers actually become part of the operating system. They work with the hardware, and so much synchronization is required with the tasking and memory-management code, that it's almost impossible to separate them.

Some device drivers load after the operating system is up and running, so you cannot put the code entry point addresses into a table in advance. You need a way to dynamically make all the device drivers accessible to outside callers without them knowing where the code is in advance. This capability requires some form of indirect calling mechanism. This mechanism may be a table to place addresses in that the operating system can get to when a standardized entry point is made by an application.

You also have to consider how to make the device appear at least somewhat generic in your system. Can the programmer read a sector from a floppy disk drive or a hard disk drive without caring which one it really is, for example? That's something to think about when you approach your design.

I was familiar with two device driver interface schemes before I started writing my own system. I knew that there were four basic requirements when it came to device drivers:

- Installation of the driver
- Initialization of the driver
- Controlling (using) the driver
- Statusing the driver

You need to consider these requirements in your design. The possibility exists that you may be writing a much more compact system than I did. If this is the case, all your device drivers may be so deeply embedded in your code that you may not want a separate layered interface.

You also must realize that it isn't just the interface alone that makes things easier for the device driver programmer. Certain system calls may have to be designed to allow them to concentrate their programming efforts on the device and *not* the hardware on the platform. You may want to satisfy these requirements by adding calls to isolate them from some of the system hardware such as DMA, timers, and certain memory-management functions.

A Device-Driver Interface Example

With the four basic requirements I mentioned in mind, I set out to design the simplest, fully functional device driver interface I could imagine. Its purpose is to work generically with external hardware, or hardware that appeared external from the operating system's standpoint.

The interface calls I came up with to satisfy my design follow:

```
InitDevDr(dDevNum, pDCBs, nDevices, fReplace)

DeviceOp(dDevice, dOpNum, dLBA, ndBlocks, pData):dError

DeviceInit(dDevice, pInitData, sdInitdata):dError

DeviceStat(dDevice, pStatRet, sdStatRetmax):dError
```

In Chapter 10, "Systems Programming," I provide detail on the use of these calls, so I will not go over them here. I do want to discuss the underlying code that enables the calls to function for multiple device drivers in a system, however. This will give you a good place to start in your design.

A requirement for the operating system to know things about each device driver led me to require that each driver maintain a common data structure called a device control block (DCB). This is a common requirement on many systems. This structure is filled out with items common to all drivers before the driver tells the operating system that it is ready to go into business serving the applications.

The first call listed in this section (`InitDevDr()`) is called from the driver to provide the operating system with the information it needs to seamlessly blend it into the system. You will note that a pointer to its DCB is provided by the driver.

The other three functions are the calls that the operating system redirects to the proper device driver as identified by the first parameter (the device number).

In a multitasking system, you also will find that some device drivers may not be re-entrant. Two simultaneous calls to the driver may not be possible (or desirable). When a user is doing serial communications, for example, this particular device must be protected from someone else trying to use it. I provided a mechanism to prevent re-entrant use of the drivers if the flag was set in the DCB when the driver initialized itself. I use system IPC messaging to block subsequent calls to a driver that indicates it is not re-entrant.

The code example in the following section is from the file DevDrvr.ASM. This is the layer in MMURTL that initializes device drivers and also redirects the three device driver calls to the proper entry points. When you look at the code, pay attention to the comments instead of the details of the code. The comments explain each of the ideas I've discussed here. It's included in this chapter to emphasize the concepts and not to directly document exactly what I've written.

Announcing the Driver to the OS

This is the call that a driver makes after it is loaded to tell the operating system that it is ready to work. This code is part of the operating system. You will see certain validity checks, and you will notice that you allocate an exchange for messaging if the driver is not re-entrant:

```
;  InitDevDr(dDevNum,   pDCBs,   nDevices, dfReplace):dError
;             EBP+24     EBP+20   EBP+16     EBP+12      sParam 16
;
;Local vars
dDevX    EQU DWORD PTR [EBP-4]
prgDevs  EQU DWORD PTR [EBP-8]
nDevs    EQU DWORD PTR [EBP-12]
dExchTmp EQU DWORD PTR [EBP-16]

PUBLIC __InitDevDr:
        PUSH EBP                        ;
        MOV EBP,ESP                     ;
        SUB ESP, 16
        MOV EAX, [EBP+24]               ;Set up local vars
        MOV dDevX, EAX
        MOV EAX, [EBP+20]
        MOV prgDevs, EAX
        MOV EAX, [EBP+16]
        MOV nDevs, EAX

InitDev00:
        CMP dDevX, nDevices             ;Valid DCB number?
        JB InitDev01
        MOV EAX, ErcBadDevNum           ;Not valid DCB number
        JMP InitDevEnd

InitDev01:  ;Now check to see if device is already installed
        ;and whether it's to be replaced

        LEA EBX, rgpDCBs                ;Point EBX to rgpDCB
```

```
        MOV EAX, dDevX                  ;dDevNum
        SHL EAX, 2
        ADD EBX, EAX
        CMP DWORD PTR [EBX], 0          ;pDCBx = 0 if not used yet
        JZ InitDev02                    ;Empty, OK to use
        CMP DWORD PTR [EBP+12], 0       ;OK to replace existing driver?
        JNZ InitDev02                   ;Yes
        MOV EAX, ErcDCBInUse            ;No - error exit
        JMP InitDevEnd

InitDev02:  ;If we got here, we can check DCB items then move ptr

        MOV EAX, prgDevs                ;EAX points to DCB
        CMP BYTE PTR [EAX+sbDevName],12 ;Check Device name size
        JA InitDev03
        CMP BYTE PTR [EAX+sbDevName], 0 ;is Devname OK?
        JA InitDev04

InitDev03:
        MOV EAX, ErcBadDevName
        JMP InitDevEnd

InitDev04:
        ;Now see if there are more devices for this driver

        DEC nDevs                       ;Decrement nDevices
        JZ InitDev05                    ;NONE left
        ADD prgDevs, 64                 ;Next caller DCB
        INC dDevX                       ;Next devnum
        JMP SHORT InitDev00             ;

        ;All error checking on DCB(s) should be done at this point

InitDev05:                              ;Alloc Exch if driver is NOT re-entrant
        MOV EBX, [EBP+20]               ;pDCBs
        CMP BYTE PTR [EBX+fDevReent], 0
        JNZ InitDev06                   ;device IS re-entrant!
        LEA EAX, dExchTmp               ;Allocate device Exchange
        PUSH EAX                        ;into temp storage
        CALL FWORD PTR _AllocExch
        CMP EAX, 0
        JNZ SHORT InitDevEnd

InitDev06:
        ;All went OK so far, now move the DCB pointer(s) into array
        ; and assign exchange from temp storage to each DCB

        MOV EAX, [EBP+16]               ;nDevices
        MOV nDevs, EAX                  ;Set nDev to number of devices again
        LEA EBX, rgpDCBs                ;Point EBX to OS rgpDCBs
        MOV EAX, [EBP+24]               ;dDevNum
        SHL EAX, 2
        ADD EBX, EAX                    ;EBX now points to correct pointer
        MOV EAX, [EBP+20]               ;EAX points to first DCB
        MOV ECX, dExchTmp               ;ECX has semaphore exchange

InitDev07:
        ;Now that EBX, EAX, and ECX are set up, loop through each
```

```
        ;DCB (if more than 1) and set up OS pointer to it, and
        ;also place Exchange into DCB. This is the same exchange
        ;for all devices that one driver controls.

        MOV [EAX+DevSemExch], ECX
        MOV [EBX], EAX
        ADD EBX, 4                      ;next p in rgp of DCBs
        ADD EAX, 64                     ;next DCB
        DEC nDevs
        JNZ InitDev07                   ;Any more DCBs??
        XOR EAX, EAX                    ;Set up for no error

        ;If the device driver was NOT re-entrant
        ;we send a semaphore message to the exchange for
        ;the first customer to use.

        MOV EBX, [EBP+20]               ;pDCBs
        CMP BYTE PTR [EBX+fDevReent], 0
        JNZ InitDev06                   ;device IS re-entrant!
        PUSH ECX                        ;ECX is still the exchange
        PUSH 0FFFFFFFEh                 ;Dummy message
        PUSH 0FFFFFFFEh
        CALL FWORD PTR _SendMsg         ;Let erc in EAX fall through (Was ISend)

InitDevEnd:
        MOV ESP,EBP                     ;
        POP EBP                         ;
        RETF 16                         ;
;
```

A Call to the Driver

This is the call that the user of the device driver makes to initially set up the device, or to reset it after a failure. You can see in the comments how you redirect this call to the proper code in the driver from the address provided in the DCB. The other two calls, `DeviceOp()` and `DeviceStat()`, are almost identical to `DeviceInit()`, and are not presented here.

```
;
;  DeviceInit(dDevNum, pInitData,  sdInitData);
;             EBP+20   EBP+16      EBP+12         Count = 12
;
PUBLIC __DeviceInit:                    ;
        PUSH EBP                        ;
        MOV EBP,ESP                     ;
        CMP DWORD PTR [EBP+20], nDevices ;Valid Device number?
        JB DevInit01
        MOV EAX, ErcBadDevNum            ;Sorry no valid DCB
        JMP DevInitEnd
DevInit01:
        LEA EAX, rgpDCBs
        MOV EBX, [EBP+20]               ;
        SHL EBX, 2                      ;
        ADD EAX, EBX                    ;
        MOV EBX, [EAX]                  ;now EBX points to DCB (maybe)
        CMP EBX, 0                      ;Is there a pointer to a DCB?
        JNZ DevInit1A                   ;Yes
```

```
        MOV EAX, ErcNoDriver            ;NO driver!
        JMP DevInitEnd
DevInit1A:
        CMP BYTE PTR [EBX+DevType], 0   ;Is there a physical device?
        JNZ DevInit02
        MOV EAX, ErcNoDevice
        JMP DevInitEnd

DevInit02:  ;All looks good with device number
        ;so we check to see if driver is re-entrant. If not we
        ;call WAIT to get "semaphore" ticket...

        CMP BYTE PTR [EBX+fDevReent], 0
        JNZ DevInit03                   ;Device IS re-entrant
        PUSH EBX                        ;save ptr to DCB
        PUSH DWORD PTR [EBX+DevSemExch] ;Push exchange number
        LEA EAX, [EBX+DevSemMsg]        ;Ptr to message area
        PUSH EAX
        CALL FWORD PTR _WaitMsg         ;Get semaphore ticket
        POP EBX                         ;Get DCB ptr back
        CMP EAX, 0
        JNE DevInitEnd                  ;Serious kernel error!

DevInit03:
        PUSH EBX                        ;Save ptr to DCB
        PUSH DWORD PTR [EBP+20]         ;Push all params for call to DD
        PUSH DWORD PTR [EBP+16]
        PUSH DWORD PTR [EBP+12]
        CALL DWORD PTR [EBX+pDevInit]
        POP EBX                         ;Get ptr to DCB back into EBX
        PUSH EAX                        ;save error (if any)

        CMP BYTE PTR [EBX+fDevReent], 0 ;Re-entrant?
        JNZ DevInit04                   ;YES

        PUSH DWORD PTR [EBX+DevSemExch] ;No, Send semaphore message to Exch
        PUSH 0FFFFFFFEh                 ;Bogus Message
        PUSH 0FFFFFFFEh                 ;
        CALL FWORD PTR _SendMsg         ;Ignore kernel error
DevInit04:
        POP EAX                         ;Get device error back

DevInitEnd:
        MOV ESP,EBP                     ;
        POP EBP                         ;
        RETF 12                         ;dump params
```

Your interface doesn't have to be in assembly language like the preceding example, but I'm sure you noticed that I added examples of the procedural interface in the comments. If you use assembly language and provide code examples, adding the procedural call examples is a good idea.

This chapter ends the basic theory for system design. The next chapter begins the documentation of and for an operating system.

III

MMURTL–Specific Details

9

Application Programming

This section introduces the MMURTL operating-system interface and provides guidance on programming techniques specific to the MMURTL environment. It also provides information on basic video and keyboard usage as well as examples of multithreading with MMURTL and information on memory-management techniques and requirements. This chapter should be used with the reference manual for the language you are using.

Before writing your first MMURTL program, you should be familiar with the material covered in Chapter 4, "Interprocess Communications," and Chapter 5, "Memory Management." These chapters provide discussions of the concepts behind message-based operating systems and paged memory management. These are important concepts you should understand before you attempt to create or port your first application.

Some of the information provided in this section may not actually be needed in order to code a program for MMURTL. I think you will find it useful, though, as you discover the methods employed by this operating system. It may even be entertaining.

Terminology

Throughout this section, I use several words that may seem language-specific or that mean different things to different programmers. The following definitions are provided to prevent confusion:

Procedure A function in C or Pascal, or a procedure in Pascal, or a procedure in Assembler. All MMURTL operating system calls return values (called *error codes*).

Function Same as *procedure.*

Call Used interchangeably with *function* and *procedure.*

Byte An eight-bit value (signed or unsigned).

Word A 16-bit value (signed or unsigned).

Dword A 32-bit value (signed or unsigned).

Parameter A value passed to a function or procedure, usually passed on the stack.

Argument Same as *parameter.*

Understanding 32-Bit Software

If you have been programming with a 16-bit operating system (for example, MS-DOS), you will have to get used to the fact that all parameters to calls, and most structures (records) have many 32-bit components. If you are used to the 16-bit names for assembler variables, you should be comfortable with MMURTL because I have maintained the Intel and Microsoft conventions of *byte, word,* and *dword.* Most MMURTL parameters to operating system calls are dwords

(32-bit values). If I were a perfectionist, I would have called a 32-bit value a word, but alas, old habits are hard to break and one of my key design goals was simplicity for the programmer on Intel-based ISA systems.

One of the things that make processors different from each other is how data is stored and accessed in memory. The Intel processors have often been called "BackWords" because they store the least significant data bytes in lower memory addresses (hence, the byte values in a word appear reversed when viewed with a program that displays hexidecimal bytes). A byte at address 0 is accessed at the same memory location as a word at address 0, and a dword is also accessed at the same address. Look at the Table 9.1 to visualize what you find in memory:

Table 9.1. Memory alignment.

Address	*0*	*1*	*2*	*3*	*Hex value*
Access as					
byte	01				01h
word	01	02			0201h
dword	01	02	03	04	04030201h

This alignment serves a very useful purpose. Conversions between types is automatic when read as a different type at the same address. This also plays a useful role in languages such as C, Pascal, and assembly language, although it's usually transparent to the programmer using a high-level language.

When you are running the Intel processors in 32-bit protected mode, this also makes a great deal of difference when values are passed on the stack. You can push a dword on the stack and read it as a byte at the same location on the stack from a procedure. MMURTL always uses a dword stack. (Values are always pushed and popped as dwords—32-bit values.) This is a hardware requirement of the 386/486 processors when 32-bit segments are used.

Operating-System Calling Conventions

All operating system calls are listed alphabetically in Chapter 15, "API Specification."

Most operating system calls in MMURTL return a dword (32-bit) error code, which is often abbreviated as `ERC`, `erc`, or `dError`. The descriptions in the API specification (Chapter 15) show it as `dError` to indicate it is a 32-bit unsigned value. These codes convey information to the caller about the status of the call. It may indicate an error, or it may simply convey information that may or may not be an error (depending on what you expect). A list of all documented status or error codes can be found in header files included with the source code.

All public operating-system calls are accessible through call gates and can be reached from "outside" programs. They are defined as *far* (this may be language-dependent) and require a 48-bit call address.

A far call address to the operating system consists of a selector and an offset. The selector is the call gate in the Global Descriptor Table (GDT), while the offset is ignored by the 386/486 processor. The GDT entry for the call gate defines the actual address that is called and the number of dwords of stack parameters to be passed to the call. You can even hard code the far address of the operating-system call because the system calls are at call gates. It doesn't matter where the operating system code is in memory. The call gate will never change unless it becomes obsolete (in which case it will left in during several major revisions of the operating system for compatibility purposes). This means that an assembly-language programmer can make the far call to what appears to be a hard address (but is really a call gate). The addresses of the calls are documented with the source code (header or include files).

Stack Usage

Each of the MMURTL operating-system calls is defined as a function with parameters. Most return a value that is an error or status code. Some do not.

All but a few of the operating-system calls use the stack to pass parameters to the operating-system functions.

All parameters are pushed from left to right and are expanded to 32-bit values. This will be done by the compiler, and in some cases, even the assembler or processor itself. This is because the processor enforces a four-byte aligned stack.

The called operating-system function always cleans the stack. In some C compilers, this is referred to as the Pascal Calling Convention. The CM32 compiler uses this convention.

It is also very easy to use from assembly language. The following is an example of a call to the operating system in assembler:

```
PUSH 20
PUSH 15
CALL FWORD PTR SetXY
```

In the preceding example, `SetXY` is actually a far pointer stored in memory that normally would contain the address of the `SetXY` function. The address is, however, really a call gate, not the real address of the `SetXY` function.

Memory Management

MMURTL uses *all* of the memory in your system and handles it as it as one contiguous address span. If you need a 2MB array, you simply allocate it. Chapter 3, "The Tasking Model," and Chapter 4, "Interprocess Communications," describe this in great detail.

MMURTL has control of all physical or linear memory allocation and hands it out it as programs ask for it. Application programs all reside at the one-gigabyte memory location inside of their own address space. The operating system resides at address 0. This doesn't change any programming techniques you normally use. What does change your techniques is the fact that you are working with a flat address space that is all referenced to one single selector. There are no far data pointers. This also means that a pointer to allocated memory will never start at 0. In fact, when an application allocates pages of memory, they will begin somewhere above one gigabyte (40000000h).

MMURTL really doesn't provide memory management in the sense that compilers and language systems provide a heap or an area that is managed and cleaned up for the caller. MMURTL is a paged memory system. MMURTL hands out (allocates) pages of memory as they are requested, and returns them to the pool of free pages when they are turned in (deallocated). Heap-management functions that allow you to allocate less than 4K at a time should be provided by your high-level language. The library code for the high-level language will allocate pages and manage them for you. This would let you allocate 30 or 40 bytes at a time for dynamic structures such as linked lists (for example, `malloc()` in C).

A MMURTL application considers itself to be the only thing running on the system aside from the operating system and device drivers. Your application sees the memory map described in Table 9.2.

Table 9.2. Basic memory map.

Address Span	*Memory Usage*
7FFFFFFFh	Top of user address space
...	
40000000h	Application base (1GB)
3FFFFFFFh	Top of OS address space
...	
00000000h	OS base address

Even if six applications are running at once, they all see this same "virtual memory map" of the system. The operating system and device drivers reside in low linear memory (0), while all applications and system services begin at the one-gigabyte mark. Paged virtual memory is implemented with the paging capabilities of the 386/486 processor. Multiple applications all load and run at what seems to be the same address range, which logically looks like all programs are loaded parallel to one another. The paging features of the 386/486 enable you to assign physical memory pages to any address ranges you like. Read up on memory management in Chapter 5 if you're really interested.

The operating system provides three basic calls for application memory management. They are as follows:

```
AllocPage(nPages, ppMemRet): dError
DeAllocPage(pMem, dnPages): dError
QueryPages(pdnPagesRet): dError
```

When your program needs a page of memory, you allocate it. When you are done with it, you should deallocate it. There are cleanup routines in the operating system that will deallocate it for you if your program exits without doing so, but remember, it's bad manners not to clean up after yourself. MMURTL is kind of like your mother. She may pick up your dirty socks, but she would much rather that you did so yourself.

Memory-page allocation routines always return a pointer to the new memory. The address of the pointer is always a multiple of the size of a page (modulo 4096).

The memory allocation routines use a *first fit algorithm* in your memory space. It is up to you to manage your own addressable memory space. Your program can address up to 1GB of space. Of course, this doesn't mean you can allocate a gigabyte of memory, but pointer arithmetic may get you into trouble if you allocate and deallocate often and lose track of what your active linear addresses are. The high-level language allocation routine may help you out with this, but I have to address the issues faced by those that write the library allocation routines for the languages.

Consider the following scenario:

1. Your program takes exactly 1MB before you allocate.
2. You allocate five pages. Address 40100000h is returned to you. You now have 20K at the 40100000h address.
3. Then you deallocate three of these pages at address 40100000h (12K). You are left with a hole in your addressable memory area. This is not a problem, as long as you know it can't be addressed.
4. Now you allocate four pages. The memory-allocation algorithm can't fit it into the three-page hole you left, so it will return a pointer to four pages just above the original five you allocated first (address 40105000h). Now you have six pages beginning at 40103000h.

If you attempt to address this "unallocated" area, a Page Fault will occur. This will terminate your application automatically. (Ouch!)

Operating-System Protection

MMURTL provides protection for the operating system and other jobs using the paging-hardware protection.

The operating system code, data, and device drivers run and can only be accessed at the system-protection level (level 0), while system services and applications run at user-level protection. The operating system prevents ill-behaved programs from intentionally or accidentally accessing another program's data or code (from "a pointer in the weeds"). It even prevents applications from executing code that doesn't belong to them, even if they know where it's located (that is, its physical or linear address). MMURTL does not, however, protect programmers from themselves.

This means that in a multitasking environment, a program that crashes and burns doesn't eat the rest of the machine in the process of croaking. In this case, the operating system acts more like a referee than a mother.

Application Messaging

The most basic applications in MMURTL may not even use messaging directly. You may not have to call an operating-system primitive-message routine throughout the execution of your entire program. However, when you write multithreaded applications (those with more than one task), you will probably need a way to synchronize their operations or to "queue up" events between the threads (tasks). This is where non-specific intertask messaging is employed. This is done with the `SendMsg()` and `WaitMsg()` calls.

You may also require the use of a system service for which there isn't a direct procedural interface, in which case you will use `Request()` and `WaitMsg()`.

The following messaging calls are provided by the operating system for applications:

```
SendMsg (dExch, dMsg1, dMsg2): dError
CheckMsg(dExch, pMsgsRet) : dError
WaitMsg(dExch,pqMsgRet):dError
Request(pSvcName, wSvcCode, dRespExch, pRqHndlRet, npSend, pData1,
cbData1, pData2, cbData2, dData0, dData1, dData2 ) : dError
```

Before you can use messaging, you will need an exchange where you can receive messages, or you will need to know of one you can send messages to. The following calls are provided to acquire exchanges and to return them to the operating system when you no longer need them:

```
AllocExch(pdExchRet): dError
DeAllocExch(dExch): dError
```

Normally, applications will allocate all of the exchanges they require somewhere early in the program execution and return them (`DeAllocExch`) before exiting.

Messaging is very powerful, but it can also be destructive (to the entire system). Memory-management protection provided by MMURTL will help some, but you can easily eat up all of the system resources and cause the entire system to fail. MMURTL is not a mind reader. For instance, accidentally sending messages in an endless loop to an exchange where no task is waiting to receive them causes the system to fail. This eats all of your link blocks. The operating system is brought to its knees in a hurry.

Starting a New Thread

Some programs need the capability to concurrently execute two or three things, such as data communications, printing, and user interaction. A simple example would be keeping the time on the user interface updated no matter what else you are doing. In these cases, you can "spawn" a new task to handle the secondary functions while your main program goes about its business.

The operating system provides the `SpawnTask()` call, which gives you the ability to point to one of the functions in your program and turn it into a "new thread" of execution. This new function will be scheduled for execution independently of your main program.

Prior to calling `SpawnTask()`, you need to allocate or provide memory from your data segment for the stack that the new task will use. Other than that, you basically have two programs running in the same memory space. They can even execute the same procedures and functions (that is, share them). The stack you allocate should be at least 512 bytes (128 dwords) plus whatever memory you require for your task. You need to ensure you don't overrun this stack. If you do, only your program will be trashed, but you don't want that, either.

You should be concerned about reentrancy if you have more than one task executing the same code. Reentrancy considerations include modifications to variables in your data segment (two tasks sharing variables), operating system messaging (sharing an exchange), and the allocation or use of system resources such as memory.

Look at the code for the monitor (Monitor.c, provided with the operating-system source code) for a good example of spawning new tasks. Several tasks are spawned in the monitor.

Job Control Block

Applications may sometimes need to see the data in a Job Control Block. Utilities that display information about all of the jobs that are running may also need to see this data.

The `GetpJCB()` call will provide a pointer to allow you to read values from a JCB. The pointer is read-only. Attempts to write directly to the JCB will cause your application to be terminated with a protection violation.

In order to use the `GetpJCB()` call, you need to know your job number, which the `GetJobNum()` call will provide.

Header files included with the source code have already defined the following structure for you.

Listing 9.1. JCB structure definition in C.

```
struct JCBRec {
long            JobNum;
char            sbJobName[14];   /* lstring */
```

```
char          *pJcbPD;        /* Linear add of Job's PD */
char          *pJcbCode;      /* Address of code segment */
unsigned long sJcbCode;       /* Size of code segment */
char          *pJcbData;      /* Adresss of data segment */
unsigned long sJcbData;       /* Size of data segment */
char          *pJcbStack;     /* Addrees of primary stack */
unsigned long sJcbStack;      /* Size of primary stack */
char           sbUserName[30]; /* User Name - LString */
char           sbPath[70];     /* current path (prefix) */
char           JcbExitRF[80];  /* Exit Run file (if any) */
char           JcbCmdLine[80]; /* Command Line - LString */
char           JcbSysIn[50];   /* std input - LString */
char           JcbSysOut[50];  /* std output - LString */
long           ExitError;      /* Error Set by ExitJob */
char          *pVidMem;        /* Active video buffer */
char          *pVirtVid;       /* Virtual Video Buffer  */
long           CrntX;          /* Current cursor position */
long           CrntY;
long           nCols;          /* Virtual Screen Size */
long           nLines;
long           VidMode;        /* 0 = 80x25 VGA color text */
long           NormVid;        /* 7 = WhiteOnBlack */
char           fCursOn;        /* 1 = Cursor is visible */
char           fCursType;      /* 0=UL, 1 = Block */
unsigned char  ScrlCnt;        /* Count since last pause  */
char           fVidPause;      /* Full screen pause  */
long           NextJCB;        /* OS Uses to allocate JCBs */
};
```

The JCB structure is a total of 512 bytes, with unused portions padded. Only the active portions are shown in the preceding listing. All of the names and filenames (strings) are stored with the first byte containing the active size of the string and with the second byte (offset 1) actually holding the first byte of string data.

To change data in the JCB, you must use the calls provided by the operating system. These include the following:

```
SetJobName(pJobName, dcbJobName): dError
SetExitJob(pFileName, dcbFileName): dError
SetPath(pPath, dcbPath): dError
SetUserName(pUserName, dcbUserName): dError
SetSysIn(pFileName, dcbFileName): dError
```

Some of the more commonly used information can be read from the JCB without defining the structure by using the following operating-system-provided calls:

```
GetExitJob(pFileNameRet, pcbFileNameRet): dError
GetCmdLine(pCmdLineRet, pdcbCmdLineRet): dError
GetPath(dJobNum, pPathRet, pdcbPathRet): dError
GetUserName(pUserNameRet, pdcbUserNameRet): dError
```

Basic Keyboard

Internal support is provided for the standard IBM PC AT-compatible 101-key keyboard or equivalent. The keyboard interface is a system service. Chapter 13, "Keyboard Service," provides complete details on the keyboard implementation. A system service allows concurrent access to system-wide resources for applications. The keyboard has a built-in device driver that the keyboard service accesses and controls.

For most applications, the single procedural `ReadKbd()` operating-system call will suffice. It provides access to all standard alphanumeric, editing, and special keys. This call returns a 32-bit value that contains the entire keyboard state. (All shifted states such as Alt and Ctrl are included.) There are tables in Chapter 13 to show all of the possible values that can be returned.

The first of the two parameters to the `ReadKbd()` call asks you to point to where the 32-bit keycode value will be returned, while the second parameter is a flag asking if you want to wait for a keystroke if one isn't available.

Basic Video

Each application and system service in MMURTL is assigned a virtual video buffer that is the same size as the video memory in standard VGA-color character operation. This is 4000 bytes (even though an entire page is allocated—4096 bytes).

The video calls built into the operating system provide TTY (teletype) as well as direct screen access (via `PutVidChars` and `GetVidChar` calls) for all applications.

MMURTL has a basic character device driver built-in, which provides an array of calls to support screen color, character placement, and cursor positioning.

Positions on the screen are determined in x and y coordinates on a standard 80 × 25 matrix (80 across and 25 down).

`X` is the position across a row and referenced from 0 to 79. `Y` is the position down the screen and referenced as 0 to 24.

The character set used is the standard IBM PC internal font loaded from ROM when the system is booted. The colors for `TTYOut`, `PutChars`, and `PutAttrs` comprise 16 foreground colors, 8 background colors, and 1 bit for blinking. Even though the attribute is passed in as a dword, only the least significant byte is used. In this byte, the high nibble is the background, and low nibble is the foreground. The high bit of each is the intensity bit.

Tables 9.3 and 9.4 describe the attributes. These names are also defined in standard header files included with the operating-system code and sample programs.

Table 9.3. Foreground colors (low nibble).

Normal	*Binary*	*Hex*	*Intensity Bit Set*	*Binary*	*Hex*
Black	0000	00h	Gray	1000	08h
Blue	0001	01h	Light blue	1001	09h
Green	0010	02h	Light green	1010	0Ah
Cyan	0011	03h	Light cyan	1011	0Bh
Red	0100	04h	Light red	1100	0Ch
Magenta	0101	05h	Light magenta	1101	0Dh
Brown	0110	06h	Yellow	1110	0Eh
White	0111	07h	Bright white	1111	0Fh

Table 9.4. Background colors (high nibble).

Normal Background	*Binary*	*Hex Byte*
Black	0000	00h
Blue	0001	10h
Green	0010	20h
Cyan	0011	30h
Red	0100	40h
Magenta	0101	50h
Brown	0110	60h
Gray	0111	70h

To specify an attribute to one of the video calls, logically `OR` the foreground, background, and blink, if desired to form a single value. Look at the following coding example. (Hex values are shown for the attributes.)

```
#define BLACK     0x00
#define BLUE      1x07
#define GREEN     0x02
#define CYAN      0x03
#define RED       0x04
#define WHITE     0x07
#define GRAY      0x08
#define LTBLUE    0x09
#define LTGREEN   0x0A
#define LTCYAN    0x0B
#define LTRED     0x0C
```

```
#define BGBLACK  0x00
#define BGBLUE   1x70
#define BGGREEN  0x20
#define BGCYAN   0x30

dError = PutVidChars(0, 0,
                     "This is a test...", 17,
                     BGBLUE|WHITE);
```

Terminating Your Application

The call `ExitJob()` is provided to end the execution of your program. The `exit()` function in a high-level language library will call this function.

It is best to deallocate all system resources before calling `ExitJob()`. This includes any memory, exchanges, and other things allocated while you were running. You should also wait for responses to any requests you made or system calls.

MMURTL will clean up after you if you don't, but an application knows what it allocated, and from an overall system standpoint it is more efficient to do it yourself. MMURTL has to hunt for them. Remember, MMURTL is more like a referee on the football field. You will be terminated and sent to the "sidelines" if necessary.

Any task in your job may call `ExitJob()`. All tasks in your job will be shut down at that time and the application will be terminated.

Replacing Your Application

Your application has the option of "chaining" to another application. This is done by calling the `Chain()` function.

This terminates your application and loads another one that you specify. Some of the basic resources are saved for the new application, such as the job-control block and virtual video memory. You can pass information to the incoming application by using the `SetCmdLine()` function.

Another option to chaining is to tell the operating system what application to run when you exit. This is done with the `SetExitJob()` call. You can specify the name of a run file to execute when your job exits. If no file is specified, all the resources are recovered from the job and it is terminated.

10

Systems Programming

This chapter is aimed at the MMURTL systems programmer. It describes writing message-based system services and device drivers. Use of the associated operating system calls is described in a generic format that can be understood by C, Pascal, and assembly language programmers.

Systems programming deals with writing programs that work directly with the operating system and hardware or that provide services to application programs.

Systems programmers require a better understanding of the operating system. Application programmers shouldn't have to worry about hardware and operating-system internal operations to accomplish their tasks. They should have a basic understanding of multitasking and messaging and should be able to concentrate on the application itself.

If you want (or need) to build device drivers or system services, you should make sure you understand the material covered in the architecture and applications programming chapters. They contain theory and examples to help you understand accessing services, messaging, and memory management operations.

Writing Message-Based System Services

System services are installable programs or built-in operating system services that provide system-wide, message-based services for application programs, as well as for other services. MMURTL is a message-based operating system and is designed to support programs in a client/server environment. Many people associate the term *client/server* only with a separate computer system that is the server for client workstations. MMURTL takes this down to a single processor level. A program that performs a specific function or group of functions that are shared by two or more programs is a system service. This is the basis behind the `Request()` and `Respond()` messaging primitives. Client programs make a *request;* the service does the processing and *responds.* This type of messaging can be compared to remote procedure calls with a twist. The file system and keyboard services are examples of message-based system services.

Initializing Your System Service

The basic steps to initialize your service and start serving requests are listed here:

1. Initialize or allocate any resources required, such as:
   Additional memory, if required.
   Main service exchange (clients send requests here).
   Additional exchanges, if needed.
   Additional tasks, if required.
   Anything else you need to prepare for business (initialize variables, and so on).
2. Call `RegisterSVC()` with your name and main service exchange.
3. Wait for messages, service them, then respond.

A Simple System Service Example

Listing 10.1 is the world's simplest system service for the MMURTL operating system. Its purpose is to hand out unique numbers to each request that comes in. It's not very useful, but it's a good clean example showing the basic "guts" of a system service. The service name is `"NUMBERS "` (note that the name is space-padded). Service names *are* case-sensitive. The service will get a pointer in the request block (`pData1`) that tells it where the user wants the number returned.

Listing 10.1. A system service.

```
/* Super Simple System Service */

struct RqBlk {            /* 64 byte request block structure */
    long ServiceExch;
    long RespExch;
    long RqOwnerJob;
    long ServiceRoute;
    char *pRqHndlRet;
    long dData0;
    long dData1;
    long dData2;
    int  ServiceCode;
    char npSend;
    char npRecv;
    char *pData1;
    long cbData1;
    char *pData2;
    long cbData2;
    long RQBRsvd1;
    long RQBRsvd2;
    long RQBRsvd3;
    };

#define ErcOK         0
#define ErcBadSvcCode 32

struct pRqBlk *RqBlk;            /* A pointer to a Request Block */
unsigned long NextNumber = 0;    /* The number to return */
unsigned long MainExch;          /* Where we wait for Requests */
unsigned long Message[2];        /* The Message with the Request */

void main(void)
{
unsigned long OSError, ErrorToUser;

    OSError = AllocExch(&MainExch);       /* get an exchange */

  if (OSError) {                      /* look for a kernel error */
    exit(OSError);

    OSError = RegisterSvc("NUMBERS ", MainExch);

  if (OSError) {                      /* look for a system error */
```

continues

Listing 10.1. continued

```
    exit(OSError);

  while (1) {                       /* WHILE forever */

    OSError = WaitMsg(MainExch, Message); /* Exch & pointer */

    if (!OSError) {

        pRqBlk = Message[0];    /* First DWORD contains ptr to RqBlk */

    if (pRqBlk.ServiceCode == 0)      /* Abort request from OS */
        ErrorToUser = ErcOK;

    else if (pRqBlk.ServiceCode == 1) {  /* User Asking for Number */
        *pRqBlk.pData1 = NextNumber++;     /* Give them a number */
        ErrorToUser = ErcOK;               /* Respond with No error */
      }
      else
        ErrorToUser = ErcBadSvcCode;       /* Unknown Service code! */

      OSError = Respond(pRqBlk, ErrorToUser): /* Respond to Request */

    }
  }  /* Loop while(1) */
}
```

The Request Block

As you can see, the system service interface using `Request()` and `Respond()` is not complicated. As a system service, when you receive a request, `Wait` returns and the two-dword message is filled in. The first dword is the request block handle. This value is actually a pointer to the request block itself in unprotected operating system memory. The memory address of the request block and the aliased pointer to the user's memory are active and available only while the request block is being serviced. This gives the system service the opportunity to look at or use any parts of the request block it needs.

Several items in the request block are important to the service. The number of items needed depends on the complexity of the service you're writing.

Items in the Request Block

The request block contains certain items that the service needs to do its job. Programs and the operating system pass information to the service in various request block items. The complexity of your service, and how much information a program must pass to you, will be determined by how much information you need to carry out the request. As a system service writer, you define the data items you need, and it is your responsibility to document this information for application programmers.

The service must never attempt to write to the request block. In fact, installable services would find themselves terminated for the attempt. The request block is in operating-system memory that is *read-only* to user-level jobs. An installable service will operate at the protection and access user levels.

The Service Code

A single word (a 16-bit value) is used for the callers to indicate exactly what service they are requesting from you. This is called the service code. For example, in the file system, one number represents `OpenFile`, another `CloseFile`, and another `RenameFile`. In the simple example above, the service only defines one service code that it handles. This is the number 1 (one).

The operating system reserves service code 0 (zero) for itself in all services. You may not use or define service code 0 to callers in your documentation. When a service receives a request with service code 0, the operating system is telling you that a job (a user program or another service) either has exited or has been shut down by the operating system. How you handle this depends on how complex your service is, and whether or not you hold requests from multiple callers before responding.

For all other values of the service code, you must describe in your documentation what request information you expect from the caller and what service you provide. If you receive a service code you don't handle or haven't defined, you should return the error message `Bad Service Code`, which is defined in standard MMURTL header files.

Caller Information in a Request

Three dwords are defined to allow a program to pass information to the service you require to do your job.

These items are `dData0`, `dData1`, and `dData2`. They are 32-bit values, and whether or not they are signed or unsigned is up to you (the system service writer). They are one-way data values from the caller to you. They may not contain pointers to the caller's memory area, because the operating system does not alias them as such. A prime example of their use is in the file system service, where callers pass in a file handle using these data items.

For larger data movement to the service, or *any* data movement back to the caller, the request block items `pData1` and `pData2` must be used. These two request block items are specifically designed as pointers to a data area in the caller's memory space. The size of the area is defined by the accompanying 32-bit values, `cbData1` and `cbData2`. The operating system is "aware" of this, and it will alias the pointers so the caller's memory space is accessible to the service. As a service, you must not attempt to access any data in the caller's space other than what is defined by `pData1`, `cbData1` and `pData2`, `cbData2`. Also, be aware that these pointers are valid only from the time you receive the request and until you respond.

Asynchronous Services

Most services will be *synchronous.* Synchronous services wait, receive a request, and then respond before waiting for the next request. All operations are handled in a serial fashion. This is the easiest way to handle requests.

A service may receive and hold more than one request before responding. This type of service is defined as *asynchronous.* This type of service may even respond to the second request it receives before responding to the first.

This type of service requires one or more additional exchanges to hold the outstanding requests. The keyboard service is a good example of an asynchronous service. In the keyboard service, callers can make a request to be notified of global key codes. These key codes may not occur for long periods of time, and many normal keystrokes may occur in the meantime. This means the keyboard service must place the global key code requests on another exchange while it handles normal requests.

The kernel primitive `MoveRequest()` takes a request you have received using `Wait`, and moves it to another exchange. You can then use `WaitMsg` or `CheckMsg` to get the requests from this second hold exchange when you know you can respond to them.

While you are holding requests asynchronously, you may receive an abort service code from the operating system (service code 0). If this happens, you need to ensure that you look at all the requests you are holding to see if this abort is for one of those requests. If so, you must respond to the held request identified by the abort immediately, before responding to any other requests you are holding. This is so that the operating system can reclaim the request block, which is a valuable system resource. The abort request contains the job number of the aborted program in `dData0`. You would look at each request block in the `RqOwnerJob` field to see if this request came from it. If so, you must respond without accessing its memory or acting on its request.

System Service Error Handling

When the service responds, the error (or status) to the user is passed back as the second dword parameter in the respond call. The first is the request block handle.

When you write system services, you must be aware of conventions used in error passing and handling. Some operating systems simply give you a true or false style of response as to whether or not the function you requested was carried out without errors.

MMURTL uses this concept but carries it one step further. You return a 0 (zero) if all went well. If an error occurred or if you need to pass some kind of status back to the caller, you return a number that indicates what you want to tell them. An example of this is how the file system handles errors. `OpenFile()` returns a 0 error code if everything went fine. However, it

may return five or six different values for various problems that can occur during the `OpenFile()` process. The file may already be opened in an incompatible mode by someone else, the file may not exist, or maybe the file system couldn't allocate an operating system resource needed to open your file. In each of these cases, you would receive a non-zero number indicating the error. You don't have to access a global in your program or make another call to determine what went wrong.

Writing Device Drivers

Device drivers control or emulate hardware on the system. Disk drives, communications ports (serial and parallel), tape drives, RAM disks, and network frame handlers are all examples of devices (or pseudo devices) that require device drivers to control them. MMURTL has standardized entry points for *all* devices on the system.

MMURTL is built in layers. The device drivers are the closest layer to the operating system. You should not confuse device drivers with message-based system services. Device drivers are accessed via call gates transparent to the applications programmer that allow callers to easily and rapidly access code with a procedural interface from high-level languages (C and Pascal for example), or by simply "pushing" parameters on the stack in the assembler and making a *call*.

MMURTL provides several built-in device drivers including floppy disk, hard disk, basic video, keyboard, and communication (serial and parallel).

Devices in MMURTL are classified as two different types: *random-* and *sequential*-oriented. Random-oriented devices are things such as disk drives and RAM disks that require the caller to tell the driver how much data to read or write, as well as where the data is on the device. Network frame handlers may also be random because a link layer address is usually associated with the read and write functions, although they may be sequential too. Some types of tape drives allow reading and writing to addresses (sectors) on the tape and therefore may also be random devices. Sequential devices are things like communications ports, keyboard, and sequential video access (ANSI drivers, and so on). Most byte-oriented devices tend to be sequential devices because they don't have a fixed-size block and are not randomly accessed.

The MMURTL standard device driver interface can handle *both* sequential and random devices with fixed or variable block and/or sector lengths. The interface is non-specific.

All device drivers use a device control block (DCB). The device driver defines and keeps the DCB in its memory space, which actually becomes the operating system memory space when you load it. When a device driver initializes itself with the call `InitDevDr()`, it passes in a pointer to its DCB. If a device driver controls more than one device, it will pass in a pointer to an array of DCBs. Things like `Device Name`, `Block Size`, and `Device Type` are examples of the fields in a DCB. Each device driver must also maintain in its data area all hardware specific data (variables) needed to control and keep track of the state of the device.

Device Driver Theory

MMURTL is a protected-mode, paged memory operating system that uses two of the four possible protection levels of the 386/486 processors. The levels are 0 and 3 and are referred to as System (0) and User (3). I have made it rather painless to build fast, powerful, device drivers and have even gone into substantial detail for beginners.

Device drivers must have access to the hardware they control. Device drivers have complete access to processor I/O ports for their devices without operating system intervention. Only system code (protection level 0) is allowed processor port I/O access. This means message- or request-based system services and application code (both at user level 3) *cannot* access ports at all; otherwise, a processor protection violation will occur. For example, the file system is at user level 3. It does not directly access hardware. The device-driver builder must not access ports other than those required to control its hardware.

Device drivers must not attempt to directly control the system DMA, timer hardware, interrupt vectors, PICUs, or any operating system structures. MMURTL provides a complete array of calls to handle all the hardware, which ensures complete harmony in MMURTL's real-time, multitasking, multithreaded environment, while providing a fair amount of hardware independence. MMURTL takes care of synchronization of system-level hardware access so device-driver programmers don't have to worry about it. Those who have written device drivers on other systems will appreciate this concept.

Building Device Drivers

When the device driver is installed by the loader, or included in the operating system code at build time, it has to make a system call to set itself up as a device driver in the system. Additional system calls may have to be made to allocate or initialize other system resources required before it can handle its device.

Device drivers can be doing anything from simple disk device emulation (RAM disk, for example), to handling complex devices such as hard drives, SCSI ports, or network frame handlers (network link layer devices).

The call to set up as a device driver is `InitDevDr()`. With `InitDevDr()`, you provided a pointer to your device control blocks (DCB) after you filled in the required information. The DCB is shared by both the device driver and operating system code.

Before calling `InitDevDr`, you will most likely have to allocate other system resources and possibly even set up hardware interrupts that you will handle from your device. Additional system resources available to device drivers are discussed later in this chapter.

The device driver looks, and is programmed, like any application in MMURTL. It has a main entry point you specify when writing the program. In C this would be `main()`; in Pascal it's the main program block. In assembler, this would be the `START` entry point specified for the

program. This code is immediately executed after loading the driver, just like any other program. This code will execute only once to allow you to initialize the driver. The main entry point will never be called again. This will generally be a very small section of code (approximately 30 lines).

How Callers Reach Your Driver

All device drivers are accessed via one of three public calls defined in the operating system. You will have functions in your driver to handle each of these calls. When you fill in the DCB, you will fill in the addresses of the three entry points the operating system will call on behalf of the original caller.

Your driver program must interface properly with the three public calls. You can name them anything you want in your program, but they must have the same number and type of parameters (arguments) as the `PUBLIC` calls defined in the operating system for all device drivers. The operating system defines the three following `PUBLIC` calls:

```
DeviceInit()
DeviceOp()
DeviceStat()
```

The parameter listings and function expectations are discussed in detail later in this chapter. Your functions may not have to do anything when called depending on the device you are controlling. They *must* at least accept the calls and return a status code of 0 (for no error) if they ignore it. When programs call one of these `PUBLIC` calls, MMURTL does some housekeeping and then calls your specified function.

Device Driver Setup and Installation

The initialization section of your device driver must make certain calls to set up and install as a driver. The generic steps are described here (your actual calls may vary depending on what system resources you need):

1. Initialize or allocate any resources required.

   Allocate any additional memory, if required.

   Allocate exchanges, if needed for messaging from a second task or the ISRs (see the floppy device driver for an example of a device driver with a separate task to handle hardware interrupts).

   Set up interrupt service routines, if needed.

   Check or set up the devices, if needed.

   Do anything else you need to before being called.
2. Enter the required data in the DCBs to pass to the `InitDevDr()` call.

3. Enable (`UnMaskIRQ()`) any hardware interrupts you are servicing, unless this is done in one of the procedures (such as `DeviceInit()`) as a normal function of your driver.
4. Call `InitDevDr()`.

`InitDevDr` never returns to you. It terminates your task and leaves your code resident to be called by the operating system when one of the three device-driver calls is made.

At this point, you have three device-driver routines that are ready to be called by the operating system. It will be the operating system, not the users of the devices, that calls them. This is because the operating system has a very small layer of code that provides common entry points to the three device-driver calls for *all* drivers. This layer of code performs several functions for you. One very important function is blocking for non–re-entrant device drivers. In a multitasking environment it is likely that a device-driver function could be called while someone else's code is already executing it, especially if the driver controls two devices. One of the items you fill out in the DCB is a flag (`fDevReent`) to tell the operating system whether your driver is re-entrant. Most device drivers are not re-entrant, so this will normally be set to FALSE (0).

System Resources for Device Drivers

Because they control the hardware, device drivers are very close to the hardware. They require special operating system resources to accomplish their missions. Many device drivers need to use DMA and timer facilities, and they must service interrupts from the devices they control. MMURTL has a complete array of support calls for such drivers.

Interrupts

Several calls are provided to handle initializing and servicing hardware interrupt service routines (ISRs) in your code. See the section that deals specifically with ISRs for complete descriptions and examples of simple ISRs. The floppy and hard-disk device drivers also provide good ISR examples. ISRs should be written in assembly language for speed and complete code control, although it's not a requirement. The following is a list with brief descriptions of the calls you may need to set up and use in your ISR:

`SetIRQVector()` is how you tell the operating system what function to call when your hardware interrupt occurs. This is often called an interrupt vector.

`EndOfIRQ()` sends an "End of Interrupt" signal to the programmable interrupt controller units (PICU), and should be called as the last thing the ISR does. In other operating systems, specifically DOS, the ISR usually sends the EOI sequence directly to the PICU. Do not do this in MMURTL, as other code may be required for execution in later versions of MMURTL, which will render your device driver obsolete.

`MaskIRQ()` and **`UnMaskIRQ()`** are used to turn on and off your hardware interrupt by masking and unmasking it directly with the PICU. More detail about each of these calls can be found in the alphabetical listing of operating system calls in Chapter 15, "API Specification".

Direct Memory Access Device (DMA)

If your device uses DMA, you will have to use the `DMASetUp()` call each time before instructing your device to make the programmed DMA transfer. The `DMASetUp()` call lets you set a DMA channel for the type, mode, direction, and size of a DMA move. DMA also has some quirks you should be aware of, such as its inability to cross-segment physical boundaries (64K). Your driver should use the `AllocDMAPage()` call to allocate memory that is guaranteed not to cross a physical segment boundary and also provide you with the physical address to pass to `DMASetUp()`. It is important to understand that the memory addresses your program uses are *linear* addresses and do not equal physical addresses. MMURTL uses paged memory, and the addresses you use will probably never be equal to the physical address! Another thing to consider is that DMA on the ISA platforms is limited to a 16MB physical address range. Program code can, and will most likely, be loaded into memory above the 16MB limit if you have that much memory in your machine. This means all DMA transfers must be buffered into memory that was allocated using `AllocDMAPage()`. See the operating system public call descriptions for more information on `DMASetUp()` and how to use `AllocDMAPage()`. Many users of DMA also need to be able to query the DMA count register to see if the DMA transfer was complete, or how much was transferred. This is accomplished with `GetDMACount()`.

Timer Facilities

Quite often, device drivers must delay (sleep) while waiting for an action to occur or have the capability to "time-out" if an expected action doesn't occur in a specific amount of time. Two calls provide these functions. The `Sleep()` call actually puts the calling process to sleep (makes it wait) for a specific number of 10-millisecond periods. The `Alarm()` call will send a message to a specified exchange after a number of caller-specified 10-millisecond increments. The `Alarm()` call can be used asynchronously to allow the device code to continue doing something while the alarm is counting down. If you use `Alarm()`, you will most likely need to use `KillAlarm()`, which stops the alarm function if it hasn't already fired off a message to you. See the call descriptions for more information on the `Sleep()`, `Alarm()`, and `KillAlarm()` functions.

Message Facilities for Device Drivers

Interrupt service routines (if you need them in your driver) sometimes require special messaging features that application programs don't need. A special call that may be required by some device drivers is `ISendMsg()`. `ISendMsg()` is a special form of `SendMsg()` that can be used inside an ISR (with interrupts disabled). It doesn't force a task switch even if the destination exchange has a higher priority process waiting there. `ISendMsg()` also does *not* re-enable interrupts before returning to the ISR that called it, so it may be called from an ISR while interrupts are disabled. `ISendMsg()` is used if an ISR must send more than one message from the interrupted condition. If you must send a message that guarantees a task switch to a higher priority task, or you only need to send one message from the ISR, you can use `SendMsg()`. If you use `SendMsg()`, it must be the *final* call in your ISR after the `EndOfIRQ()` call has been made and just before the `IRETD` (return from interrupt). This will guarantee a task switch if the message is to a higher priority task.

Detailed Device Interface Specification

The following sections describe the device control block (DCB), each of the three PUBLIC device calls, and how to implement them in your code.

Device Control Block Setup and Use

The DCB structure is shown in the following table with sizes and offsets specified. How the structure is implemented in each programming language is different, but easy to figure out. The length of each field in the DCB is shown in bytes, and all values are *unsigned*. You can look at some of the included device driver code with MMURTL to help you.

The size of a device control block is 64 bytes. You must ensure there is no padding between the variables if you use a record in Pascal, a structure in C, or assembler to define the DCB. If your driver controls more than one device, you will need to have a DCB for each one it controls. Multiple DCBs must be contiguous in memory. Table 10.1 presents the fields that define a DCB.

Table 10.1. Device control block definition.

Name	*Size*	*Offset*	*Description*
DevName	12	0	Device name, left justified
cbDevName	1	12	Length of device name in bytes
DevType	1	13	1 = RANDOM device, 2 = SEQUENTIAL
nBPB	2	14	Bytes Per Block (1 to 65535 max.) (0 for variable block size)
dLastDevErc	4	16	Last error code from an operation
ndDevBlocks	4	20	Number of blocks in device (0 for sequential)
pDevOp	4	24	Pointer to device operation handler
pDevInit	4	28	Pointer to device Init handler
pDevStat	4	32	Pointer to device status handler
fDevReent	1	36	Is device handler re-entrant?
fSingleUser	1	37	Is device usable from 1 job only?
wJob	2	38	If fSingleUser true, this is job
OSUseONLY1	4	40	Allocate and 0, but do not modify
OSUseONLY2	4	44	Allocate and 0, but do not modify
OSuseONLY3	4	48	Allocate and 0, but do not modify
OSuseONLY4	4	52	Allocate and 0, but do not modify
OSuseONLY5	4	56	Allocate and 0, but do not modify
OSuseONLY6	4	60	Allocate and 0, but do not modify

The fields of the DCB must be filled in with the initial values for the device you control before calling `InitDevDr()`. Most of the fields are explained satisfactorily in the descriptions next to them, but the following fields require detailed explanations:

`DevName` Each driver names the device it serves. Some names are standardized in MMURTL, but this doesn't prevent the driver from assigning a non-standard name. The name is used so that callers don't need to know a device's number in order to use it. For instance, floppy disk drives are usually named FD0 and FD1, but a driver for non-standard floppy devices can name them anything (up to 12 characters). Each device name must be unique.

`cbDevName` A single byte containing the number of bytes in the device name.

`DevType` This indicates whether the device is addressable. If data can be reached at a specific numbered address as specified in the `DeviceOP()` call parameter `dLBA` (logical block address), then it's considered a *random* device and this byte should contain a 1. If the device is sequential, and the concept of an address for the data has no meaning (such as with a communications port), then this should contain a 2.

`nBPB` The number of bytes per block. Disk drives are divided into sectors. Each sector is considered a block. If the sectors are 512-byte sectors (MMURTL standard), then this would be 512. If it is another device, or it's a disk drive that is initialized for 1024-byte sectors, then this would reflect the true block size of the device. If it is a single-byte oriented device such as a communications port or sequential video, then this would be 1. If the device has variable length blocks, then this should be 0.

`dLastDevErc` When a driver encounters an error, usually an unexpected one, it should set this variable with the last error code it received. The driver does not need to reset this value as MMURTL resets it to zero on entry to each call to `DeviceOP` for each device.

`ndDevBlocks` This is the number of addressable blocks in your device. For example, on a floppy disk this would be the number of sectors × number of cylinders × number of heads on the drive. MMURTL expects device drivers to organize access to each *random* device by 0 to *n* logical addresses. An example of how this is accomplished is with the floppy device driver. It numbers all sectors on the disk from 0 to `nTotalSectors` (`nTotalSectors` is `nHeads * nSecPerTrack * nTracks`). The head number takes precedence over track number when calculating the actual position on the disk. In other words, if you are on track zero, sector zero, head zero, and you read one sector past the end of that track, we then move to the next head, *not* the next track. This reduces disk arm movement, which increases throughput. The hard disk driver operates the same way. Addressing with 32 bits limits address access to 4GB on a single-byte device and 4GB × sector size on disk devices, but this is not much of a limitation. If the device is mountable, or allows multiple media sizes, this value should be set initially to allow access to any media for identification; this is device dependent.

`pDevOp`, `pDevInit`, and `pDevStat` These are pointers to the calls in your driver for the following `PUBLIC` device functions:

`DeviceOp` — Performs I/O operations in device

`DeviceInit` — Initializes or resets the device

`DeviceStats` — Returns device-specific status

MMURTL uses indirect 32-bit near calls to your driver's code for these three calls. The return instruction you use should be a 32-bit near return. This is usually dictated by your compiler or assembler and how you define the function or procedure. In C-32 (MMURTL's standard C compiler), defining the function with no additional attributes or modifiers defaults it to a 32-bit near return. In assembler, `RETN` is the proper instruction.

`fDevReent` This is a 1-byte flag (0 false, non-zero true) that tells MMURTL if the device driver is re-entrant. MMURTL handles conflicts that occur when more than one task attempts to access a device. Most device drivers have a single set of variables that keep track of the current state of the device and the current operation. In a true multitasking environment more than one task can call a device driver. For devices that are *not* re-entrant, MMURTL will *queue* up tasks that call a device driver currently in use. The queuing is accomplished with an exchange using the `SendMsg()` and `WaitMsg()` primitives. A portion of the reserved DCB is used for this purpose. On very complicated devices such as a SCSI driver, where several devices are controlled though one access point, the driver may have to tell MMURTL it's re-entrant, and handle device conflicts internally.

`fSingleUser` If a device can only be assigned and used by one job (user) at a time, this flag should be true (non-zero). This applies to devices like communications ports that are assigned and used for a session.

`wJob` If the `fSingleUser` is true, this will be the job that is currently assigned to the device. If this is zero, no job is currently assigned to the device. If `fSingleUser` is false, this field is ignored.

`OSUseONLY1,2,3,4,5,6` These are reserved for operating system use and device drivers should not alter the values in them after the call to `InitDevDr()`. They must be set to zero before the call to `InitDevDr()`.

Standard Device Call Definitions

As described previously, all calls to device drivers are accessed through three predefined `PUBLIC` calls in MMURTL. They are these:

```
DeviceOp(dDevice, dOpNum, dLBA, ndBlocks, pData):dError

DeviceInit(dDevice, pInitData, sdInitdata):dError

DeviceStat(dDevice, pStatRet, sdStatRetmax):dError
```

Detailed information follows for each of these calls. The procedural interfaces are shown with additional information such as the *call frame offset*, where the parameters will be found on the stack after the function has set up its stack frame using Intel standard stack frame entry techniques. Note that your function should also remove these parameters from the stack. All functions in MMURTL that return errors use the EAX register. Your function implementations must do the same. Returning zero indicates successful completion of the function. A non-zero value indicates an error or status that the caller should act on. Standard device errors should be used when they describe well the error or status you wish to convey to the caller. Device-specific error codes should be included with the documentation for the driver.

DeviceOp Function Implementation

The `DeviceOp()` function is used by services and programs to carry out normal operations such as Read and Write on a device. The call is not device specific and allows any device to be interfaced through it. The `dOpNum` parameter tells the driver which operation is to be performed. An almost unlimited number of operations can be specified for the device operation call (2^{32}). The first 256 operations (dOp number) are pre-defined or reserved. They equate to standard device operations such as read, write, verify, and format. The rest of the dOp Numbers may be implemented for any device-specific operations as long as they conform to the call parameters described here.

```
dError = DeviceOp(dDevice,dOpNum,dLBA,dnBlocks,pData);
```

The call frame offsets for assembly language programmers are these:

```
dDevice  [EBP+24]
   dOpNum   [EBP+20]
   dLBA     [EBP+16]
   dnBlocks [EBP+12]
   pData    [EBP+08]
```

Parameter descriptions:

`dDevice` The device number

`dOpNum` Identifies which operation to perform

0 Null operation

1 Read (receive data from the device)

2 Write (send data to the device)

3 Verify (compare data on the device)

4 Format Block (tape or disk devices)

5 Format Track (disk devices only)

6 Seek Block (tape or disk devices only)

	7 Seek Track (disk devices only)
	(Communications devices)
	10 OpenDevice (communications devices)
	11 CloseDevice (communications devices)
	(RS-232 devices with explicit modem control)
	15 SetDTR
	16 SetCTS
	Undefined operation numbers below 255 are *reserved*
	256-n driver defined (driver specific)
`dLBA`	Logical block address for I/O operation.
	For sequential devices, this parameter will be ignored.
`dnBlocks`	Number of *contiguous* blocks for the operation specified. For sequential devices, this will simply be the number of bytes.
`pData`	Pointer to data (or buffer for reads) for specified operation

DeviceStat Function Implementation

The `DeviceStat()` function provides a way for device-specific status to be returned to a caller if needed. Not all devices will return status on demand. In cases where the function doesn't, or can't, return status, you should return 0 to `pdStatusRet` and return the standard device error `ErcNoStatus`.

```
dError = DeviceStat(dDevice,pStatRet,dStatusMax,pdStatusRet);
```

The call frame offsets for assembly language programmers are these:

```
dDevice    [EBP+20]
pStatusRet [EBP+16]
dStatusMax [EBP+12]
pdStatusRet[EBP+08]
```

Parameter descriptions:

`dDevice`	Device number to status
`pStatBuf`	Pointer to buffer where status will be returned
`dStatusMax`	Caller sets this to tell you the maximum size of status to return in bytes
`pdStatusRet`	Pointer to dword where you return actual size of status returned in bytes

DeviceInit Function Implementation

Some devices may require a call to initialize them before use or to reset them after a catastrophe. An example of initialization would be a communications port for baud rate, parity, and so on. The size of the initializing data and its contents are device specific and should be defined with the documentation for the specific device driver.

```
dError = DeviceInit(dDevice,pInitData,dInitData);
```

The call frame offsets for assembly language programmers are these:

```
dDevice   [EBP+16]
pInitData [EBP+12]
dInitData [EBP+08]
```

Parameter descriptions:

`dDevice`	Dword indicating device number
`pInitData`	Pointer to device specific data for initialization to be returned
`dInitData`	Dword indicating maximum size of status to return in bytes

Initializing Your Driver

`InitDevDr()` is called from a device driver after it is first loaded to let the operating system integrate it into the system. After the device driver has been loaded, it should allocate *all* system resources it needs to operate and control its devices while providing service through the three standard entry points. A 64-byte DCB must be filled out for each device the driver controls before this call is made. When a driver controls more than one device, it must provide the device control blocks for each device. The DBCs must be contiguous in memory. If the driver is flagged as not re-entrant, then all devices controlled by the driver will be locked out when the driver is busy. This is because one controller, such as a disk or SCSI controller, usually handles multiple devices through a single set of hardware ports, and one DMA channel if applicable, and can't handle more than one active transfer at a time. If this is not the case, and the driver can handle two devices simultaneously, the driver should be broken into two separate drivers.

The definition and parameters to `InitDevDr()` are as follows:

```
InitDevDr(dDevNum, pDCBs, nDevices, fReplace)
```

`dDevNum` This is the device number that the driver is controlling. If the driver controls more than one device, this is the first number of the devices. This means the devices are numbered consecutively.

`pDCBs` This is a pointer to the DCB for the device. If more than one device is controlled, this is the pointer to the first in an array of DCBs for the devices. This means the second DCB must be located at `pDCBs` + 64, the third at `pDCBs` + 128, and so on.

`nDevices` This is the number of devices that the driver controls. It *must* equal the number of contiguous DCBs that the driver has filled out before the `InitDevDr` call is made.

`fReplace` If true, the driver will be substituted for the existing driver functions already in place. This does not mean that the existing driver will be replaced in memory, it only means the new driver will be called when the device is accessed. A replacement driver *must* specify at least as many devices as the original driver handled.

Operating System Functions for Device Drivers

The following is a list of functions that were specifically designed for device drivers. They perform many of the tedious functions that would otherwise require assembly language, and a very good knowledge of the hardware. Please use the MMURTL API reference for a detailed description of their use.

`AllocDMAMem` Allocates memory that will be compatible with DMA operations. It also returns the physical address of the memory which is required for the DMASetUp call.

`EndOfIRQ` Resets the programmable interrupt controller unit (PICU) at the end of the ISR sequence.

`MaskIRQ` Masks one interrupt (prevents it from interrupting) by programming the PICU.

`SetIRQVector` Sets up a vector to your ISR.

`DMASetUp` Programs the DMA hardware channel specified to move data to or from your device.

`UnMaskIRQ` Allows interrupts to occur from a specified channel on the PICU.

Standard Device Error Codes

The MMURTL device-driver interface code will return errors for certain conditions such as a device that's not installed being called. Your device drive will also return error codes for problems it has honoring the device call. If an error code already defined in MMURTL's standard header file adequately describes the error or status, then use it.

11

The MMURTL Monitor Program

The Monitor program is included as part of the operating system.

Many of the functions performed by the monitor are for testing the system. The monitor also serves as a context display controller by monitoring for global keys that indicate the user's desire to change, or terminate, the job he or she is currently viewing. When the user presses these global keys, the keyboard, as well as the video display, is assigned to the new job or is terminated.

In future versions of MMURTL, many of the functions now performed by the monitor will be moved to external programs. If you intend to write an operating system, you will find a need for something similar to the monitor as an initial piece of code for testing built into the operating system.

Active Job (Video and Keyboard)

When multiple jobs are running in MMURTL, only one job at a time can be displayed or accept keyboard input.

The Monitor program enables you to select the job with which you are currently interacting. You select a job by pressing the Ctrl-Alt-PageDown keys to move forward through the active jobs until you see the one you want. The current job may also be terminated by pressing Ctrl-Alt-Delete.

Even when a job is not being displayed, it is still running unless it's waiting for something such as keyboard input or some other form of communication. If it's displaying video data, it continues to run because a job never actually knows if it is displaying data to the real screen or to its own virtual screen, a buffer in memory.

Initial Jobs

After all internal device drivers and services are loaded, the monitor attempts to open a text file, called Initial.Job, in the system directory.

This file contains the filenames of one or more run files you want to execute automatically on boot up. The format is very simple. Each line lists the full file specification of a RUN file. The name must start in the first column of a line. No spaces, tabs or other characters are allowed before the RUN filename. A line that begins with a semicolon is ignored and considered a comment line.

Listing 11.1 shows a sample Initial.Job file.

Listing 11.1. Sample Initial.Job file.

```
;INITIAL.JOB - This is the initial jobs file.
;You may list the jobs that you wanted executed upon
```

```
;system boot in this file. One run file name per line,
;no spaces in front of the name and a proper end-of-line
;after each entry. Any spaces, tabs or comments after
;the run file name are ignored. No parameters are passed
;to the run file. The file name must contain the FULL
;file name including path. (e.g., Drive:\DIR\NAME.RUN)
;Comment lines begin with a semicolon
;Maximum line length is 80 characters total.
;
C:\MSamples\Service\Service.run
C:\MMSYS\CLI.RUN    <---- this will be loaded on bootup
;End of file
```

If the executable job listed is CLI.RUN, the video and keyboard will be assigned to this job and taken away from the monitor.

Monitor Function Keys

The monitor also provides other functions such as system resource display, job display, and access to the built-in debugger. The functions are provided by function keys that are labeled across the bottom of the display. Table 11.1 describes the function of each assigned key.

Table 11.1. Monitor function keys.

Key	*Label*	*Function*
F1	LDCLI	Load a command line interpreter
F2	JOBS	List jobs
F3	STATS	Show system resource status
F8	BOOT	Reboots the system (hard reset)
F10	DEBUG	Enter the debugger

Monitor Program Theory

After all internal static and dynamic structures are initialized, the monitor is reached with a JMP instruction.

The first thing accomplished by the monitor is the initialization of all internal device drivers and system services. During this process, any errors returned are displayed. A status code of zero indicates successful initialization.

The device drivers that are initialized include the hard disk, floppy disk, RS-232 serial communications port, and the parallel port (LPT) driver.

The initialized services include the keyboard and the file system.

The initialization also includes starting two tasks inside the monitor code.

The first additional task provides the context-switching capabilities by leaving a global key request with the keyboard service. It looks for the Ctrl-Alt-PageDown key combination, and shifts the keyboard and video to the next job in numerical order. This task also looks for the Ctrl-Alt-Delete key combination which terminates a job.

The second task is used to recover resources when a job ends or is terminated for memory access or other protection violations.

After the initialization is complete, user interaction is provided via the function keys discussed in Table 11.1. The monitor echoes other keys to the screen to show you it is working properly. They are displayed as typed.

Performance Monitoring

Performance monitoring is provided by sampling statistic-gathering public variables the operating system maintains and updates.

The statistics provide a snapshot of system resources every half second. These statistics include:

Free 4K memory pages—This is the total number of 4K memory pages available for allocation by the memory-management code.

Task switches total—This is the total count of task switches since boot up.

Preemptive task switches—This is the total number of times a task was preempted for a higher priority task.

CPU idle ticks (no work)—This is the number of times the operating system had nothing to do. There were no tasks ready to run. This number does not directly relate the timer ticks because all tasks can become idle several times between the time interrupt interval.

Tasks Ready to Run—This is the number of tasks currently queued to run. If this number is greater than zero, you will see the preemptive task switch count increase.

Free Task State Segments—These are task-management structures. The memory they occupy is dynamically allocated, but the count is fixed at OS build time. Each new task uses one of these.

Free Job Control Blocks—This is the number of free job control blocks left. Job control blocks are also dynamically allocated with a fixed count at OS build time. Each job uses one.

Free Request Blocks—Each time a request is made, a request block is allocated. When the response is made by the service, the request block is returned for re-use. These structures are also in dynamically allocated memory, but have a fixed count at OS build time.

Free link blocks—The link block is a small structure used by all messages sent and received, including requests. The number may seem a little high, but dozens of messages may be sent between tasks, even in the same job, that don't get picked up right away. Free link blocks are static and the count is determined at OS build time.

Free exchanges—This is the number of free exchanges that can be allocated. Exchanges are also dynamically allocated with a fixed count at OS build time.

Monitor Source Listing

Text-formatting functions are provided with the `xprintf` function defined in the monitor. I included no C library code in the operating system itself. In future versions, when a lot of the monitor functionality is moved out of the operating system, the C library code can be used. See Listing 11.2.

Listing 11.2. Monitor program source listing.

```
#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char

#include "MKernel.h"
#include "MMemory.h"
#include "MData.h"
#include "MTimer.h"
#include "MVid.h"
#include "MKbd.h"
#include "MJob.h"
#include "MFiles.h"
#include "MDevDrv.h"


#define ok 0
#define ErcNoDevice 504

static char rgStatLine[] =
"mm/dd/yy  00:00:00              MMURTL Monitor                    Tick:0          ";

static char rgMonMenu1[] = "LdCLI\xb3Jobs  \xb3Stats \xb3      ";
static char rgMonMenu2[] = "     \xb3      \xb3      \xb3Reboot";
static char rgMonMenu3[] = "     \xb3Debug \xb3      \xb3     ";

static char rgCPR1[]
="MMURTL (tm) - Message based, MUltitasking, Real-Time kerneL";
static char rgCPR2[]
="Copyright (c) R.A. Burgess, 1990-1995, All Rights Reserved";

static char *CRLF = "\r\n\r\n";

static unsigned long Color = WHITE¦BGBLACK;     /* Color test for xprintf */
```

continues

Listing 11.2. continued

```
static long time, date, tick;

unsigned long KillExch;     /* Messaging for stat task KILL proc */

static unsigned long KillMsg[2];    /* First DWORD = TSSExch, second is ERROR */
static unsigned long KillError;
static unsigned long KillJobNum;
static unsigned char fKilled;

static unsigned long MngrExch;      /* Messaging for stat task */
static unsigned long MngrMsg[2];
static unsigned long MngrHndl;
static unsigned long gcode;

static unsigned long GPExch;        /* Messaging for main */
static unsigned long GPMsg[2];
static unsigned long GPHndl;

static unsigned long GP1Exch;       /* Extra Messaging for main */
static unsigned long GP1Msg[2];
static unsigned long GP1Hndl;

/* Structure for disk device driver status and setup */

static struct diskstattype {
  U32 erc;
  U32 blocks_done;
  U32 BlocksMax;
  U8 fNewMedia;
  U8 type_now;      /* current disk type for drive selected */
  U8 resvd1[2];     /* padding for DWord align  */
  U32 nCyl;         /* total physical cylinders */
  U32 nHead;        /* total heads on device    */
  U32 nSectors;     /* Sectors per track        */
  U32 nBPS;         /* Number of bytes per sect */

  U32 LastRecalErc0;
  U32 LastSeekErc0;
  U8  LastStatByte0;
  U8  LastErcByte0;
  U8  fIntOnReset;  /* Interrupt was received on HDC_RESET */
  U8  filler0;
  U32 LastRecalErc1;
  U32 LastSeekErc1;
  U8  LastStatByte1;
  U8  LastErcByte1;
  U8  ResetStatByte;    /* Status Byte immediately after RESET */
  U8  filler1;
  U32 resvd1[2];    /* out to 64 bytes */
  };

static struct diskstattype DiskStatus;

#define nMaxJCBs 34     /* 32 dynamic plus 2 static */
static struct JCBRec *pJCB;
```

```
static long StatStack[256]; /* 1024 byte stack for Stat task */

static long MngrStack[256]; /* 1024 byte stack for Mngr task */

static unsigned char Buffer[512];
static unsigned long nMemPages;

extern unsigned long oMemMax;
extern unsigned long nSwitches;
extern unsigned long nSlices;
extern unsigned long nHalts;
extern unsigned long nReady;
extern unsigned long nRQBLeft;
extern unsigned long nJCBLeft;
extern unsigned long nTSSLeft;
extern unsigned long nLBLeft;
extern unsigned long nEXCHLeft;
extern unsigned long BootDrive;

/*============ protos (NEAR MMURTL support calls) =================*/

extern long InitKBDService(void);   /* From Keyboard.asm */
extern long fdisk_setup(void);      /* From Floppy.c */
extern long hdisk_setup(void);      /* From HardIDE.c */
extern long coms_setup(void);       /* From RS232.c */
extern long lpt_setup(void);        /* From Parallel.c */
extern long InitFS(void);           /* From Fsys.c */

extern long GetExchOwner(long Exch, char *pJCBRet);
extern long DeAllocJCB(long *pdJobNumRet, char *ppJCBRet);

/*================== START OF CODE ====================*/

/*************************************************************
 Formatted output routines for montitor program
 xprintf, xsprintf.
*************************************************************/

#include <stdarg.h>

#define S_SIZE  100

/*****************************************
* Determine if a character is a numeric digit
*****************************************/

static long isdigit(long chr)
{
;
#asm
    MOV EAX,[EBP+8]
    CMP AL, 30h     ;0
    JL isdigit0     ;No
    CMP AL, 39h     ;
    JLE isdigit1    ;Yes
isdigit0:
    XOR EAX,EAX     ;No
    JMP SHORT isdigit2
```

continues

Listing 11.2. continued

```
isdigit1:
    MOV EAX, -1
isdigit2:

#endasm
}

static long strlen(char *cs)
{
;
#asm
    XOR EAX, EAX
    MOV ESI,[EBP+8]
_strlen0:
    CMP BYTE PTR [ESI],0
    JE _strlen1
    INC ESI
    INC EAX
    JMP SHORT _strlen0
_strlen1:
#endasm
}

/*************************************************************
 This does the actual parsing of the format and also moves to
 the next arg(s) in the list from the passed in arg pointer.
 The number of chars written is returned (not incl \0).
*************************************************************/
static long _ffmt(char *outptr, char *fmt, long *argptr)
{
char numstk[33], *ptr, justify, zero, minus, chr;
unsigned long width, value, i, total;

    total = 0;
    while(chr = *fmt++)
    {
        if(chr == '%')
        {                       /* format code */
            chr = *fmt++;
            ptr = &numstk[32];
            *ptr = justify = minus = 0;
            width = value = i = 0;
            zero = ' ';
            if(chr == '-')
            {                   /* left justify */
                --justify;
                chr = *fmt++;
            }
            if(chr == '0')                  /* leading zeros */
                zero = '0';
            while(isdigit(chr))
            {           /* field width specifier */
                width = (width * 10) + (chr - '0');
                chr = *fmt++;
            }
```

```
value = *--argptr;              /* get parameter value */

switch(chr)
{
    case 'd' :                  /* decimal number */
        if(value & 0x80000000)
        {
            value = -value;
            ++minus;
        }
    case 'u' :                  /* unsigned number */
        i = 10;
        break;
    case 'x' :                  /* hexidecimal number */
    case 'X' :
        i = 16;
        break;
    case 'o' :                  /* octal number */
        i = 8;
        break;
    case 'b' :                  /* binary number */
        i = 2;
        break;
    case 'c' :                  /* character data */
        *--ptr = value;
        break;
    case 's' :                  /* string */
        ptr = value;            /* value is ptr to string */
        break;
    default:                    /* all others */
        *--ptr = chr;
        ++argptr;               /* backup to last arg */
}

if(i)       /* for all numbers, generate the ASCII string */
    do
    {
        if((chr = (value % i) + '0') > '9')
            chr += 7;
        *--ptr = chr;
    }
    while(value /= i);

/* output sign if any */

if(minus)
{
    *outptr++ = '-';
    ++total;
    if(width)
        --width;
}

/* pad with 'zero' value if right justify enabled  */

if(width && !justify)
{
    for(i = strlen(ptr); i < width; ++i)
```

continues

Listing 11.2. continued

```
                *outptr++ = zero;
                ++total;
            }

            /* move in data */

            i = 0;
            value = width - 1;

            while((*ptr) && (i <= value))
            {
                *outptr++ = *ptr++;
                ++total;
                ++i;
            }

            /* pad with 'zero' value if left justify enabled */

            if(width && justify)
            {
                while(i < width)
                {
                    *outptr++ = zero;
                    ++total;
                    ++i;
                }
            }
        }
        else
        {
            /* not format char, just move into string  */
            *outptr++ = chr;
            ++total;
        }
    }

    *outptr = 0;
    return total;
}

/***********************************
    Formatted print to screen
***********************************/

long xprintf(char *fmt, ...)
{
    va_list ap;
    long total;
    char buffer[S_SIZE];

    va_start(ap, fmt);      /* set up ap pointer */
    total = _ffmt(buffer, fmt, ap);
    TTYOut(buffer, strlen(buffer), Color);
    va_end(ap, fmt);
    return total;
}
```

```
/***********************************
    Formatted print to string s
***********************************/

long xsprintf(char *s, char *fmt, ...)
{
    va_list ap;
    long total;

    va_start(ap, fmt);          /* set up ap pointer */
    total = _ffmt(s, fmt, ap);
    va_end(ap, fmt);
    return total;
}

/*********************************************
 Checks to ensure we don't scroll the function
 keys off the screen.
*********************************************/

void CheckScreen()
{
long iCol, iLine;

    GetXY(&iCol, &iLine);
    if (iLine >= 23)
    {
        ScrollVid(0,1,80,23,1);
        SetXY(0,22);
    }
}

/*****************************************************
    This is called to initialize the screen.
*****************************************************/

static void InitScreen(void)
{
    ClrScr();
    xsprintf(&rgStatLine[70], "%d", tick);
    PutVidChars(0,0, rgStatLine, 80, WHITE¦BGBLUE);
    PutVidChars(0,  24, rgMonMenu1, 26, BLUE¦BGWHITE);
    PutVidChars(27, 24, rgMonMenu2, 26, BLUE¦BGWHITE);
    PutVidChars(54, 24, rgMonMenu3, 25, BLUE¦BGWHITE);
    SetXY(0,1);
    return;
}

/*****************************************************
    This is the status task for the Monitor.
    Besides displaying the top status line for
    the monitor, it has the job of looking for
    messages from the job code that indicate
    a job has terminated. When it gets one,
    it recovers the last of the resources and
    then notifies the user on the screen.
*****************************************************/
```

continues

Listing 11.2. continued

```
static void StatTask(void)
{
unsigned long erc, i, Exch, Msg[2];
U8 *pPD, *pVid;

for(;;)
{
    GetCMOSTime(&time);
    rgStatLine[10] = '0' + ((time >> 20) & 0x0f);
    rgStatLine[11] = '0' + ((time >> 16) & 0x0f);
    rgStatLine[13] = '0' + ((time >> 12) & 0x0f);
    rgStatLine[14] = '0' + ((time >> 8) & 0x0f);
    rgStatLine[16] = '0' + ((time >> 4) & 0x0f);    /* seconds */
    rgStatLine[17] = '0' + (time & 0x0f);

    GetCMOSDate(&date);
    rgStatLine[0] = '0' + ((date >> 20) & 0x0f); /* month */
    rgStatLine[1] = '0' + ((date >> 16) & 0x0f);
    rgStatLine[3] = '0' + ((date >> 12) & 0x0f); /* day */
    rgStatLine[4] = '0' + ((date >> 8)  & 0x0f);
    rgStatLine[6] = '0' + ((date >> 28) & 0x0f); /* year */
    rgStatLine[7] = '0' + ((date >> 24) & 0x0f);

    GetTimerTick(&tick);
    xsprintf(&rgStatLine[70], "%d", tick);
    PutVidChars(0,0, rgStatLine, 80, WHITE|BGBLUE);

    Sleep(50);  /* sleep 0.5 second */

    GetTimerTick(&tick);
    xsprintf(&rgStatLine[70], "%d", tick);
    PutVidChars(0,0, rgStatLine, 80, WHITE|BGBLUE);

    Sleep(50);  /* sleep 0.5 second */

    /* Now we check for tasks that are Jobs that are killing
    themselves (either due to fatal errors or no exitjob).
    The message has Error, TSSExch in it.
    */

    erc =  CheckMsg(KillExch, KillMsg);
    if (!erc)
    {    /* someone's there wanting to terminate...*/

        /* Get and save the error (KillMsg[0]) */
        KillError = KillMsg[0];

        /* Call GetExchOwner which gives us pJCB */

        erc = GetExchOwner(KillMsg[1], &pJCB);
        if (!erc)
        {

            KillJobNum = pJCB->JobNum;
            Tone(440,50);
```

```
            xprintf("Job number %d terminated. Error: %d\r\n",
                    KillJobNum, KillError);
            CheckScreen();

            pPD  = pJCB->pJcbPD;
            pVid = pJCB->pVirtVid;

            /* Must change video to monitor if this guy owned it */

            GetVidOwner(&i);
            if (i == KillJobNum)
2            {
                GetTSSExch(&Exch);  /* Use our TSS exchange for Request */
                SetVidOwner(1);
                erc = Request("KEYBOARD", 4, Exch, &i, 0,
                    0, 0,   0, 0,   1, 0, 0);
                erc = WaitMsg(Exch, Msg);
            }

            /* Now we deallocate the JCB and the TSSExch
            which will free the TSS automatically!
            */

            DeAllocExch(KillMsg[1]);
            DeAllocJCB(pJCB);   /* No error returned */

            /* When the JCB was created, the PD and its 1st
            PDE (PT) were allocated as two pages next to each other
            in linear memory.  So we can just deallocate both
            pages in one shot. Then we deallocate the single
            page for virtual video.
            */

            DeAllocPage(pPD, 2);
            DeAllocPage(pVid, 1);

            fKilled = 1;
            /* We're done (and so is he...) */
        }
    }

} /* for EVER */
}

/*******************************************************
    This is the Manager task for the Monitor.
    It allows us to switch jobs with the
    Ctrl-Alt-PageDown key.
    We don't actually switch jobs, we just
    ressign video and keyboard to the next
    active job (except the Debugger).
    Also, if the debugger has the video...
    we don't do it at all!
    This also looks for Ctrl-Alt-Delete which kills
    the job that owns the keyboard/video so long as it's
    not the Monitor or the Debugger.
*******************************************************/
```

continues

Listing 11.2. continued

```
static void MngrTask(void)
{
long erc, i, j, fDone;
char *pJCB;

/* Leave a Global Key Request outstanding with KBD service
   for the status task
*/

erc = Request("KEYBOARD", 2, MngrExch, &MngrHndl, 0, &gcode,
              4, 0, 0, 0, 0, 0);

for(;;)
{
    erc = WaitMsg(MngrExch, MngrMsg);

    if (!erc)
    {
        if ((gcode & 0xff) == 0x0C)
        {
            /*  Find next valid Job that is NOT the
                debugger and assign Vid and Keyboard to
                it.
            */

            erc = GetVidOwner(&j);
            fDone = 0;
            i = j;
            while (!fDone)
            {
                i++;
                if (i==2) i = 3;
                else if (i>34) i = 1;
                erc = GetpJCB(i, &pJCB);        /* erc = 0 if valid JCB */
                if ((!erc) || (i==j))
                    fDone = 1;
            }
            if (i != j)
            {
                SetVidOwner(i);
                erc = Request("KEYBOARD", 4, MngrExch, &MngrHndl, 0,
                    0, 0,   0, 0,   i, 0, 0);
                erc = WaitMsg(MngrExch, MngrMsg);

            }
        }
        else if ((gcode & 0xff) == 0x7F)    /* CTRL-ALT-DEL (Kill)*/
        {
            erc = GetVidOwner(&j);
            erc = KillJob(j);
        }

        /* leave another global key request */
        erc = Request("KEYBOARD", 2, MngrExch, &MngrHndl, 0, &gcode,
                  4, 0, 0, 0, 0, 0);
    }
```

```
} /* for EVER */
}

/*******************************************************
  This simply does a software interrupt 03 (Debugger).
*******************************************************/

static void GoDebug(void)
{
;
#asm
    INT 03
#endasm
return;
}

/*******************************************************
  This strobes keyboard data port 60h with 00 after
  sending 0D1 command to Command port.  We have to loop
  reading the status bit of the command port to make
  sure it's OK to send the command. This resets the
  processor which then executes the boot ROM.
*******************************************************/

static void Reboot(void)
{
;
#asm
        CLI                     ;first we clear interrupts
        MOV ECX, 0FFFFh         ;check port up to 64K times
Reboot0:
        IN AL,64h               ;Read Status Byte into AL
        TEST AL,02h             ;Test The Input Buffer Full Bit
        LOOPNZ Reboot0
        MOV AL,0FEh             ;Strobe bit 0 of keyboard crtlr output
        OUT 64h,AL
        STI
#endasm
return;
}

/*******************************************************
  This reads a file called Initial.Job from the system
  directory and loads all jobs specified in the file.
  Video and keyboard are not assigned to any of these
  jobs unless it is cli.run and the last job loaded.
*******************************************************/

void LoadJobFile(void)
{
long erc, fh, sjobfile, cbrunfile, i, j, job;
unsigned char sdisk;
char ajobfile[50];
char arunfile[80];
char fdone, fcli;

    GetSystemDisk(&sdisk);
```

continues

Listing 11.2. continued

```
    sdisk &= 0x7F;
    sdisk += 0x41;      /* 0=A, 1=B, 2=C etc. */
    ajobfile[0] = sdisk;
    CopyData(":\\MMSYS\\INITIAL.JOB\0", &ajobfile[1], 20);
    sjobfile = strlen(ajobfile);

    erc =  OpenFile(ajobfile, sjobfile, 0, 1, &fh);
    if (!erc)
    {
        fdone = 0;
        job = 0;
        fcli = 0;
        while (!fdone)
        {
            i = 0;
            do
            {
                erc = ReadBytes(fh, &arunfile[i++], 1, &j);

            } while ((!erc) && (arunfile[i-1] != 0x0A) && (i < 80));

            if ((!erc) && (i > 1))
            {
                if (arunfile[0] == ';')  /* a comment line */
                    continue;

                cbrunfile = 0;
                while ((arunfile[cbrunfile] != 0x0A) &&
                       (arunfile[cbrunfile] != 0x0D) &&
                       (arunfile[cbrunfile] != 0x20) &&
                       (arunfile[cbrunfile] != 0x09) &&
                       (arunfile[cbrunfile]))
                    cbrunfile++;

                if (cbrunfile > 2)
                {
                    arunfile[cbrunfile] = 0; /* null terminte for display */

                    if ((cbrunfile > 8) &&
                        (CompareNCS(&arunfile[cbrunfile-7],
                                    "cli.run", 7) == -1))
                        fcli = 1;
                    else
                        fcli = 0;

                    xprintf("Loading: %s...\r\n", arunfile);
                    CheckScreen();
                    erc =  LoadNewJob(arunfile, cbrunfile, &job);
                    if (!erc)
                    {
                        xprintf("Successfully loaded as job %d\r\n", job);
                        CheckScreen();
                        Sleep(50);
                    }
                    else
                    {
```

```
                    xprintf("ERROR %d Loading job\r\n", erc);
                    CheckScreen();
                    Sleep(50);
                    job = 0;
                    erc = 0;
                }
            }
        }
        else fdone = 1;
    }
    CloseFile(fh);

    /* if the last succesfully loaded job was a CLI,
       assign the keybaord and video to it.
    */

    if ((job > 2) && (fcli))
    {
        SetVidOwner(job);
        erc = Request("KEYBOARD", 4, GP1Exch, &GP1Hndl, 0,
            0, 0,   0, 0,   job, 0, 0);
        if (!erc)
            erc = WaitMsg(GP1Exch, GP1Msg);
    }
  }
  else
  {
    xprintf("INITIAL.JOB file not found in system directory.\r\n");
    CheckScreen();
  }
}


/*******************************************************
  This Loads the MMURTL Command Line Interpreter
  and switches video and keyboard to give the
  user access to it.
*******************************************************/

static long LoadCLI(void)
{
long erc, job;
unsigned char sdisk, acli[40];

  GetSystemDisk(&sdisk);
  sdisk &= 0x7F;
  sdisk += 0x41;      /* 0=A, 1=B, 2=C etc. */
  acli[0] = sdisk;
  CopyData(":\\MMSYS\\CLI.RUN\0", &acli[1], 16);
  xprintf("Loading: %s...", acli);

  erc =  LoadNewJob(acli, strlen(acli), &job);
  if (!erc)
  {
    xprintf("New CLI Job Number is: %d\r\n", job);
    CheckScreen();
    Sleep(50);
    SetVidOwner(job);
```

continues

Listing 11.2. continued

```
        erc = Request("KEYBOARD", 4, GP1Exch, &GP1Hndl, 0,
            0, 0,   0, 0,   job, 0, 0);
        if (!erc)
            erc = WaitMsg(GP1Exch, GP1Msg);
    }
return erc;
}

/*******************************************************
    This is the main procedure called from the OS after
    all OS structures and memory are initialized.
*******************************************************/

void Monitor(void)
{
long erc, i, j, k, iCol, iLine;
unsigned long ccode, ccode1;
unsigned char c;
char text[70];

InitScreen();

Tone(250,15);       /* 250 Hz for 150ms */
Tone(1000,33);      /* 250 Hz for 330ms */

/* Allocate an exchange for the Manager task global keycode */

erc = AllocExch(&MngrExch);
if (erc)
  xprintf("AllocExch (Mngr Exch) Error: %d\r\n", erc);

erc = SpawnTask( &StatTask, 24, 0, &StatStack[255], 1 );    /* Task 4 */
if (erc)
  xprintf("SpawnTask (StatTask) Error: %d\r\n", erc);

erc = AllocExch(&KillExch);
if (erc)
  xprintf("AllocExch (Kill Exch) Error: %d\r\n", erc);

Color = YELLOW|BGBLACK;
xprintf("MMURTL (tm) - Message based, MUltitasking, Real-Time kerneL\r\n");
xprintf("Copyright (c) R.A.Burgess, 1991-1995  ALL RIGHTS RESERVED\r\n\r\n");

Color = WHITE|BGBLACK;

c = ((BootDrive & 0x7f) + 0x41);
xprintf("BootDrive: %c\r\n", c);

i = (oMemMax+1)/1024;
xprintf("Total memory (Kb): %d\r\n", i);

erc = QueryPages(&nMemPages);
i = (nMemPages*4096)/1024;
xprintf("Free memory  (Kb): %d\r\n", i);

erc = InitKBDService();
```

```
xprintf("Init KBD Service Error: %d\r\n", erc);

erc = coms_setup();
xprintf("Init Serial Comms Device Driver Error: %d\r\n", erc);

erc = lpt_setup();
xprintf("Init Parallel LPT Device Driver Error: %d\r\n", erc);

/* Allocate general purpose exchanges to use in the monitor */

erc = AllocExch(&GPExch);
if (erc)
  xprintf("AllocExch Error: %d\r\n", erc);

erc = AllocExch(&GP1Exch);
if (erc)
  xprintf("AllocExch GP1 Error: %d\r\n", erc);

xprintf("Init floppy device driver... Error: ");
erc = fdisk_setup();
xprintf("%d\r\n", erc);

xprintf("Init hard disk device driver... Error: ");
erc = hdisk_setup();
xprintf("%d\r\n", erc);

xprintf("Initializing file system...\r\n");
erc = InitFS();
xprintf("File System... Error: %d\r\n", erc);

/* Spawn manager task */

SpawnTask( &MngrTask, 10, 0, &MngrStack[255], 1 );

/*
   Call LoadJobFile to read job file from system directory
   and execute and jobs listed there.
*/

LoadJobFile();

for (;;)  /* Loop forEVER looking for user desires */
{

    /* Make a ReadKbd Key Request with KBD service. Tell it
    to wait for a key.
    */

    erc = Request("KEYBOARD", 1, GPExch, &GPHndl, 0, &ccode,
                    4, 0, 0, 1, 0, 0);
    if (erc)
        xprintf("Kbd Svc Request KERNEL ERROR: %d\r\n", erc);

    /* wait for the keycode to come back */

    erc = WaitMsg(GPExch, GPMsg);
    if (erc)
     xprintf("KERNEL Error from Wait msg:  %d\r\n", erc);
```

continues

Listing 11.2. continued

```
c = ccode & 0xff; /* lop off everything but the key value */

switch (c)
{
case 0x0F:      /* F1 Run */
        erc = LoadCLI();
        if (erc)
            xprintf("Error from LoadCLI:  %d\r\n", erc);
        break;
case 0x10:      /* F2 Jobs */
        InitScreen();
        j = 2; /* Line */
        k = 0; /* Col offset */
        for (i=1; i<nMaxJCBs; i++)
        {
            if (j > 20)
                k = 40;
            erc = GetpJCB(i, &pJCB);        /* erc = 0 if valid JCB */
            if (!erc)
            {
                SetXY(k,j);
                xprintf("Job: %d\r\n", pJCB->JobNum);
                SetXY(k+10,j);
                CopyData(&pJCB->sbJobName[1], text, 13);
                text[pJCB->sbJobName[0]] = 0;
                xprintf("Name: %s\r\n", text);
                j++;
            }
        }
        break;
case 0x11:      /* F3 Stats - loops displaying status till key is hit */
        InitScreen();
        while (erc = ReadKbd(&ccode1, 0))
        { /* ReadKbd no wait until no error */
            SetXY(0,1);
            erc = QueryPages(&nMemPages);
            xprintf("Any key to dismiss status... \r\n");
            xprintf("Free 4K memory pages:      %d\r\n", nMemPages);
            xprintf("Task switches total:       %d\r\n", nSwitches);
            xprintf("Preemptive task switches:  %d\r\n", nSlices);
            xprintf("CPU idle ticks (no work):  %d\r\n", nHalts);
            xprintf("Tasks Ready to Run:        %d\r\n", nReady);
            xprintf("Free Task State Segments:  %d\r\n", nTSSLeft);
            xprintf("Free Job Control Blocks:   %d\r\n", nJCBLeft);
            xprintf("Free Request Blocks:       %d\r\n", nRQBLeft);
            xprintf("Free Link Blocks:          %d\r\n", nLBLeft);
            xprintf("Free Exchanges:            %d\r\n", nEXCHLeft);
            SetXY(0,1);
            PutVidChars(29, 1, "|",  1, GREEN|BGBLACK); Sleep(9);
            PutVidChars(29, 1, "/",  1, GREEN|BGBLACK); Sleep(9);
            PutVidChars(29, 1, "-",  1, GREEN|BGBLACK); Sleep(12);
            PutVidChars(29, 1, "\\", 1, GREEN|BGBLACK); Sleep(9);
            PutVidChars(29, 1, "|",  1, GREEN|BGBLACK); Sleep(9);
            PutVidChars(29, 1, "/",  1, GREEN|BGBLACK); Sleep(9);
            PutVidChars(29, 1, "-",  1, GREEN|BGBLACK); Sleep(12);
```

```
                PutVidChars(29, 1, "\\", 1, GREEN¦BGBLACK); Sleep(9);
                PutVidChars(29, 1, " ",  1, GREEN¦BGBLACK);
            }
            SetXY(0,12);
            xprintf ("\r\n");
            break;
    case 0x16:      /* F8 Reboot */
            xprintf("\r\nF8 again to reboot, any other key to cancel");
            erc = ReadKbd(&ccode1, 1);
            if ((ccode1 & 0xff) == 0x16)
                Reboot();
            xprintf("...Cancelled\r\n");
            break;
    case 0x18:      /* F10 Debug */
            GoDebug();
            break;
    case 0x00:      /* No Key */
            Sleep(3);   /* Sleep for 30 ms */
            break;
    case 0x12:      /* F4 - future use */
    case 0x13:      /* F5  */
    case 0x14:      /* F6  */
    case 0x15:      /* F7  */
    case 0x17:      /* F9  */
    case 0x19:      /* F11 */
    case 0x1A:      /* F12 */
            break;
    default:
        if (((c > 0x1F) && (c < 0x80)) ¦¦
            (c==0x0D) ¦¦ (c==8))
            {
                if (c==0x0D)
                    TTYOut (CRLF, 2, WHITE¦BGBLACK);
                else
                    TTYOut (&c, 1, WHITE¦BGBLACK);
            }
 2  }
    GetXY(&iCol, &iLine);
    if (iLine >= 23)
    {
        ScrollVid(0,1,80,23,1);
        SetXY(0,22);
    }

} /* for EVER */

}
```

12

The Debugger

The debugger is built into the operating system. It is not a separate run file. The debugger provides the following functions:

Display of instructions (disassembled)
Dumping of linear memory as bytes or dwords
Display of exchanges, and waiting messages or tasks
Display of active tasks
Set and clear an instruction breakpoint
Full register display
Selection of linear address to display
Display of important OS structure addresses

Using the Debugger

In its current incarnation, the debugger is an assembly language, nonsymbolic debugger. Extensive knowledge of the Intel processor instruction set is required to properly use the debugger.

Entering the Debugger

The debugger may be entered using the Debugger function key in the monitor, the Debug command in the command-line interpreter, or by placing an `INT 03` instruction anywhere in your application's assembly language file. See the "Debugger Theory" section for more information on `INT 03` instruction usage.

The debugger may also start on its own if certain exceptions occur. Some exceptions are designed to take you directly into the debugger and display an error code indicating the problem. The most common exceptions are the General Protection Fault (0D hex) and the Page Fault (0E hex). When the exception occurs, the debugger is entered, and a red banner is displayed, along with the exception number. The registers can be examined to see where it happened, and why. This usually provides enough information to correct the offending code.

Exiting the Debugger

Pressing the Esc key will exit the debugger and begin execution at the next instruction.

If the debugger was entered on a fault, pressing Esc will restart the debugger at the offending address and you will re-enter the debugger. See the "Debugger Theory" section for more information.

The debugger will also exit using the Single Step command (F1 function key), and will be re-entered immediately following the execution of the next complete instruction.

Debugger Display

The debugger display is divided into three sections. The complete general register set is displayed along the right side of the screen, function keys are across the bottom, and the left side is the instruction and data display area.

Certain debugger display functions, such as dumping data, will clear the screen for display. The registers and function keys will be redisplayed when the function is finished.

Debugger Function Keys

The debugger function key actions are described here:

F1 SStep— Single Step. This returns to the application and executes one instruction, after which it immediately returns to the debugger where the next active code address and its associated instruction are displayed.

F2 SetBP— Set Breakpoint. This sets the breakpoint at the currently displayed instruction. You may move down through the instructions, without executing them, using the down arrow key. You may also use F8 CrntAdd (Current Address) to select and display a new address to display before setting the breakpoint.

F3 ClrBP—Clear Breakpoint. This clears the single breakpoint.

F4 CS:EIP—Goto CS:EIP. This redisplays the current instruction pointer address. This becomes the active address.

F5 Exch—Display Exchanges. This displays all exchanges and shows messages or tasks that may be waiting there. Only exchanges that actually have a message or task will be displayed.

F6 Tasks—Display Tasks. This displays all active tasks on the system. The task number, the associated job number, the address of the task's JCB, the address of the TSS, and the priority of the task are displayed with each one.

F8 CrntAdd—Change Current Address. This enables you to set the code address the debugger is displaying (disassembling). This *does not* change the address scheduled for execution. The F4 CS:EIP function key can be used to return to the next address scheduled for execution.

F9 DumpB—Dump Bytes. This requests a linear address, and then dumps the bytes located at that address. The address is displayed on the left, and all the data, including the ASCII, is shown to the right. The format is the same as the dump command in the CLI.

F10—Dump Dwords. This requests a linear address, and then dumps the dwords located at that address. The address is displayed on the left, and all the data, including the ASCII, is shown to the right. Each byte of a dword is displayed in the proper order (high order, next, next, low order). This command is useful for dumping structures that are dword oriented, as most of them are.

F12 AddInfo—Address Information. This lists names and linear addresses of important structures used during OS development. These structures can also be very useful during program debugging. These values are hard coded at system build. The abbreviations are given in the following list, with the full name of the structure and what use it may be during debugging.

IDT—Interrupt Descriptor Table. This can be dumped to view interrupt vector types and addresses. This may be useful if you have set up an interrupt and you want to ensure it was encoded properly and placed in the table.

GDT—Global Descriptor Table. This can be dumped to view all of the GDT entries. The GDT contains many items, including the TSS descriptor entries for your program. TSS descriptors point to your task state segment and also indicate the privilege level of the task.

RQBs—Request Block Array. This is the address of the first request block in the array of request blocks allocated during system initialization. The request handle that you receive from the request primitive is actually a pointer, a linear address, that will be somewhere in this array.

TSS1—First TSS. The first two task state segments are static structures and are used for the operating system Monitor and debugger. This is the address of the monitor TSS. The debugger TSS is 512 bytes after this address.

TSS3—Dynamic TSSs. The rest of the task state segments are in several pages of allocated memory and are initialized during system startup. TSSs are numbered sequentially, which makes the first of the dynamic TSSs numbered three. Each TSS is 512 bytes.

LBs—Link Blocks. These are 16-byte structures used as links in chains of messages and tasks waiting at an exchange. The array of link blocks are allocated as a static array.

RdyQ—Ready Queue. This is the address of the 32 queue structures where tasks that are ready to run are queued.

JCBs—Job Control Blocks. This the address of the array of Job Control Blocks. This is an array of 512-byte structures. The JCB is discussed in detail in Chapter 9, "Application Programming."

SVCs—Services Array. This is the address of the array of currently registered services. The names of the services and the exchanges they service are contained here.

Exch—Exchanges. This is the address of the array of exchanges in allocated memory. When you receive an exchange number from the `AllocExch()` function, it is the index of that exchange in the array. Each exchange is 16 bytes.

aTmr—Timer Blocks. This is the address of the array of timer blocks. The `Sleep()` and `Alarm()` functions each set up a timer block.

Debugging Your Application

If your application is "going south" on you (crashing, locking up, entering the debugger, or whatever), you can set breakpoints in several locations of your program by editing the .ASM files, listed in your ATF file. You would insert the `INT 03` instruction wherever you feel it would be useful.

In some cases, just knowing how far your program made it is enough to solve the problem. You can start by searching for `_main` in your primary assembly language file generated by the C compiler, or for `.START` in an assembly language program, and placing an `INT 03` instruction there.

It may help you to be able to identify high-level entry and exit code sequences (begin and end of a C function), and also how local variables are accessed.

The following example shows typical entry and exit code found in a high-level procedure.

```
PUSH EBP
MOV EBP, ESP
; Lots of code here
MOV ESP, EBP
POP EBP
RETN
```

When local variables are accessed, they are always referenced below the frame pointer in memory. A single 4-byte integer is referenced as [EBP-4] or [EBP+FFFFFFFC], and displayed as an unsigned number.

Stack parameters, or arguments, are always referenced above the frame pointer, such as [EBP+12]. With *near* calls like you would make inside your program, the last parameter is always [EBP+8].

There is a symbolic debugger in MMURTL's near future, but for right now, it's down and dirty troubleshooting.

Debugger Theory

The Intel 32-bit processors have very powerful debugging features. They have internal debug registers that allow you to set code and data breakpoints. What this means is that you can set the processor to execute an interrupt, actually a trap, whenever any linear memory location is accessed, or at the beginning of any instruction.

The `INT 03` instruction may also be placed in your code, like with the older processors, without having to fill in special registers. There are four debug address registers in the processor, which means you can have four active breakpoints in your program if you use only the registers. The MMURTL debugger currently uses only one of these registers and allows only instruction breakpoints to be set.

The INT 03 interrupt is just like any other on the system. When it occurs, entry 3 in the interrupt table is used to vector to a procedure or task. In MMURTL, it is an interrupt procedure that will switch to the debugger task after some fix-ups have been completed.

Debugging in a multitasking operating system is complicated to say the least. The 386/486 processors make it much easier because they let you determine if the breakpoint is local to a single task or global to all tasks. Remember that several tasks may be executing the same piece of code. If you set a breakpoint using the INT 03 instruction, every task that executes it will produce the interrupt and enter the debugger. For applications, this really doesn't apply as much because they each have their own linear address space. The paging hardware translates the linear address to a physical address for its own use.

MMURTL's debugger is set up as a separate job. It has its own Job Control Block (JCB) and one task (its own TSS). The debugger is the highest priority task on the system (level 1). No other task operates at a priority that high. The debugger also has its own virtual screen, just like applications do.

When writing an operating system, the debugger becomes one of the most important tools you can have. If you have problems before the debugger is initialized, it can mean hours of grueling debugging with very little indication of what's going on, usually a blank screen and nothing else.

Even though the debugger has its own JCB and looks like another application, it has special privileges that allow it to operate outside the direct control of the kernel. It can remove the currently running task and make itself run, and do the reverse without the kernel even being aware of what has happened. The debugger must be able to do this to get its job done.

When the debug interrupt activates, you initially enter an interrupt procedure that places the running task in a hold status. Hold status simply means that it is not placed on the ready queue as if it were a normal task switch. Instead, the current pRunTSS, a pointer to the currently running TSS, is saved by the debugger, and the interrupt procedure does a task switch to the debugger's task. The debugger becomes the active job. This effectively replaces the running task with the debugger task.

A few housekeeping chores must be done before the task switch is made. These include copying the interrupted task's page directory entry into the debugger's JCB and the debugger's TSS. This is so the debugger is using the task's exact linear memory image. If you are going to debug a task, you must be able to access all of its memory, including allocated memory.

The debugger *cannot* debug itself. It's actually an orphan when it comes to being a program in its own right. It doesn't have its own Page Directory or Page Tables like other jobs (programs). It almost, but not quite, becomes a task in the job it interrupted.

The INT 03 interrupt is not the only way to enter MMURTL's debugger. The debugger is also entered when other fatal processor exceptions occur. If you look through the 386/486 documentation, you will see that there are many interrupts, called exceptions, or faults, that can

occur. Some of these are used by the OS to indicate something needs to be done, while others should not happen and are considered fatal for the task that caused them. Until MMURTL uses demand page virtual memory, a page fault is fatal to a task. It means it tried to access memory that didn't belong to it. The exceptions that cause entry into the debugger include *faults*, *traps*, and *aborts* (processor fatal) and are shown in Table 12.1.

Table 12.1. Processor exceptions.

No.	*Type*	*Description*
0*	F	Divide by zero
1	T/F	Debug exception (debugger uses for single step)
3	T	Breakpoint (set by debugger, or `INT 03` in code)
4	T	Overflow (`INT0` instruction)
5*	F	Bounds check (from bound instruction)
6*	F	Invalid opcodes (reserved instructions)
7*	F	Coprocessor not available (on Esc and `wait`)
8*	A	Double fault (real bad news...)
9		
10*	F	Invalid TSS
11*	F	Segment not present
12*	F	Stack fault
13*	F/T	General protection fault
14*	F	Page fault
16	F	Coprocessor error

The asterisk (*) indicates that the return address points to faulting instruction. This return address is used by the debugger entry code.

Some of the exceptions even place an error code on the stack after the return address. MMURTL has a very small interrupt procedure for each of these exceptions. These procedures get the information from the stack before switching to the debugger. The debugger uses this information to help you find the problem. When the debugger is entered, it displays the registers from the interrupted task. To do this, it uses the values in the task's Task State Segment (TSS). All the values are the same as when the exception occurred, except the CS and EIP. These contain the address of the exception handler. To be able to see the proper address of the code that was interrupted, you must get the return address from the stack and place it back into the TSS. This also has the effect of starting the task back at the instruction *after* it was interrupted, which is the next instruction to execute. This makes the debugger transparent, which is exactly what you want. This also allows you to restart the application after a breakpoint at the proper place.

If the debugger was entered because of a fault, you should kill the offending job and restart, setting a breakpoint *before* the exception occurs, then single step while watching the registers to see where the problem is.

The debugger exit code is just about the reverse of the code used to enter the debugger. The debugger removes itself as the running task and returns the interrupted task to the running state with the values in the TSS. This also includes switching the video back to the rightful owner.

While you are in the debugger, all the registers from the interrupted task are displayed, along with any errors that may have been removed from the stack on a fault.

13

Keyboard Service

The keyboard is handled with a system service. As described in Chapter 10, "Systems Programming," system services are accessed with the Request primitive. The keyboard is handled with a system service because it, like the file system, is a shared resource. Several programs can have outstanding requests to the keyboard service at one time. Only one program (job) will be assigned to receive keystrokes at any one time. The exception to this rule is the global keyboard request to receive Ctrl-Alt keystrokes.

The service name is `KEYBOARD` (uppercase as always). Each of the functions is identified by its service code number. here is an example of a keyboard service request:

```
erc = Request(
        "KEYBOARD",  /* Service Name */
        1,           /* wSvcCode for ReadKbd */
        MyExch,      /* dRespExch  */
        &MyRqhandle, /* pRqHndlRet */
        0,           /* npSend (no Send Ptrs) */
        &KeyCode     /* pData1  */
        4,           /* cbData1 (size of KeyCode)*/
        0            /* pData2 - not used (0) */
        0,           /* Not used (0)   */
        1,           /* dData0 - fWait for Key */
        0,           /* dData1 - Not used (0) */
        0);          /* dData2 - Not used (0) */
```

All message-based services use the same request interface. Using the request interface allows for asynchronous program operation. You can make a request and then go do something else before you return to wait or to check the function to see if you received a response from your request. This is true multitasking.

If you don't need asynchronous keyboard access, the public call `ReadKbd()` is provided. This is a blocking call with only two parameters. It's easier to use but not as powerful. I'll describe it after I discuss the following services.

Available Service

Table 13.1 shows the services provided by the service code.

Table 13.1. Available keyboard services.

Service Code	*Function*
1	Read Keyboard
2	Notify On Global Keys
3	Cancel Notify On Global Keys
4	Assign Keyboard

The following sections describe these four functions.

Read Keyboard

The Read Keyboard function (1) allows a program to request keyboard input from the service. The first request pointer points to an unsigned dword where the key code will be returned. The key codes are described in detail in tables later in this chapter. The `dData0` determines whether the service will hold the request until a key is available. A value of 0 means the request will be sent back to you immediately, even if a key is not available. The error from the service is `ErcNoKeyAvailable` (700) if no key was available. The key code is undefined if no key was available (so you must check the error). A value of 1 in `dData0` causes the service to hold your request until a key is available. If a key was available, the error should be 0.

The following are the Request parameters for Read Keyboard:

```
wSvcCode = 1
npSend   = 0
pData1   = Pointer where the KeyCode will be returned
dcbData1 = 4 - Count of bytes in the code
pData2   = 0 - Not used
dcbData2 = 0 - Not used.
dData0   = fWaitForKey (0 or 1)
         0 - Return with error if no key
           waiting
         1 - Wait for Key
dData1   = 0 - Not used
dData2   = 0 - Not used
```

Notify On Global Keys

The Notify On Global Keys function (2) allows a program to look for any keystroke entered with the Ctrl and Alt keys depressed. This allows for "hot-key" operation.

Unlike regular keystrokes, keys that are pressed when the Ctrl-Alt keys are pressed are not buffered. This means that users of the Read Keyboard function (1) will not see these keys. It also means that your application must have an outstanding Notify request with the keyboard service to receive these keys. If you have an outstanding Notify request and you no longer want to receive Global key notifications, you should send a Cancel Notify request (service code 3).

The Global key codes returned are identical to the regular key codes, described in detail later.

These are the parameters for the Notify On Global Keys function:

```
wSvcCode = 2
npSend   = 0
pData1   = Pointer where the KeyCode will be returned
dcbData1 = 4 - Count of bytes in the code
```

```
pData2  = 0 - Not used
dcbData2 = 0 - Not used
dData0  = 0 - Not used
dData1  = 0 - Not used
dData2  = 0 - Not used
```

Cancel Notify On Global Keys

The Cancel Notify On Global Keys function (3) cancels an outstanding Notify On Global Keys request. If you have an outstanding Notify request, this function cancels it. The Notify request is sent back to the exchange with an error stating it was canceled. The Cancel Notify request is returned also. This cancels *all* outstanding Notify requests from the same job number for your application.

Here are the parameters for Notify On Global Keys function:

```
wSvcCode = 3
npSend   = 0
pData1   = 0 - Not used
dcbData1 = 0 - Not used
pData2   = 0 - Not used
dcbData2 = 0 - Not used
dData0   = 0 - Not used
dData1   = 0 - Not used
dData2   = 0 - Not used
```

Assign Keyboard

The Assign Keyboard request assigns a new job to receive keys from the service. This request should be used only by services such as the Monitor or a program that manages multiple jobs running under the MMURTL OS.

These are the parameters for the Assign Keyboard request:

```
wSvcCode = 3
npSend   = 0
pData1   = 0 - Not used
dcbData1 = 0 - Not used
pData2   = 0 - Not used
dcbData2 = 0 - Not used
dData0   = x - New Job Number (1 to nJobs)
dData1   = 0 - Not used
dData2   = 0 - Not used
```

Key Codes and Status

MMURTL supports the standard AT 101-key advanced keyboard. All ASCII text and punctuation is supported directly, while providing complete keyboard status to allow further translation for all ASCII control codes.

Alphanumeric Key Values

The key code returned by the keyboard service is a 32-bit (4-byte) value. The low-order byte is the actual key code, as shown in Table 13.4.

To eliminate all the key status from the 4-byte key code to get the keystroke value itself, logically AND the key code by 0FFh (0xff in C). This leaves you with the 8-bit code for the key. The key will be properly shifted according to the state of the Shift and Lock keys.

The upper three bytes provide the status of special keys (Shifts, Locks, and so on). The second byte provides the shift state, which is six bits for Shift, Alt, and Ctrl, as shown in Table 13.2. The third byte is for lock states (three bits for Caps, Num, and Scroll), as shown in Table 13.3. The high-order byte is for the Numeric Pad indicator and is described later.

Shift-State Byte (Second Byte)

If you need to know the shift state of any of the shift keys (Ctrl, Alt, or Shift), you can use the second byte in the 32-bit word returned. Table 13.2 shows how the bits are defined.

Table 13.2. Shift-state bits.

Bit	*Meaning When Set (1)*
0	Left Ctrl key down
1	Right Ctrl key down
2	Left Shift key down
3	Right Shift key down
4	Left Alt key down
5	Right Alt key down
6	Not used (0)
7	Not used (0)

The following masks in assembly language can be used to determine whether the Ctrl, Shift, or Alt keys were pressed for the key code being read.

```
CtrlDownMask  EQU 00000011b

ShftDownMask  EQU 00001100b

AltDownMask   EQU 00110000b
```

Lock-State Byte (Third Byte)

If you need to know the lock state of any of the lock-capable keys (Caps, Num, or Scroll), you can use the third byte in the 32-bit word returned. Table 13.3 shows how the bits are defined:

Table 13.3. Lock-state bits.

Bit	*Meaning When Set (1)*
0	Scroll Lock On
1	Num Lock On
2	Caps Lock On
3	Not used (0)
4	Not used (0)
5	Not used (0)
6	Not used (0)
7	Not used (0)

The following masks in assembly language can be used to determine whether one of the lock keys was active for the key code being read.

```
CpLockMask    EQU 00000100b
NmLockMask    EQU 00000010b
ScLockMask    EQU 00000001b
```

Numeric Pad Indicator

Only one bit is used in the high-order byte of the key code. This bit (Bit 0, LSB) will be set if the keystroke came from the numeric keypad. You need this if you use certain keys from the numeric pad differently than their equivalent keys on the main keyboard. For example, in your program, the Enter key on the numeric keypad might do something differently than the typewriter Enter key.

Key Codes

Table 13.4 shows the hexadecimal value provided in the low-order byte of the key code (least significant) by the keyboard service for each key on a 101-key keyboard. The Shift Code

column shows the value returned if the Shift key is active. The Caps Lock key does not affect the keys shown with an asterisk (*) in the Shift Code column. A description is given only if required.

If a shift code is not shown, the same value is provided in all shifted states (Shift, Ctrl, Alt, or Locks). All codes are 7-bit values (1 to 127—1h to 7Fh). Zero and values above 127 will not be returned.

Table 13.4. Key values.

Base Key	*Desc.*	*Key Code*	*Shifted Key*	*Shift Code*	*Desc*
Esc	Escape	1Bh			
1		31h	!	21h	exclamation
2		32h	@	40h	at
3		33h	#	23h	pound
4		34h	$	24h	dollar
5		35h	%	25h	percent
6		36h	^	5Fh	carat
7		37h	&	26h	ampersand
8		38h	*	2Ah	asterisk
9		39h	(	28h	open paren
0		30h	)	29h	close paren
-	Minus	2Dh	_	5Fh	underscore
=	Equal	3Dh	+	2Bh	
BS	Backspace	08h			
HT	Tab	09h			
a		61h	A	41h	
b		62h	B	42h	
c		63h	C	43h	
d		64h	D	44h	
e		65h	E	45h	
f		66h	F	46h	
g		67h	G	47h	
h		68h	H	48h	
i		69h	I	49h	
j		6Ah	J	4Ah	

continues

Table 13.4. continued

Base Key	*Desc.*	*Key Code*	*Shifted Key*	*Shift Code*	*Desc*
k		6Bh	K	4Bh	
l		6Ch	L	4Ch	
m		6Dh	M	4Dh	
n		6Eh	N	4Eh	
o		6Fh	O	4Fh	
p		70h	P	50h	
q		71h	Q	51h	
r		72h	R	52h	
s		73h	S	53h	
t		74h	T	54h	
u		75h	U	55h	
v		76h	V	56h	
w		77h	W	57h	
x		78h	X	58h	
y		79h	Y	59h	
z		7Ah	Z	5Ah	
[		5Bh	{	7Bh	open brace
\		5Ch	|	7Ch	vert. bar
]		5Dh	}	7Dh	close brace
`	Accent	60h	~	7Eh	tilde
CR	Enter	ODh			
;	Semicolon	3Bh	:	3Ah	colon
'	Apostr.	27h	"	22h	quote
,	Comma	2Ch	<	3Ch	less
.	Period	2Eh	>	3Eh	greater
/	Slash	2Fh	?	3Fh	question
Space		20h			

Function Strip Key Codes

Shift and Lock states do not affect the function key values. Shift and Lock state bytes must be used to determine program functionality. Table 13.5 shows the key code value returned.

Table 13.5. Function key strip codes.

Base Key	*Key Code*
F1	0Fh
F2	10h
F3	11h
F4	12h
F5	13h
F6	14h
F7	15h
F8	16h
F9	17h
F10	18h
F11	19h
F12	1Ah

Numeric Pad Key Codes

The Shift values shown in Table 13.6 are returned when the shift keys or Num Lock are active. Caps Lock does not affect these keys.

Table 13.6. Numeric pad key codes.

Base Key	*Key Code*	*Shift Code*	*Shifted Key*
End	0Bh	31h	1
Down	02h	32h	2
Pg Dn	0Ch	33h	3
Left	03h	34h	4
Blank key	1Fh	35h	5 (Extra Code)
Right	04h	36h	6
Home	06h	37h	7
Up	01h	38h	8
Pg Up	05h	39h	9
Insert	0Eh	30h	0
Delete	7Fh	2Eh	. (Period)

continues

Table 13.6. continued

Base Key	*Key Code*	*Shift Code*	*Shifted Key*
-	Dash	2Dh	
*	Asterisk	2Ah	
/	Slash	2Fh	
+	Plus	2Bh	
CR	Enter	0Dh	

Cursor, Edit, and Special Pad Key Codes

None of the keys in these additional keypads are affected by Shift or Lock states. The values returned as the key code are shown in Table 13.7.

Table 13.7. Additional key codes.

Base Key	*Key Code*
Print Screen	1Ch
Pause	1Dh
Insert	0Eh
Delete	7Fh
Home	06h
End	0Bh
Pg Up	05h
Pg Dn	0Ch
Up	01h
Down	02h
Left	03h
Right	04h

Your Keyboard Implementation

You may not want all of the functionality in your system that I provided, or you may want more, such as international key translations. The keyboard translation tables could be used to nationalize this system, if you desire.

You may also want the keyboard more deeply embedded in your operating system code. Chapter 25, "Keyboard Code," contains the source code behind the system service described in this chapter. Pieces of it may be useful for your system.

I also recommend that you concentrate on the documentation for your keyboard implementation, as I have tried to do in this chapter. Remember, when all else fails, the programmer *will* read the documentation.

14

File System Service

The file system included with MMURTL is compatible with the MS-DOS FAT (File Allocation Table) file system. This means that MMURTL can read and write MS-DOS disks (both hard and floppy). This was done so that you could use MMURTL without reformatting your hard disks or working from floppies. It certainly wasn't done because I liked the design or the filename length limitations. MS-DOS has the unenviable task of being compatible with its previous versions. This ties its hands, so to speak. Better file systems (disk formats) are in use, but none is as widespread.

The internal management of a file system is no trivial matter. Because you are simply accessing one that was designed by someone else, there isn't anything new about the format, just how you access it. I have tried to follow the simplicity motto, and I limited the number of functions to what is necessary for decent operation. The service name is "FILESYS".

File Specifications

A filename consists of one to eight characters, a period, then up to three more characters, for example, *filename.txt*.

A full file specification for the FAT-compatible file system consists of a drive letter identifier followed by a colon, then the path—which is each directory separated by the backslash character—and finally the filename itself. Here's an example:

```
C:\Dir1\Dir2\Dir3\Filename.txt
```

Network File Request Routing

File system requests are routed based on the full file specification. A filename that is prefixed with a node name, and optionally, a network name, will be routed to the service with the same name.

A full network file specification consists of three parts.

The first part is the network name enclosed in *brackets* [ ]. The second part is the network's node name enclosed in *braces* { }. The third part is the filename as previously described. A complete network filename looks like this:

```
[network]{Node}C:\Dir1\Dir2\Filename.txt
```

If the network name is omitted from the specification, and a node name is specified, the default name NETWORK is used.

The network name matches the name of the network service when it was installed, hence the default system service name for a network routing service is *network*. This also means that network names are limited to eight characters. Even though service names must be capitalized and space padded, this is not necessary with network names because the file system fixes them.

When a network file is opened, the network service receives the request from the file system. The file handle returned will be unique on the system that originated the request.

The filename across a network may not be limited to the MS-DOS file naming conventions described here. This depends on the type of file system on the node being accessed.

File Handles

When a file is opened in any mode or type, a number called a *file handle* is returned to you. Make no assumptions about this number. The file handle is used in all subsequent file operations on that file. This number is how you refer to that file until the file is closed. A file handle is a 32-bit unsigned number (dword).

File Open Modes

A file may be opened for reading and writing, or just reading alone. These two modes are called *Modify* (read and write) and *Read* (read-only).

Only a single user may open a file in Modify mode. This user is granted exclusive access to the file while it is open.

Multiple users may open and access a read-mode file. When a file is opened in Read mode, it may not be opened in Modify mode by any user.

File Access Type

A file may be opened in Block or Stream mode.

Block Mode

Block mode operation is the fastest file access method because no internal buffering is required. The data is moved in whole blocks by the fastest means determined by the device driver. No internal file buffers are allocated by the file system in Block mode. The only restriction is that whole blocks must be read or written. The standard block size is 512 bytes for disk devices. Block mode operation allows use of the following file system functions:

```
ReadBlock()
WriteBlock()
CloseFile()
GetFileSize()
SetFileSize()
DeleteFile()
```

Stream Mode

In Stream mode, the file system allocates an internal buffer to use for all read and write operations. This is a one-page, 4096-byte buffer. Stream mode operation allows use of the following file system functions:

```
CloseFile()
ReadBytes()
WriteBytes()
SetFileLFA()
GetFileLFA()
GetFileSize()
SetFileSize()
DeleteFile()
```

Errors are returned from any function not compatible with the file-access type you specified when the file was opened.

Logical File Address

All files are logically stored as 1–*n* bytes. The Logical File Address (LFA) is an unsigned dword (dLFA). This is the byte offset in the file to read or write from. LFA 0 is the beginning of the file. The file size minus 1 is the last logical byte in the file.

Block access file system functions require to you specify a Logical File Address (LFA). Block access files have no internal buffers and do not maintain a current LFA or file pointer.

Stream access files have internal buffers and maintain your current LFA. This is called a file pointer in some systems. For stream files, you do not specify an LFA to read or write from. The current LFA for stream files is updated with each `ReadBytes()` and `WriteBytes()` function. Additional functions are provided to allow you to find and set the current LFA. When a file is initially opened for Stream access, the file pointer is set to 0, no matter what mode the file was opened in

File System Requests

The file system is a message-based system service. This means it can be accessed directly with the Request primitive. The procedural interface for Request has 12 parameters. The file system is an ideal candidate for a message-based service. It meets all the requirements. The small amount of time for message routing will not make any measurable difference in the speed of its operation because most of its time is spent accessing hardware. It also provides the shared access required for a true multitasking system. The file system actually runs as a separate task.

Listing 14.1 is an example of a File System request in the C programming language.

Listing 14.1. `OpenFile` request in C.

```
dError= Request(
"FILESYS ",      /* ptr to name of service */
1,               /* wSvcCode    — 1 for OpenFile */
MyExch,          /* dRespExch   — respond here */
&MyRqhandle,     /* pRqHndlRet — may be needed */
1,               /* nSendPtrs   — 1 Send ptr */
&"AnyFile.doc", /* pData1      — ptr to name  */
11,              /* cbData1     — size of name */
&FileHandle      /* pData2      — returned handle */
4,              /* cbData2     — Size of a handle */
1,              /* dData0      — ModeRead */
0,              /* dData1      — Block Type Access */
0);             /* dData2      — not used */
```

Unused request parameters *must* be set to 0.

All message-based services use the same request-based interface. Using the request interface allows for asynchronous program operation. You can make a request, then go do something else before you come back to wait or check the function to see if it's completed (true multitasking). Each of the functions is identified by its Service Code number. Table 14.1 shows the service codes the file system supports.

Table 14.1. File system service codes.

Function	*Service Code*
`OpenFile`	1
`CloseFile`	2
`ReadBlock`	3
`WriteBlock`	4
`ReadBytes`	5
`WriteBytes`	6
`GetFileLFA`	7
`SetFileLFA`	8
`GetFileSize`	9
`SetFileSize`	10
`CreateFile`	11
`RenameFile`	12
`DeleteFile`	13
`CreateDirectory`	14
`DeleteDirectory`	15
`GetDirectorySector`	16

Procedural Interfaces

If you don't have a need for asynchronous disk access, you can use a blocking procedural interface which is included in the file system itself. The procedural interface actually makes the request for you by using your TSS Exchange. The request is transparent to the caller. It is easier to use because it has fewer parameters. This means there is a simple procedural interface call for each file system function. The blocking procedural interface is described with each of the call descriptions that follow later in this chapter.

Device Access Through the File System

Teletype fashion stream access to the NUL, VID, and KBD devices may be accessed through the file system, but only through the procedural interfaces. The Request interface will not allow device access. High-level language libraries that implement device access (such as `putchar()` in C) must use the procedural interfaces for file access.

All system device names are reserved and should not be used as filenames on the system. Table 14.2 lists the device names that are reserved and also the mode of stream access supported.

Table 14.2. Device stream access.

Device	*Description*	*Access*
NUL	NULL device	Write-only
KBD	Keyboard	Read-only
VID	Video	Write-only
LPT1	Printer 1	None
LPT2	Printer 2	None
COM1	RS-232 1	None
COM2	RS-232 2	None
COM3	RS-232 3	None
COM4	RS-232 4	None
FD0	Floppy disk	None
FD1	Floppy disk	None
HD0	Hard disk	None
HD1	Hard disk	None

Device access is currently implemented for NUL, KBD, and VID devices only.

File System Functions in Detail

The following pages detail each of the file system functions that are available.

OpenFile

Procedural interface:

```
OpenFile (pName, dcbname, dOpenMode, dAccessType, pdHandleRet): dError
```

`OpenFile()` opens an existing file in the current path for block or stream operations. A full file specification may be provided to override the current job path.

Opening in `ModeModify` excludes all others from opening the file. Multiple users can open a file in `ModeRead`. If a file is open in `ModeRead`, it cannot be opened in `ModeModify` by any other users.

Procedural parameters:

`pName`—Pointer to the filename or full file specification

`dcbName`—Dword with length of the filename

`dOpenMode`—READ= 0, MODIFY= 1

`dAccessType`—Block= 0, Stream= 1.

`pdHandleRet`—Pointer to a dword where the handle to the file will be returned to you

Request Parameters for `OpenFile`:

```
wSvcCode    = 1
nSend       = 1
pData1      = pName
cbData1     = dcbName
pData2      = pdHandleRet
cbData2     = 4 (Size of a file handle)
dData0      = dOpenMode
dData1      = dAccessType
dData2      = Not used (0)
```

CloseFile

Procedural interface:

```
CloseFile (dHandle): dError
```

`CloseFile()` closes a file that was previously opened. If stream access type was specified, all buffers are flushed and deallocated.

Procedural parameters:

dHandle—A dword that was returned from OpenFile()

Request Parameters for CloseFile:

```
wSvcCode    = 2
nSend       = 0
pData1      = 0
cbData1     = 0
pData2      = 0
cbData2     = 0
dData0      = dHandle
dData1      = 0
dData2      = 0
```

ReadBlock

Procedural interface:

```
ReadBlock(dHandle, pDataRet, nBytes, dLFA, pdnBytesRet): dError
```

This reads one or more blocks from a file. The file must be opened for Block access or an error occurs.

Procedural parameters:

dHandle—Dword with a valid file handle (as returned from OpenFile()

pDataRet—Pointer to a buffer large enough to hold the count of blocks you specify to read

nBytes—Dword with number of bytes to read. This *must* be a multiple of the block size for the 512-byte disk.

dLFA—Logical File Address to read from. This *must* be a multiple of the block size.

pdnBlkRet—Pointer to a dword where the count of bytes successfully read will be returned. This will always be a multiple of the block size ($n \times 512$).

Request parameters for ReadBlock:

```
wSvcCode    = 3
nSend       = 0
pData1      = pDataRet
cbData1     = nBytes (multiple of 512)
pData2      = pdnBytesRet
cbData2     = 4 (size of dnBytesRet)
dData0      = dHandle
dData1      = 0
dData2      = 0
```

WriteBlock

Procedural interface:

```
WriteBlock(dHandle, pData, nBytes, dLFA, pdnBytesRet): dError
```

This writes one or more blocks to a file. The file must be opened for Block access in Modify mode or an error occurs. Writing beyond the current file length is *not* allowed. `SetFileSize()` must be used to extend the file length if you intend to write beyond the current file size. See the `SetFileSize()` section.

Procedural parameters:

`dHandle`—Dword with a valid file handle as returned from `OpenFile()`.

`pData`—Pointer to the data to write

`nBytes`—Dword with number of bytes to write. This must always be a multiple of 512. One block = 512 bytes.

`dLFA`—Logical File Address to write to. This *must* be a multiple of the block size.

`pdnBytesRet`—Pointer to a dword where the number of bytes successfully written will be returned. This will always return a multiple of 512.

Request parameters for ReadBlock:

`wSvcCode` = 4
`nSend` = 1
`pData1` = `pData`
`cbData1` = `nBytes` (512= 1 Block)
`pData2` = `pdnBytesRet`
`cbData2` = 4 (size of `dnBytesRet`)
`dData0` = `dHandle`
`dData1` = 0
`dData2` = 0

ReadBytes

Procedural interface:

```
ReadBytes(dHandle, pDataRet, nBytes, pdnBytesRet): dError
```

This reads one or more bytes from a file. The file must be opened for Stream access or an error occurs. The bytes are read from the current LFA. The LFA is updated to the next byte address following the data you read. Use `SetFileLFA()` to move the stream file pointer if you want to read from an LFA other than the current LFA. `GetFileLFA` may be used to find the current LFA.

Parameters:

dHandle—Dword with a valid file handle, as returned from OpenFile()

pDataRet—Pointer to a buffer large enough to hold the count of bytes you specify to read

nBytes—Dword with number of bytes to read

pdnBytesRet—Pointer to a dword where the number of bytes successfully read will be returned

Request Parameters for ReadBytes:

wSvcCode = 5
nSend = 0
pData1 = pDataRet
cbData1 = nBytes
pData2 = pdnBytesRet
cbData2 = 4 (size of dnBytesRet)
dData0 = dHandle
dData1 = 0
dData2 = 0

WriteBytes

Procedural interface:

```
WriteBytes(dHandle, pData, nBytes, pdnBytesRet): dError
```

This writes one or more bytes to a file. The file must be opened for stream access in Modify mode or an error occurs. The bytes are written beginning at the current LFA. The LFA is updated to the next byte address following the data you wrote. The file length is extended automatically if you write past the current End Of File. Use SetFileLFA() to move the stream file pointer if you want to read from an LFA other than the current LFA. GetFileLFA() may be used to find the current LFA.

Procedural parameters:

dHandle—Dword with a valid file handle, as returned from OpenFile

pData—Pointer to the data to write

nBytes—Dword with number of bytes to write

pdnBytesRet—Pointer to a dword where the number of bytes successfully written will be returned

Request parameters for `ReadBytes`:

`wSvcCode`	= 6
`nSend`	= 1
`pData1`	= `pData`
`cbData1`	= `nBytes`
`pData2`	= `pdnBytesRet`
`cbData2`	= 4 (size of `dnBytesRet`)
`dData0`	= `dHandle`
`dData1`	= 0
`dData2`	= 0

GetFileLFA

Procedural Interface:

```
GetFileLFA(dHandle, pdLFARet): dError
```

This gets the current LFA for files with stream mode access. An error occurs if the file is opened for Block access.

Procedural parameters:

`dHandle`—Dword with a valid file handle, as returned from `OpenFile`

`pdLFARet`—Pointer to a dword where the current LFA will be returned

Request parameters for `GetFileLFA`:

`wSvcCode`	= 7
`nSend`	= 0
`pData1`	= `pdLFARet`
`cbData1`	= 4
`pData2`	= 0
`cbData2`	= 0
`dData0`	= `dHandle`
`dData1`	= 0
`dData2`	= 0

SetFileLFA

Procedural interface:

```
SetFileLFA(dHandle, dLFA): dError
```

This sets the current LFA (file pointer) for Stream Access files. An error occurs if the file is opened for Block access. The file LFA can not be set past the End Of File (EOF). The file will be set to EOF if you specify 0FFFFFFFF hex (-1 for a signed long value in C).

Procedural parameters:

dHandle—Dword with a valid file handle, as returned from OpenFile().

dLFA—Dword with the LFA you want to set. 0FFFFFFFF hex will set the current LFA to End Of File.

Request parameters for SetFileLFA:

```
wSvcCode    = 8
nSend       = 0
pData1      = 0
cbData1     = 0
pData2      = 0
cbData2     = 0
dData0      = dHandle
dData1      = dLFA
dData2      = 0
```

GetFileSize

Procedural interface:

```
GetFileSize(dHandle, pdSizeRet): dError
```

This gets the current file size for files opened in any access mode.

Procedural parameters:

dhandle—Dword with a valid file handle, as returned from OpenFile()

pdSizeRet—Pointer to a dword where the current size of the file will be returned

Request parameters for GetFileLFA:

```
wSvcCode    = 9
nSend       = 0
pData1      = pdSizeRet
cbData1     = 4
pData2      = 0
cbData2     = 0
dData0      = dHandle
dData1      = 0
dData2      = 0
```

SetFileSize

Procedural interface:

```
SetFileSize(dHandle, dSize): dError
```

This sets the current file size for Stream and Block Access files. The file length, and allocated space on the disk, will be extended or truncated as necessary to accommodate the size you specify. The current LFA is not affected for Stream files unless it is now past the new EOF, in which case it is set to EOF automatically.

Procedural parameters:

`dHandle`—Dword with a valid file handle as returned from `OpenFile()`

`dSize`—Dword with the new size of the file

Request Parameters for `SetFileSize`:

```
wSvcCode    = 10
nSend       = 0
pData1      = 0
cbData1     = 0
pData2      = 0
cbData2     = 0
dData0      = dHandle
dData1      = dSize
dData2      = 0
```

CreateFile

Procedural interface:

```
CreateFile (pName, dcbName, dAttributes): dError
```

This creates a new, empty file in the path specified. The file is closed after creation, and the file is ready to be opened and accessed.

Procedural parameters:

`pName`—Pointer to the filename or full file specification to create

`dcbName`—Dword with length of the filename

`dAttributes`—Dword with one of the following values:

0 = Normal File
2 = Hidden File
4 = System File
6 = Hidden System file

Request parameters for `CreateFile`:

```
wSvcCode   = 11
nSend      = 1
pData1     = pName
cbData1    = dcbName
pData2     = 0
cbData2    = 0
dData0     = dAttributes
dData1     = 0
dData2     = 0
```

RenameFile

Procedural interface:

```
RenameFile (pName, dcbname, pNewName dcbNewName): dError
```

This renames a file. The file must not be opened by anyone in any mode.

Procedural parameters:

`pName`—Pointer to the current filename or full file specification

`dcbName`—Dword with length of the current filename

`pNewName`—Pointer to the new filename or full file specification

`dcbNewName`—Dword with length of the new filename

Request parameters for `RenameFile`:

```
wSvcCode   = 12
nSend      = 1
pData1     = pName
cbData1    = dcbName
pData2     = pNewname
cbData2    = dcbNewname
dData0     = 0
dData1     = 0
dData2     = 0
```

DeleteFile

Procedural interface:

```
DeleteFile (dHandle): dError
```

This deletes a file from the system. No further access is possible, and the filename is available for re-use.

The file must be opened in Modify mode, which grants exclusive access.

Procedural parameters:

`dHandle`—Dword that was returned from `OpenFile`

Request parameters for `DeleteFile`:

```
wSvcCode    = 13
nSend       = 0
pData1      = 0
cbData1     = 0
pData2      = 0
cbData2     = 0
dData0      = dHandle
dData1      = 0
dData2      = 0
```

CreateDirectory

Procedural interface:

```
CreateDirectory (pPath, dcbPath): dError
```

This creates a new, empty directory for the path specified. The path must contain the new directory name as the last element. This means all subdirectories above the last directory specified must already exist. If only the new directory name is specified, as opposed to a full path, the directory is created in your current path, specified in the Job Control Block.

Procedural parameters:

`pPath`—Pointer to the path (directory name) to create

`dcbPath`—Dword with length of the path

Request parameters for `CreateDirectory`:

```
wSvcCode    = 14
nSend       = 1
pData1      = pPath
cbData1     = dcbPath
pData2      = 0
cbData2     = 0
dData0      = 0
dData1      = 0
dData2      = 0
```

DeleteDirectory

Procedural interface:

```
DeleteDirectory (pPath, dcbPath, fAllFiles): dError
```

This deletes a directory for the path specified. The path must contain the directory name as the last element. If only the existing directory name is specified, as opposed to a full path, the name is appended to the path from your job control block.

All files and subdirectories will be deleted if `fAllFiles` is *non-zero.*

Procedural parameters:

`pPath`—Pointer to the path, or directory name, to delete

`dcbPath`—Dword with length of the path

`fAllFiles`—If set to *non-zero,* all files in this directory, all subdirectories, and files in subdirectories will be deleted.

Request parameters for `DeleteDirectory`:

```
wSvcCode    = 15
nSend       = 1
pData1      = pPath
cbData1     = dcbPath
pData2      = 0
cbData2     = 0
dData0      = 0
dData1      = 0
dData2      = 0
```

GetDirSector

Procedural Interface:

```
GetDirSector (pPath, dcbPath, pSectRet, dSectNum): dError
```

Directories on the disk are one or more sectors in size. Each sector contains 16 directory entries. This call returns one sector from a directory.

You specify which logical directory sector you want to read (from 0 to `nTotalSectors-1`). The directory must exist in the path specified. If no path is specified, the path in your JCB will be used.

A single directory entry looks like this with 16 per sector:

```
Name        8 Bytes
Ext         3 Bytes
```

Attribute	1 Byte
Reserved	10 Bytes
Time	2 Bytes
Date	2 Bytes
StartClstr	2 Bytes
FileSize	4 Bytes

The fields are read and used exactly as MS-DOS uses them.

Procedural parameters:

pPath—Pointer to the path, or directory name, to read the sector from

dcbPath—Dword with length of the path

pSectRet—Pointer to a 512-byte block where the directory sector will be returned

dSectNum—Dword with the logical sector number to return

Request parameters for GetDirSector:

wSvcCode	= 16
nSend	= 1
pData1	= pPath
cbData1	= dcbPath
pData2	= pSectRet
cbData2	= 512
dData0	= dSectNum
dData1	= 0
dData2	= 0

File System Theory

Understanding a bit more about how the file system is managed may help you make better use of it.

Internal File System Structures

The file system has several internal structures it uses to manage the logical disks and open files. Some structures are dynamically allocated, while others are part of the operating system data segment.

File Control Blocks

All file system implementations have some structure that is used to manage an open file. Quite a few of them use this same name, FCB.

The file system allocates file system structures dynamically when it is initialized. The maximum number of open files is currently limited to 128 files by this initial allocation.

The file system starts with an initial 4K (one memory page) of FCBs. Each FCB is 128 bytes. Each open file is allocated an FCB. If the file is opened in Stream mode, you allocate a one-page buffer (4K) for reading and writing.

The FCB actually contains a copy of the 32-byte directory entry from the disk. This is read into the FCB when the file is opened.

File User Blocks

Because this is a multitasking operating system, the FCB is not directly associated with the job that opened the file. Another structure, called the file user block (FUB), is what links the user of a file to the FCB. There is one FUB for each instance of a user with an open file.

The FUBs contain the file pointer, the open mode, buffer pointers, and other things that are required to manage the user's link to the file. The file handle is actually a pointer to the FUB. Because it is a pointer, extra checking is done to ensure it is valid in every call. A bad pointer could do some real damage here if you let it.

The FCB keeps track of the number of FUBs for the file. When it reaches 0, the FCB is deallocated for re-use.

FAT Buffers

In the FAT file system, the location of every cluster of one or more sectors of a file is managed in a linked list of sorts. This linked list is contained in a table called the File Allocation Table. A more accurate name would be cluster allocation table because it's actually allocating clusters of disk space to the files, it's not allocating files.

The disk is allocated in clusters, with each cluster having an associated entry in the FAT. On most hard disks, they use what is called a FAT16. This means each FAT entry is a 16-bit word. The beginning of the FAT represents the beginning of the usable data area on the logical disk.

Let's look at an example of how you use the FAT to locate all parts of file on the disk. In this example, you'll assume that each FAT entry is worth 4K on disk, which is cluster size, and you'll assume the file size is 12K, or three entries.

Assume the directory entry you read says the first FAT entry is number 4.
X= not your file. L= Last entry in your file.

FAT Position	1	2	3	4	5	6	7	8	9	10
Entry in FAT	X	X	X	5	6	L				

This means your file occupies the fourth, fifth, and sixth clusters in the data area on the disk. The fourth entry points to the fifth, the fifth points to the sixth, and the sixth is marked as the last. Have you ever gotten *lost clusters* when you ran a `Chkdsk` in MS-DOS? Who hasn't? It usually means it found some clusters pointing to other clusters that don't end, or it found a beginning cluster with no corresponding directory entry pointing to it.

From this example, you can see that every file you access will require a minimum of two disk accesses to read the file, *unless* the piece of the FAT you need is already in memory.

This means it's important to attempt to keep as much FAT information in memory as you can and manage it wisely. If your disk isn't too badly fragmented and if you are sequentially reading all of a file into memory (such as when loading an executable file), you can usually do it quite rapidly. Just 1K of FAT space represents 2MB of data space on the disk if clusters are 4K. A badly fragmented disk can force as many as four or five accesses to different parts of the FAT just to read one file.

The disk controller (IDE/MFM) can read 256 contiguous sectors in one read operation. That's 128K. That's a lot of FAT (or file). However, because of the design of the FAT file system, the actual contiguous read will usually be limited to the size of a cluster.

A great number of files are read and written sequentially. These include, run files being executed, source files for compilers, the initial filling of buffers for a word processor (and the final save usually), vector image files, and raster image files. The list could go on and on. Large database files are the major random users, and only a good level-4 mindreader knows where, inside of a 400MB file, the user will strike next.

MMURTL's DOS FAT file system allocates a 1K FAT buffer, based on file access patterns. A simple LRU (least recently used) algorithm ensures the most accessed FAT sectors will remain in memory.

File Operations

Reading a file is a multipart process. Look at what the file system goes through to open, read, and close a file:

1. Look through the FCBs to see if someone already has it open in `ModeRead`. If so, allocate an FUB and attach it. If it's open in `ModeWrite`, return an error `ErcFileInUse`.
2. Locate the directory entry and read it into the FCB. Directories are stored as files. This means that you have to go to the FAT to find where the directory is stored so you can look through it. This is in itself time-consuming.
3. Load FAT buffer if required. The first cluster of the file is listed in the directory entry. From this, you calculate which disk sector in the FAT has this entry, and you read it into the FAT buffer if it is not already in one. The file is open.

Read

What actually happens on a read depends on whether the file is opened in Block or Stream mode. If opened in Block mode, you simply issue the read to the device driver to read the sectors into the caller's buffer (locating them using the FAT). The hardware is limited to 128K reads in one shot, and can only do this if 128K of the file is located in contiguous sectors. Usually, the commands to the device driver will be broken into several smaller reads as the file system finds each contiguous section listed in the FAT.

Write

Writing to a file is a little more complicated, and depends on how the file was opened. If the file is opened in Block mode, you issue the write command to the device driver to write out the sectors as indicated by the user. It sounds simple, but much overhead is involved. If the sectors have already been allocated in the FAT, you simply write over the existing data while you follow the cluster chain. If the file is being extended, you must find unallocated clusters, attach them to the chain, then write the data. On many disks there are two copies of the FAT for integrity purposes. Both copies must be updated.

Close

This is the easy part. You flush buffers to disk if necessary for Write mode. Then you deallocate the FUB and FCB.

15

API Specification

This section describes all operating system public calls. They are listed by functional group first; then an alphabetical list provides details for each call and possibly an example of its use.

If you are going to write your own operating system, you'll find the documentation to be almost 30 percent of the work—and it's a very important part of the entire system, I might add. An improperly or poorly described function call can cause serious agony and hair loss in a programmer.

Function calls provided by system services (accessed with Request and Respond interface) are described in detail in the chapters of the book that apply to that service (Chapter 14, "File System Service," for example).

Public Calls

Public calls are those accessible through a call gate and can be reached from "outside" programs. They are defined as *far* and require a 48-bit call address. The call address consists of a selector and an offset. The selector is the call gate entry in the global descriptor table (GDT), while the offset is ignored by the 386/486 processor. The offset should be 0. The descriptor for the call gate defines the actual address called and the number of dwords of stack parameters to be passed to the operating system call.

Parameters to Calls (Args)

All stack parameters are described by their names. The prefixes indicate the size and type of the data that is expected. Table 15.1 lists the prefixes.

Table 15.1. Size and type prefixes.

Prefix	*Description*
`b`	Byte (1 byte unsigned)
`w`	Word (2 bytes unsigned)
`d`	Double Word (dword—4 bytes unsigned)
	`d` is the default when `b` or `w` is not present.
`ib`	Integer byte (1 byte signed)
`iw`	Integer word (2 bytes signed)
`id`	Integer dword (4 bytes signed)
`p`	Pointer (4 bytes—32-bit near pointer)

Additional descriptive prefixes may be used to help describe the parameter. These are known as compound descriptions. Table 15.2 lists these additional prefix characters.

Table 15.2. Additional prefixes.

Prefix	*Description*
`a`	Array (of)
`c`	Count (of)
`s`	Size (of)
`o`	Offset (into)
`n`	Number (of)

Examples of compound prefixes are shown in Table 15.3.

Table 15.3. Compound prefixes.

Prefix	*Description*
`pab`	Pointer to an array of bytes
`pad`	Pointer to an array of double words
`pdcb`	Pointer to a dword that is a count of bytes
`pib`	Pointer to a signed (integer) byte
`pb`	Pointer to a byte
`cb`	Dword that contains a count of bytes (defaults to size `d` if not preceded by another size prefix)
`n`	Dword that is a number of something
`ppd`	Pointer to a pointer that points to a double word

Descriptive suffixes may also be used to help describe some parameters. *Ret* means Returned, *Max* means Maximum, and *Min* means Minimum.

Each compound set of prefixes—and optionally, suffixes—accurately describes the provided or expected data types. Combine these prefixes with variable or parameter names and you have a complete description of the data. The following are examples of parameter names using prefixes and suffixes.

`dcbFileName`—Dword that contains the count of bytes of a filename

`pdDataRet`—Pointer to a dword where data is returned

`dcbDataRetMax`—Dword with the maximum count of bytes that can, or should be, returned

Stack Conventions

MMURTL keeps a dword (4 bytes) aligned stack. It is processor (hardware) enforced. Bytes and words (8- or 16-bit values) are pushed as dwords automatically. The caller need not worry about whether the upper part of a dword on the stack is cleared or set when a value is pushed, because MMURTL will only use as much of it as is defined by the call. If a parameter is a byte, MMURTL procedures will only get the byte from the stack, while the upper three bytes of the dword are ignored. MMURTL high-level languages follow this convention. This information is provided for the assembly language programmers.

Calls Grouped by Functionality

The following is a list of all currently implemented operating system public calls, grouped by type. The parameter (argument) names follow the naming convention described earlier in this chapter. Many of these calls will not be used by the applications programmer.

The operating-system calls are described in a generic fashion that does not directly correspond to a particular programming language such as C, Pascal, or Assembler. The function name is followed by the parameters, and finally, if a value is returned by the function, a colon with the returned value. The `dError`, the most common returned value shown, indicates a dword error or status code is returned from the function.

Messaging

```
AllocExch(pdExchRet): dError

DeAllocExch(dExch): dError

SendMsg (dExch, dMsg1, dMsg2): dError

CheckMsg(dExch, pMsgsRet): dError

ISendMsg (dExch, dMsg1, dMsg2): dError

WaitMsg(dExch,pqMsgRet): dError

Request(pSvcName, wSvcCode, dRespExch, pRqHndlRet, npSend, pData1, cbData1,
pData2, cbData2, dData0, dData1, dData2 ) : dError

MoveRequest(dRqBlkHndl, dDestExch): dError

Respond(dRqHndl, dStatRet) : dError
```

Job Management

```
Chain(pFileName, dcbFileName, dExitError): dError

ExitJob(dExitError): dError

LoadNewJob(pFileName, dcbFileName, pdJobNumRet): dError

GetpJCB(dJobNum, pJCBRet): dError
```

```
GetJobNum(pdJobNumRet): dError
SetJobName(pJobName, dcbJobName): dError
SetExitJob(pFileName, dcbFileName): dError
GetExitJob(pFileNameRet, pcbFileNameRet): dError
SetCmdLine(pCmdLine, dcbCmdLine): dError
GetCmdLine(pCmdLineRet, pdcbCmdLineRet): dError
SetPath(pPath, dcbPath): dError
GetPath(dJobNum, pPathRet, pdcbPathRet): dError
SetUserName(pUserName, dcbUserName): dError
GetUserName(pUserNameRet, pdcbUserNameRet): dError
SetSysIn(pFileName, dcbFileName): dError
GetSysIn(pFileNameRet, pdcbFileNameRet): dError
SetSysOut(pFileName, dcbFileName): dError
GetSysOut(pFileNameRet, pdcbFileNameRet): dError
```

Task Management

```
NewTask(dJobNum, wCodeSeg, dPriority, dfDebug, dExch, dESP, dEIP): dError
SpawnTask(pEntry, dPriority, fDebug, pStack, fOSCode):dError
SetPriority(bPriority): dError
GetTSSExch(pdExchRet): dError
```

System Service and Device Driver Management

```
DeviceOp(dDevice, dOpNum, dLBA, dnBlocks, pData): dError
DeviceStat(dDevice, pStatRet, dStatusMax, dStatusRet):dError
DeviceInit(dDevice, pInitData, dInitData): dError
InitDevDr(dDevNum, pDCBs, nDevices, fReplace): dError
RegisterSvc(pSvcName, dExch): dError
UnRegisterSvc(pSvcName): dError
```

Memory Management

```
AllocPage(nPages, ppMemRet): dError
AllocDMAPage(nPages, ppMemRet, pdPhyMemRet): dError
AllocOSPage(nPages, ppMemRet): dError
DeAllocPage(pMem, dnPages): dError
AliasMem(pMem, dcbMem, dJobNum, ppAliasRet): dError
```

```
DeAliasMem(pAliasMem, dcbAliasBytes, JobNum): dError
GetPhyAdd(dJobNum, LinAdd, pPhyAddRet): dError
QueryPages(pdnPagesRet): dError
```

High-Speed Data Movement

```
FillData(pDest, cBytes, bFill)
CompareNCS(pS1, pS2, dSize): returned offset or -1
Compare(pS1, pS2, dSize): returned offset or -1
CopyData(pSource, pDestination, dBytes)
CopyDataR(pSource, pDestination, dBytes)
```

Timer and Timing

```
Alarm(dExchRet, dTicks): dError
KillAlarm (dAlarmExch): dError
Sleep(nTicks): dError
MicroDelay (dn15us): dError
GetCMOSTime(pdTimeRet): dError
GetCMOSDate(pdDateRet): dError
GetTimerTick(pdTickRet): dError
```

ISA Hardware Specific I/O

```
DmaSetUp(dPhyMem, sdMem, dChannel, fdRead, dMode): dError
GetDMACount(dChannel, pwCountRet): dError
InByte(dPort): Byte
InWord(dPort): Word
InDWord(dPort): DWord
InWords(dPort, pDataIn,dBytes)
OutByte(Byte, dPort)
OutWord(Word,dPort)
OutDWord(DWord, dPort)
OutWords(dPort, pDataOut, dBytes)
ReadCMOS(bAddress):Byte
```

Interrupt and Call Gate Management

```
AddCallGate: dError
AddIDTGate: dError
```

```
SetIRQVector(dIRQnum, pVector)
GetIRQVector (dIRQnum, pVectorRet): dError
MaskIRQ (dIRQnum): dError
UnMaskIQR (dIRQnum): dError
EndOfIRQ (dIRQnum)
```

Keyboard, Speaker, and Video

```
SetVidOwner(dJobNum): dError
GetVidOwner(pdJobNumRet): dError
GetNormVid(pdNormVidRet): dError
SetNormVid(dNormVidAttr): dError
Beep(): dError
ClrScr(): dError
GetXY(pdXRet, pdYRet): dError
SetXY(dNewX, dNewY): dError
PutVidChars(dCol, dLine, pbChars, nChars, dAttrib): dError
GetVidChar(dCol, ddLine, pbCharRet, pbAttrRet): dError
PutVidAttrs(dCol, dLine, nChars, dAttr): dError
ScrollVid(dULCol, dULline, nCols, nLines, dfUp): dError
ReadKbd (pdKeyCodeRet, dfWait): dError
Tone(dHz, dTicks10ms): dError
TTYOut (pTextOut, dTextOut, dAttrib): dError
EditLine( pStr, dCrntLen, dMaxLen, pdLenRet,
pbExitChar, dEditAttr): dError
```

Alphabetical Call Listing

The remaining sections of this chapter describe each of the operating system calls available to programmers.

AddCallGate

```
AddCallGate: dError
```

`AddCallGate()` makes an entry in the global descriptor table (GDT) for a publicly available operating-system function. `AddCallGate()` is primarily for operating system use, but can be used by experienced systems programmers.

This call doesn't check to see if the GDT descriptor for the call is already defined. It assumes you know what you are doing and overwrites the descriptor if it is already there. The selector number is checked to make sure you're in range (40h through the maximum call gate number). Unlike most other operating system calls, the parameters are *not* stack-based. They are passed in registers. `AddCallGate()` can only be executed by code running at supervisor level (0).

Parameters:

`AX`—Word with call gate ID type as follows:

DPL entry of 3 EC0x (most likely)

DPL entry of 2 CC0x

DPL entry of 1 AC0x

DPL entry of 0 8C0x

(x = count of dword parameters 0–F)

`CX`—Selector number for call gate in GDT (these are constants)

`ESI`—Offset of entry point in segment of code to execute

`EAX`—Returns an error, or 0 if all went well.

AddIDTGate

```
AddIDTGate: dError
```

`AddIDTGate()` makes an entry in the Interrupt Descriptor Table for traps, exceptions, and interrupts.

`AddIDTGate()` is primarily for operating system use, but can be used by experienced systems programmers.

`AddIDTGate()` builds and adds an IDT entry for trap gates, exception gates and interrupt gates. This call doesn't check to see if the IDT descriptor for the call is already defined. It assumes you know what you are doing and overwrites the descriptor if already defined. Unlike most other operating system calls, the parameters are *not* stack based. They are passed in registers. `AddIDTGate()` can only be executed by code running at supervisor level (0).

The selector of the call is always 8 (operating-system code segment) for interrupt or trap, and is the TSS descriptor number of the task for a task gate.

Parameters:

`AX`—Word with Gate ID type as follows:

Trap Gate with DPL of 3	8F00h
Interrupt Gate with DPL of 3	8E00h
Task Gate with DPL of 3	8500h

BX—Selector of gate (08 or TSS selector for task gates)

CX—Word with Interrupt Number (00–FF)

ESI—Offset of entry point in operating system code to execute. This must be zero (0) for task gates.

EAX—Returns Error, else 0 if all went OK

Alarm

```
Alarm(dExchRet, dnTicks): dError
```

`Alarm()` is a public routine that will send a message to an exchange after a specified period of 10-millisecond increments has elapsed. `Alarm()` can be used for a number of things. One important use would be a time-out function for device drivers that wait for interrupts but may never get them due to hardware difficulties. `Alarm()` is similar to `Sleep()`, except it can be used asynchronously. This means you can set an alarm, do some other processing, and then "`CheckMsg()`" on the exchange to see if it has gone off. In fact, you can set it to send a message to the same exchange where you are expecting another message, then see which one gets there first by simply waiting at the exchange. This would be for the time-out function described above. `Alarm()` should not be called for extremely critical timing in applications. The overhead included in entry, exit and messaging code makes the alarm delay timing a little more than 10 milliseconds and is not precisely calculable, even though it's very close. The larger `dnTicks` is, the more precise the timing. `Alarm()` is accomplished using the same timer blocks as `Sleep()`, except you provide the exchange, and don't get placed in a waiting condition. The message that is sent is two dwords, each containing 0FFFFFFFF hex (-1).

Parameters:

`dnTicks`—Dword with the number of 10-millisecond periods to wait before sending a message the specified exchange

AliasMem

```
AliasMem(pMem, dcbMem, dJobNum, ppAliasRet): dError
```

`AliasMem()` provides an address conversion between application addresses. Each application has its own linear address space. Even addresses of identical numeric values are not the same addresses when shared between applications. This procedure is used by the operating system to perform address conversions when passing data using the `Request()` and `Respond()` primitives.

Parameters:

`pMem`—Pointer to the memory address from the other application's space that needs to be aliased

dcbMem—The count of bytes pMem is pointing to. You must not attempt to access more than this number of bytes from the other application's space. If you do, you will cause a fault and your application will be terminated.

dJobNum—Job number from the other application

ppAliasRet—Pointer to the pointer you want to use to access the other application's memory. (The address of the aliased memory pointer you will use).

AllocDMAPage

```
AllocDMAPage(nPages, ppMemRet, pdPhyMemRet): dError
```

AllocDMAPage() allocates one or more pages of memory and adds the entry(s) to the caller's page tables. Memory is guaranteed to be contiguous and will be within reach of the DMA hardware. It will always begin on a page boundary. No cleanup is done on the caller's memory space. If the caller continuously allocates pages and then deallocates them, this could lead to fragmentation of the linear memory space. The allocation routine uses a first-fit algorithm.

Parameters:

nPages—Dword (4 bytes). This is the count of 4K pages to allocate.

ppRet—Pointer to 4-byte area where the pointer to the memory is returned

pdPhyMemRet—A pointer to 4-byte area where the physical address of the memory is returned. This pointer is only valid when used with the DMA hardware. Using it as a normal pointer will surely cause horrible results.

AllocExch

```
AllocExch(pdExchRet): dError
```

The kernel Allocate Exchange primitive. This procedure allocates an exchange (message port) for the job to use. Exchanges are required in order to use the operating system messaging system (SendMsg, WaitMsg, and so on).

Parameters:

pdExchRet—Pointer to the dword where the exchange number is returned

AllocOSPage

```
AllocOSPage(nPages, ppMemRet): dError
```

Allocate operating system page allocates one or more pages of memory and adds the entry to the operating system page tables. Memory is guaranteed to be contiguous and to always begin on a page boundary. No cleanup is done on the caller's memory space. If the caller continuously allocates pages and then deallocates them this could lead to fragmentation of the linear memory space. The allocation routine uses a first-fit algorithm.

Parameters:

`nPages`—A dword (4-bytes). This is the count of 4K pages to allocate.

`ppRet`—A pointer to 4-byte area where the pointer to the memory is returned.

AllocPage

```
AllocPage(nPages, ppMemRet): dError
```

Allocate page allocates one or more pages of memory and adds the entry to the caller's page tables in upper linear memory. Memory is guaranteed to be contiguous and will always begin on a page boundary. No cleanup is done on the caller's memory space. If the caller continuously allocates pages and then deallocates them this could lead to fragmentation of the linear memory space. The allocation routine uses a first-fit algorithm.

Parameters:

`nPages`—A dword (4 bytes). This is the count of 4K pages to allocate.

`ppRet`—A pointer to 4-byte area where the pointer to the memory is returned.

Beep

```
Beep: dError
```

This is a public routine that sounds a 250-millisecond tone at 800Hz.

Parameters: None

Chain

```
Chain(pFileName, dcbFileName, dExitError): dError
```

`Chain()` is called by applications and services to replace themselves with another application (as specified by the filename). `Chain()` terminates all tasks belonging to the calling job and frees system resources that will not be used in the new job. If chain cannot open the new job or the new run file is corrupted, `ErcBadRunFile`, or similar errors, may be returned to the application and the chain operation will be canceled. If the chain is too far along, and running the new application fails, the `ExitJob`, if the JCB contained a valid one, will be loaded and run. If there is not an exit job specified, the job will be terminated as described in the `ExitJob()` call.

Parameters:

`pFilename`—This points to the name of a valid executable run file.

`dcbRunFilename`— A dword containing the length of the run file name.

`dExitError`—A dword containing an error code to place in the `ErrorExit` field of the JCB. The chained application may or may not use this error value.

CheckMsg

```
CheckMsg(dExch, pMsgsRet) : dError
```

The kernel `CheckMsg()` primitive allows a task to receive information from another task *without blocking* the caller. In other words, if no message is available, `CheckMsg()` returns with an error to the caller. If a message *is* available, the message is returned to the caller immediately. The caller is never placed on an exchange and the Task Ready queue is not evaluated. This call is used to check for non-specific messages and responses from services.

The first dword's value (`dMsgHi`) determines if it's a non-specific message or a response from a service. If the value is less than 80000000h (a positive-signed long integer in C) it is a response from a service, otherwise it is a non-specific message that was sent to this exchange by `SendMsg()` or `ISendMsg()`. If it was a response from a service, the second dword is the status code from the service.

NOTE

The value of the first dword is an agreed-upon convention. If you allocate an exchange that is only used for messaging between two of your own tasks within the same job, then the two dwords of the message can represent any two values you like, including pointers to your own memory area. If you intend to use the exchange for both messages and responses to requests, you should use the numbering convention. The operating system follows this convention with the `Alarm()` function by sending 0FFFFFFFFh as the first and second dwords of the message.

Parameters:

`dExch`—The exchange to check for a waiting message.

`pMsgsRet`—A pointer to two dwords in contiguous memory where the two dword messages should be returned if a message is waiting at this exchange.

`dError`—Returns 0 if a message was waiting, and has been placed in `qMsg`, else `ErcNoMsg` is returned.

ClrScr

```
ClrScr()
```

This clears the virtual screen for the current job. It may or may not be the one you are viewing (active screen).

Parameters: None

Compare

```
Compare(pS1, pS2, dSize) : returned offset or -1
```

This is a high-speed string compare function using the Intel string instructions. This version is ASCII case sensitive. It returns the offset of the first byte that doesn't match between the `pS1` and `pS2`, or it returns `-1` (0xFFFFFFFF) if all bytes match out to `dSize`.

`pS1` and `pS2`—Pointers to the data strings to compare.

`dSize`—Number of bytes to compare in `pS1` and `pS2`.

CompareNCS

```
CompareNCS(pS1, pS2, dSize) : returned offset or -1
```

This is a high-speed string compare function using the Intel string instructions. This version is *not case sensitive* (NCS). It returns the offset of the first byte that doesn't match between the `pS1` and `pS2`, or it returns `-1` (0xFFFFFFFF) if all bytes match (ignoring case) out to `dSize`.

Parameters:

`pS1` and `pS2`—Pointers to the data strings to compare.

`dSize`—Number of bytes to compare in `pS1` and `pS2`.

CopyData

```
CopyData(pSource, pDestination, dBytes)
```

This is a high-speed string move function using the Intel string move intrinsic instructions. Data is always moved as dwords if possible.

Parameters:

`pSource`— Pointer to the data you want moved.

`pDestination`—Pointer to where you want the data moved.

`dBytes`—Count of bytes to move.

CopyDataR

```
CopyDataR(pSource, pDestination, dBytes)
```

This is a high-speed string move function using the Intel string move intrinsic instructions. This version should be called if the `pSource` and `pDest` address overlap. This moves the data from the highest address down so as not to overwrite `pSource`.

Parameters:

`pSource`—Pointer to the data you want moved.

pDestination—Pointer to where you want the data moved.

dBytes—Count of bytes to move.

DeAliasMem

```
DeAliasMem(pAliasMem, dcbAliasBytes, dJobNum): dError
```

This invalidates the an aliased pointer that you created using the `AliasMem()` call. This frees up pages in your linear address space. You should use this call for each aliased pointer when you no longer need them.

Parameters:

pAliasMem—The linear address you want to dealias. This should be the same value that was returned to you in the `AliasMem()` call.

dcbAliasBytes—Size of the memory area to dealias. This must be the same value you used when you made the `AliasMem()` call.

dJobNum—This is the job number of the application whose memory you had aliased. This should be the same job number you provided for the `AliasMem()` call.

DeAllocExch

```
DeAllocExch(dExch): dError
```

Deallocate exchange releases the exchange back to the operating system for reuse. If tasks or messages are waiting at the exchange they are released and returned as reusable system resources.

Parameters:

dExch—The exchange number that is being released.

DeAllocPage

```
DeAllocPage(pMem, dnPages) : dError
```

Deallocate page deletes the memory from the job's page table. This call works for any memory, OS and DMA included. Access to this memory after this call will cause a page fault and terminate the job.

Parameters:

pMem—Pointer which should contain a valid memory address, rounded to a page boundary, for previously allocated pages of memory.

dnPages—Dword with the count of pages to deallocate. If `dnPages` is greater than the existing span of pages allocated at this address, an error is returned, but as many pages as can be, *will* be deallocated. If fewer pages are specified, only that number will be deallocated.

DeviceInit

```
DeviceInit(dDevice, pInitData, dInitData) : dError
```

Some devices may require a call to initialize them before use, or to reset them after a catastrophe. An example of initialization would be a comms port, for baud rate, parity, and so on. The size of the initializing data and its contents are device-specific and are defined with the documentation for the specific device driver.

Parameters:

`dDevice`—Dword indicating device number.

`pInitData`—Pointer to device-specific data for initialization. This is documented for each device driver.

`dInitData`—Total number of bytes in the initialization data.

DeviceOp

```
DeviceOp(dDevice, dOpNum, dLBA, dnBlocks, pData): dError
```

The `DeviceOp()` function is used by services and programs to carry out normal operations such as Read and Write on all installed devices. The `dOpNum` parameter tells the driver which operation is to be performed. The first 256 operation numbers, out of over 4 billion, are pre-defined or reserved. These reserved numbers correspond to standard device operations such as read, write, verify, and format. Each device driver documents all device operation numbers it supports.

All drivers must support `dOpNum` 0 (Null operation). It is used to verify the driver is installed. Most drivers will implement `dOpNums` 2 and 3 (Read and Write). Disk devices (DASD—Direct Access Storage Devices) will usually implement the first eight or more.

Parameters:

`dDevice`—The device number.

`dOpNum`—Identifies which operation to perform 0 Null operation.

1	Read (receive data from the device)
2	Write (send data to the device)
3	Verify (compare data on the device)
4	Format Block
5	Format Track (disk devices only)
6	Seek Block
7	Seek Track (disk devices only)
8–255	RESERVED
256-*n*	Driver Defined (driver specific)

dLBA—Logical Block Address for I/O operation. For sequential devices, this parameter will usually be ignored. See the specific device driver documentation.

dnBlocks—Number of *contiguous* blocks for the operation specified. For sequential devices, this will probably be the number of bytes (comms, for example). Block size is defined by the driver. Standard block size for disk devices is 512 bytes.

pData—Pointer to data, or return buffer for reads, for specified operation.

DeviceStat

```
DeviceStat(dDevice, pStatRet, dStatusMax, dStatusRet):dError
```

The DeviceStat() function returns device-specific status to the caller if needed. Not all devices will return status on demand. In cases where the driver doesn't, or can't, return status, ErcNoStatus will be returned. The status information will be in record or structure format that is defined by the specific device driver documentation.

Parameters:

dDevice—Device number to status.

pStatBuf—Pointer to buffer where status will be returned.

dStatusMax—This is the maximum number of bytes of status you will accept.

pdStatusRet—Pointer to dword where the number of bytes of status returned to is reported.

DMASetUp

```
DMASetUp(dPhyMem, sdMem, dChannel, fdRead, dMode): dError
```

This is a routine used by device drivers to set up a DMA channel for a device Read or Write. Typical use would be for a disk or comms device, in which the driver would setup DMA for the move, then setup the device, which controls DMA through the DREQ/DACK lines, to actually control the data move.

Parameters:

dPhyMem—Physical memory address. Device drivers can call the GetPhyAdd() routine to convert linear addresses to physical memory addresses.

sdMem—Number of bytes to move.

dChannel—Legal values are 0, 1, 2, 3, 5, 6, and 7. Channels 5, 6, and 7 are for 16-bit devices, and though they move words, you must specify the length in *bytes* using sdMem.

fdRead—Any non-zero value sets DMA to read (Read is a read from memory which sends data *to* a device).

`dMode`—0 is Demand mode, 1 is Single Cycle mode (disk I/O uses single cycle), 2 is Block mode, and 3 is Cascade mode, which is for operating system use only.

EditLine

```
EditLine( pStr, dCrntLen, dMaxLen, pdLenRet, pbExitChar, dEditAttr): dError
```

This function displays a single line of text and allows it to be edited on screen. Display and keyboard input are handled inside of `EditLine()`. The line must be on a single row of the display with no character wrap across lines. The first character of the line is displayed at the current X and Y cursor coordinates for the caller's video screen.

Parameters:

`pStr`—Pointer to the character string to be edited.

`dCrntLen`—Current length of the string. The maximum is 80.

`dMaxlen`—Maximum length of the string. The maximum is 80.

`pdLenRet`— Pointer to a dword where the length of the string after editing will be returned.

`pbExitCharRet`— Pointer to a byte where the exit key from the edit operation is returned.

`dEditAttr`—Editing attribute.

The display and keyboard are handled entirely inside `EditLine()`. The following keys are recognized and handled internally for editing features. Any other key causes `EditLine()` to exit, and return the contents and length of the string in its current condition.

`08` (Backspace)—Move cursor to left, replacing char with 20h, which is a destructive backspace.

Left-Arrow—Same as Backspace (destructive)

20-7E Hex—Places character in current position and advances position.

Special shift keys combination (Ctrl and Alt) are *not* interpreted (Ctrl-D = D, and so on).

EndOfIRQ

```
EndOfIRQ (dIRQnum)
```

This call is made by an interrupt service routine to signify it has completed servicing the interrupt. End of Interrupt commands are sent to the proper PICU(s) for the caller.

Parameter:

`dIRQnum`—The hardware interrupt request number for the IRQ that your ISR has finished serving. For a description of hardware IRQs, see the call description for `SetIRQVector()`.

ExitJob

```
ExitJob(dExitError): dError
```

`ExitJob()` is called by applications and by services to terminate all tasks belonging to the job, and to free system resources. This is the usual method to terminate your application program. A high-level language's exit function would call `ExitJob()`.

After freeing certain resources for the job, `ExitJob()` attempts to load the `ExitJob` run file as specified in the `SetExitJob()` call. It is kept in the JCB. Applications can exit, and replace themselves with another program of their choosing, by calling `SetExitJob()`, and specifying the Run filename, before calling `ExitJob()`. A command line interpreter, or executive, which runs your program from a command line, would normally call `SetExitJob()`, specifying itself so it will run again when your program is finished.

If no `ExitJob` is specified, `ExitJob()` errors out with `ErcNoExitJob` and frees up all resources that were associated with that job, including, but not limited to, the JCB, TSSs, exchanges, link blocks, and communications channels. If this happens while video and keyboard are assigned to this job, the video and keyboard are reassigned to the OS Monitor program.

Parameters:

> `dExitError`—This is a dword that indicates the completion state of the application. The operating system does *not* interpret, or act on this value, in any way except to place it in the Job Control Block.

FillData

```
FillData(pDest, cBytes, bFill)
```

This is used to fill a block of memory with a repetitive byte value such as zeroing a block of memory. This uses the Intel string intrinsic instructions.

Parameters:

> `pDest`—A pointer to the block of memory to fill.
>
> `cBytes`—The size of the block.
>
> `bFill`—The byte value to fill it with.

GetCmdLine

```
GetCmdLine(pCmdLineRet, pdcbCmdLineRet): dError
```

Each Job Control Block has a field to store the command line that was set by the previous application or command line interpreter. This field is not directly used or interpreted by the operating system. This call retrieves the command line from the JCB. The complimentary call,

`SetCmdLine()`, is provided to set the command line. The `CmdLine` is preserved across `ExitJobs`, and while chaining to another application.

High-level language libraries may use this call to retrieve the command line and separate or expand parameters (arguments) for the applications main function.

Parameters:

`pabCmdLineRet`—Pointer to an array of bytes where the command line string will be returned. This array must be large enough to hold the largest CmdLine, which is 79 characters.

`pdcbCmdLineRet`—A pointer to a dword where the length of the command line returned will be stored.

GetCMOSTime

```
GetCMOSTime(pdTimeRet) : dError
```

This retrieves the time from the on-board CMOS battery backed up clock. The time is returned from the CMOS clock as a dword defined as:

Low order byte is the seconds (BCD),
Next byte is the minutes (BCD),
Next byte is the hours (BCD 24 hour),
High order byte is 0.

Parameters:

`pdTimeRet`—A pointer to a dword where the time is returned in the previously described format.

GetCMOSDate

```
GetCMOSDate(pdDateRet): dError
```

This retrieves the date from the on-board CMOS battery backed up clock. The date is returned from the CMOS clock as a dword defined as:

Low order byte is the Day of Week (BCD 0–6 0=Sunday),
Next byte is the Day (BCD 1–31),
Next byte is the Month (BCD 1–12),
High order byte is Year (BCD 0–99).

Parameters:

`pdTimeRet`—A pointer to a dword where the date is returned in the previously described format.

GetDMACount

```
GetDMACount(dChannel, pwCountRet): dError
```

A routine used by device drivers to get the count of bytes for 8-bit channels, or words for 16-bit DMA channels, left in the Count register for a specific DMA channel.

Parameters:

`dChannel`—Legal values are 0, 1, 2, 3, 5, 6, and 7. Channels 5, 6, and 7 are for 16-bit devices.

`pwCountRet`—This is the value found in the DMA count register for the channel specified. Note that even though you specify bytes on 16-bit DMA channel to the `DMASetUp()` call, the count returned is always the exact value found in the count register for that channel. The values in the DMA count registers are always programmed with *one less* than the count of the data to move. Zero (0) indicates one word or byte, 15 indicates 16 words or bytes. If you program a 16-bit DMA channel to move data from a device for 1024 words (2048 bytes) and you call `GetDMACount()`, which returns `1` to you, this means all but two words were moved.

GetExitJob

```
GetExitJob(pabExitJobRet, pdcbExitJobRet): dError
```

Each Job Control Block has a field to store the `ExitJob` filename. This field is used by the operating system when an application exits, to see if there is another application to load in the context of this job (JCB). Applications may use this call to get the `ExitJob` filename. The complimentary call, `SetExitJob()`, is provided. The `ExitJob` remains set until it is reset. Calling `SetExitJob()` with a zero length value clears the exit job file name from the JCB.

Parameters:

`pabExitJobRet`—Points to an array of bytes where the `ExitJob` filename string will be returned. This array must be large enough to hold the largest `ExitJob` filename of 79 characters.

`pdcbExitJobRet`—A pointer to a dword where the length of the `ExitJob` filename returned will be stored.

GetIRQVector

```
GetIRQVector (dIRQnum, pVectorRet): dError
```

The get interrupt vector call returns the 32-bit offset address in operating system address space for the ISR that is currently serving `dIRQnum`.

Parameters:

`dIRQnum`—The hardware interrupt request number for the vector you want returned. For a description of IRQs, see the call description for `SetIRQVector()`.

`pVectorRet`—A pointer where the address of the ISR will be returned.

GetJobNum

```
GetJobNum(pdJobNumRet):dError
```

This returns the job number for the task that called it. All tasks belong to a job.

Parameters:

`pbJobNumRet`—A pointer to a dword where the job number is returned.

GetNormVid

```
GetNormVid(pdNormVidAttrRet): dError
```

Each job is assigned a video screen, virtual or real. The normal video attributes used to display characters on the screen can be different for each job. The default foreground color, which is the color of the characters themselves, and background colors, may be changed by an application. The normal video attributes are used with the `ClrScr()` call, and are also used by stream output from high level language libraries. This returns the normal video attribute for the screen of the caller. See `SetNormVid()` for more information.

Parameters:

`pdNormVidAttrRet`—This points to a dword where the value of the normal video attribute will be returned. The attribute value is always a byte, but is returned as a dword for this, and some other operating system video calls.

See Chapter 9, "Application Programming," for a table containing values for video attributes.

GetPath

```
GetPath(dJobNum, pPathRet, pdcbPathRet): dError
```

Each Job Control Block has a field to store an application's current path. The path is interpreted and used by the file system. Refer to file system documentation. This call allows applications and the file system to retrieve the current path string.

Parameters:

`dJobNum`—This is the job number for the path string you want.

`pPathRet`—This points to an array of bytes where the path will be returned. This string must be long enough to contain the longest path of 69 characters.

pdcbPathRet—This points to a dword where the length of the path name returned is stored.

GetPhyAdd

```
GetPhyAdd(dJobNum, dLinAdd, pdPhyRet): dError
```

This returns a physical address for the given linear address. It is used by device drivers for DMA operations on DSeg memory.

Parameters:

dJobNum—The job number of owner of dLinAdd.

dLinAdd—The linear address you want the physical address for. This is the address itself, not a pointer to it.

pdPhyRet—Points to the dword where the physical address is returned.

GetpJCB

```
GetpJCB(dJobNum, ppJCBRet): dError
```

This returns a pointer to the Job Control Block for the specified job number. Each job has a structure to control and track present job resources and attributes. The data in a JCB is *read-only* to applications and system services.

Parameters:

dJobNum—This is the job number for the Job Control Block you want a pointer to.

ppJCBRet—This is a pointer that points to a pointer where the pJCB will be returned.

GetSysIn

```
GetSysIn(pFileNameRet, pdcbFileNameRet): dError
```

This returns the name of the standard output file, or device, from the application's Job Control Block.

Parameters:

pFileNameRet—Pointer to an array where the name will be returned. This array must be at least 30 characters long.

pdcbFileNameRet—Pointer to a dword where the length of the name will be returned.

GetSysOut

```
GetSysOut(pFileNameRet, pdcbFileNameRet): dError
```

This returns the name of the standard output file or device from the application's Job Control Block.

Parameter:

`pFileNameRet`—Pointer to an array where the name will be returned. This array must be at least 30 characters long.

`pdcbFileNameRet`—Pointer to a dword where the length of the name will be returned.

GetTimerTick

```
GetTimerTick( pdTickRet) : dError
```

This returns the operating system tick counter value. When MMURTL is booted, it begins counting 10-millisecond intervals forever. This value will roll over every 16 months or so. It is an unsigned dword.

Parameter:

`pdTickRet`—A pointer to a dword where the current tick is returned.

GetTSSExch

```
GetTSSExch(pdExchRet): dError
```

This returns the exchange that belongs to the task as assigned by the operating system during `SpawnTask()` or `NewTask()`. This exchange is used by some direct calls, while blocking a thread on entry to non-reentrant portions of the operating system. It can also be used by system services or library code as an exchange for the request primitive for that task, as long as no timer functions will be used, such as `Alarm()` or `Sleep()`.

Parameter:

`pdExchRet`—A pointer to a dword where the TSS exchange will be returned.

GetUserName

```
GetUserName(pUserNameRet, pdcbUserNameRet): dError
```

The Job Control Block for an application has a field to store the current user's name of an application. This field is not directly used or interpreted by the operating system. This call retrieves the user name from the JCB. The complimentary call `SetUserName()` is provided to set the user name. The name is preserved across `ExitJobs`, and while chaining to another application.

Parameters:

pabUsernameRet—Points to an array of bytes where the user name will be returned.

dcbUsername—The length of the string to store.

GetVidChar

```
GetVidChar(dCol, ddLine, pbCharRet, pbAttrRet): dError
```

This returns a single character and its attribute byte from the caller's video screen.

Parameters:

dCol—The column location, or X position, of the character you want returned.

dLine—The line location, or Y position, of the character you want returned.

pbCharRet—Pointer to a byte where the character value will be returned.

pbAttrRet—Pointer to a byte where the video attribute for the character is returned.

GetVidOwner

```
GetVidOwner(pdJobNumRet): dError
```

This returns the job number for the owner of the real video screen, the one you are viewing.

Parameter:

pdJobNumRet—Pointer to a dword where the current video owner will be returned.

GetXY

```
GetXY(pdXRet, pdYRet): dError
```

This returns the current X and Y position of the cursor in your virtual video screen.

Parameters:

pdXRet—A pointer to where you want the current horizontal cursor position returned.

pdYRet—A pointer to where you want the current vertical cursor position returned.

InByte

```
InByte(dPort) : Byte
```

This reads a single byte from a port address, and returns it from the function.

Parameter:

dPort—The address of the port.

InDWord

```
InDWord(dPort) : DWord
```

This reads a single dword from a port address, and returns it from the function.

Parameter:

`dPort`—The address of the port.

InWord

```
InWord(dPort) : Word
```

This reads a single word from a port address, and returns it from the function.

Parameter:

`dPort`—The address of the port.

InWords

```
InWords(dPort, pDataIn, dBytes)
```

`InWords` reads one or more words from a port using the Intel string read function. The data is read from `dPort` and returned to the address `pDataIn`. `dBytes` is the total count of bytes to read (words × 2). This call is specifically designed for device drivers.

Parameters:

`dPort`—The address of the port.

`pDataIn`—The address where the data string will be returned.

`dBytes`—The total number of bytes to be read. This number is the number of words you want to read from the port times two.

InitDevDr

```
InitDevDr(dDevNum, pDCBs, nDevices, fReplace) : dError
```

`InitDevDr()` is called from a device driver after it is first loaded, to let the operating system integrate it into the system. After the device driver has been loaded, it should allocate *all* system resources it needs to operate and control its devices, while providing the three standard entry points. A 64-byte DCB must be filled out for each device the driver controls before this call is made.

When a driver controls more than one device, it must provide the device control blocks for each device. The DBCs must be contiguous in memory. If the driver is flagged as not reentrant, then all devices controlled by the driver will be locked out when the driver is busy.

This is because one controller, such as a disk or SCSI controller, usually handles multiple devices through a single set of hardware ports, and one DMA channel if applicable, and can't handle more than one active transfer at a time. If this is not the case, and the driver can handle two devices simultaneously at different hardware ports, then it may be advantageous to make it two separate drivers.

See Chapter 10, "Systems Programming," for more detailed information on writing device drivers.

Parameters:

`dDevNum`—This is the device number that the driver is controlling. If the driver controls more than one device, this is the first number of the devices. This means the devices are numbered consecutively.

`pDCBs`—This is a pointer to the DCB for the device. If more than one device is controlled, this is the pointer to the first in an array of DCBs for the devices. This means the second DCB would be located at `pDCBs + 64`, the third at `pDCBs + 128`, and so on.

`nDevices`—This is the number of devices that the driver controls. It *must* equal the number of contiguous DCBs that the driver has filled out before the `InitDevDr()` call is made.

`fReplace`—If true, the driver will be substituted for the existing driver functions already in place. This does not mean that the existing driver will be replaced in memory, it only means the new driver will be called when the device is accessed. A driver *must* specify and control at least as many devices as the original driver handled.

ISendMsg

```
ISendMsg (dExch, dMsgHi, dMsgLo): dError
```

This is the Interrupt Send primitive. It provides access to the operating system to allow an interrupt procedure to send information to another task via an exchange. This is the same as `SendMsg()` except *no* task switch is performed and interrupts remain cleared. If a task is waiting at the exchange, the message is associated with that task and it is moved to the Ready Queue. It will get a chance to run the next time the `RdyQ` is evaluated by the kernel. Interrupt tasks can use `ISendMsg()` to send single or multiple messages to exchanges during their execution.

NOTE

Interrupts are cleared on entry in `ISendMsg()` and will not be set on exit! It is the responsibility of the caller to set them if desired. This procedure should only be used by device drivers and interrupt service routines.

The message is two double words that must be pushed onto the stack independently. When `WaitMsg()`, or `CheckMsg()`, returns these to your intended receiver, it will be into an array of dwords, unsigned long `msg[2]` in C. `dMsgLo` will be in the lowest array index (`msg[0]`), and `dMsgHi` will be in the highest memory address (`msg[1]`).

Parameters:

`dExch`—A dword (4 bytes) containing the exchange to where the message should be sent.

`dMsgHi`—The upper dword of the message.

`dMsgLo`—The lower dword of the message.

KillAlarm

```
KillAlarm (dAlarmExch) : dError
```

A public routine that kills an alarm message that was set to be sent to an exchange by the `Alarm()` operating-system function. All alarms set to fire off to the exchange you specify are killed. For instance, if you used the `Alarm()` function to send a message to you in three seconds so you could time-out on a piece of hardware, and you didn't need it anymore, you would call `KillAlarm()`. If the alarm is already queued through the kernel, *nothing* will stop it, and you should expect the `Alarm()` message at the exchange.

Parameter:

`dAlarmExch`—The exchange you specified in a previous `Alarm()` operating system call.

LoadNewJob

```
LoadNewJob(pFileName, dcbFileName, pdJobNumRet) : dError
```

This loads and executes a run file. This allocates all the system resources required for the new job including a Job Control Block, page descriptor, initial Task State Segment, virtual video, and the memory required to run code, stack, and data pages.

Parameters:

`pFilename`—Pointer to the filename to run.

`dcbFileName`—Length of the run filename.

`pdJobNumRet`—Pointer to a dword where the new job number will be returned.

MaskIRQ

```
MaskIRQ (dIRQnum) : dError
```

This masks the hardware Interrupt Request specified by `dIRQnum`. When an IRQ is masked in hardware, the CPU will not be interrupted by the particular piece of hardware even if interrupts are enabled.

Parameter:

> `dIRQnum`—The hardware interrupt request number for the IRQ you want to mask. For a description of IRQs, see the call description for `SetIRQVector()`.

MicroDelay

```
MicroDelay(dn15us) : dError
```

`MicroDelay` is used for very small, precise, delays that may be required when interfacing with device hardware. It will delay further execution of your task by the number of 15-microsecond intervals you specify. You should be aware that your task does *not* go into a wait state, but instead is actually consuming CPU time for this interval.

Interrupts are not disabled, so the time could be much longer, but a task switch will probably not occur during the delay even though this is possible. `MicroDelay` guarantees the time to be no less than `n-15us` where *n* is the number of 15-microsecond intervals you specify. The timing is tied directly to the `15us` RAM refresh hardware timer.

The recommended maximum length is 20-milliseconds, although longer periods will work. Interrupt latency of the system is not affected for greater delay values, but application speed or appearance may be affected if the priority of your task is high enough, and you call this often enough.

Parameter:

> `dn15us`—A dword containing the number of 15-microsecond intervals to delay your task.

MoveRequest

```
MoveRequest(dRqBlkHndl, DestExch) : dError
```

The kernel primitive `MoveRequest()` allows a system service to move a request to another exchange it owns. An example of use is when a system service receives a request it can't answer immediately. It would move it to a second exchange until it can honor the request.

This *cannot* be used to forward a request to another service or job because the data pointers in the outstanding request block have been aliased for the service the request was destined for.

Parameters:

dRqBlkHndl—Handle of the request block to forward.

DestExch—Exchange where the request is to be sent.

NewTask

```
NewTask(dJobNum, dCodeSeg, dPriority, dfDebug, dExch, dESP, dEIP) : dError
```

This allocates a new Task State Segment (TSS), then fills in the TSS from the parameters to this call and schedules it for execution. If the priority of this new task is greater than the running task, it is made to run, else it's placed on the Ready Queue in priority order.

Most applications will use `SpawnTask()` instead of `NewTask()` because it's easier to use. The operating system uses this to create the initial task for a newly loaded job.

Parameters:

JobNum—Job number the new task belongs to (JCB)

CodeSeg—Which code segment, OS=8 or User=18

Priority—0–31 (Primary application tasks use 25)

fDebug—Non-zero if you want the enter the debugger immediately upon execution.

Exch—Exchange for TSS.

ESP—Stack pointer (offset in DSeg)

EIP—Initial instruction pointer (offset in CSeg)

OutByte

```
OutByte(Byte, dPort)
```

This writes a single byte to a port address. No error is returned.

Parameters:

Byte—The byte value to write to the port.
dPort—The address of the port.

OutDWord

```
OutDWord(DWord, dPort)
```

This writes a single dword to a port address.

DWord—The dword value to write to the port.
dPort—The address of the port.

OutWord

```
OutWord(Word, dPort)
```

This writes a single word to a port address.

Parameters:

`Word`—The word value to write to the port.
`dPort`—The address of the port.

OutWords

```
OutWords(dPort, pDataOut, dBytes)
```

This uses the processor string intrinsic function `OUTSW` to write the words pointed to by `pDataOut` to `dPort`. You specify the number of bytes total (words × 2).

Parameters:

`dPort`—The address of the port.

`pDataOut`—A pointer to one or more dwords to send out the port.

`dBytes`—The count of bytes to write to the port address. This is the number of words × 2.

PutVidAttrs

```
PutVidAttrs(dCol, dLine, dnChars, dAttr): dError
```

This applies the video attribute specified to the characters at the column and line specified on the video screen, or if not displayed, the virtual screen.

This is done independently of the current video stream which means X and Y cursor coordinates are not affected.

Parameters:

`dCol`—Column to start on (0–79)

`dLine`—Line (0–24)

`dnChars`—Dword with number of characters to apply the attribute to.

`dAttr`—Number of characters `pbChars` is pointing to. The Basic Video section in Chapter 9, "Application Programming," describes all of the attribute values.

PutVidChars

```
PutVidChars(dCol, dLine, pbChars, nChars, dAttrib): dError
```

This places characters with the specified attribute directly on the caller's video screen. It is independent of the current video stream, which means X and Y cursor coordinates are not affected.

Parameters:

`dCol`—Column to start on (0–79)

`dLine`—Line (0–24)

`pbChars`—Pointer to the text to be displayed

`nChars`—Number of characters `pbChars` is pointing to

`dAtrib`—Color or attribute to use during display. The "Basic Video" section in Chapter 9 describes all of the attribute values.

QueryPages

```
QueryPages(pdnPagesRet): dError
```

This returns the number of free physical memory pages left in the entire operating system managed memory space. Each page of memory is 4096 bytes.

Parameter:

`pdnPagesRet`—A pointer to a dword where the current count of free pages will be returned.

ReadCMOS

```
ReadCMOS(bAddress):Byte
```

This reads a single byte value at the address specified in the CMOS RAM and returns it from the function. This is specific to the ISA/EISA PC architecture.

Parameter:

`bAddress`—The address of the byte you want from 0 to `MaxCMOSSize`

ReadKbd

```
ReadKbd (pdKeyCodeRet, fWait) : dError
```

`ReadKbd()` is provided by the keyboard system service as the easy way for an application to read keystrokes. Access to all other functions of the keyboard service are only available through the Request and Wait primitives.

This call blocks your task until completion. If `fWait` is true (non-zero), the call will wait until a key is available before returning. If `fWait` is false (0), the call will return the keycode to

pdKeyCodeRet and return with no error. If no key was available, error ErcNoKeyAvailable will be returned.

Parameters:

pdKeyCodeRet—A pointer to a dword where the keycode will be returned. See the documentation on the keyboard service for a complete description of keycodes.

fWait—If true (non-zero), the call will wait until a keycode (user keystroke) is available before returning.

RegisterSvc

```
RegisterSvc(pSvcName, dExch) : dError
```

This procedure registers a message-based system service with the operating system. This will identify a system service name with a particular exchange. This information allows the operating system to direct requests for system services without the originator (Requester) having to know where the actual exchange is located, or on what machine the request is being serviced, if forwarded across a network.

Parameters:

pSvcName—A pointer to an 8-byte string. The string must be left justified, and padded with spaces (20h). Service names *are* case sensitive. Examples: "KEYBOARD" and "FILESYS "

dExch—The exchange to be associated with the service name.

Request

```
Request( pSvcName,
         SvcCode,
         dRespExch,
         pRqHndlRet,
         npSend,
         pData1,
         cbData1,
         pData2,
         cbData2,
         dData0,
         dData1,
         dData2 ): dError
```

This is the operating system Request primitive. It allows a caller to send a request for services to a message-based system service.

With all kernel primitives, dError, the value returned from the Request() call itself, is an indication of whether or not the Request() primitive functioned normally. Zero will be returned if the Request() primitive was properly routed to the service, otherwise an operating system status code will be returned. For example, if you made a Request() to QUEUEMGR and no

service named QUEUEMGR was installed, a `dError` indicating this problem would be returned (`ErcNoSuchService`). This error is *not* the error returned by the service. An error from the service is returned to your exchange (`dRespExch`) in the response message.

Parameters:

`pSvcName`—A pointer to an 8-byte, left justified, space padded, service name. Service name examples are "QUEUEMGR" or "FAXIN ". The service must be registered with the operating system. It must be installed or included in the operating system at build time.

`wSvcCode`—The service code is a 16-bit unsigned word that identifies the action the service will perform for you. Values for the service code are documented by each service. Function codes 0, 65534 (0FFFEh), and 65535 (0FFFFh) are always reserved by the operating system.

`dRespExch`—The exchange you want the response message to be sent to upon completion of the request. See `Respond()`.

`pRqHndlRet`—A pointer to a dword that will hold the request handle the request call will fill in. This is how you identify the fact that it is a request and not a non-specific message when you receive a message after waiting or checking at an exchange.

`npSend`—A number (0, 1, or 2) that indicates how many of the two `pData` pointers, 1 and 2, are sending data to the service. If `pData1` and `pData2` both point to data in your memory area that is going to the service, then `npSend` would be 2. What each service expects is documented by the service for each service code. The descriptions for each service code within a service will tell you the value of `npSend`. The service actually knows which pointers it's reading or writing. The `npSend` parameter is so network transport systems will know which way to move data on the request if it is routed over a network. Always assume it will be network-routed.

`pData1`—A pointer to your memory area that the service can access. This may be data sent to the service, or a memory area where data will be returned from the service. Use of `pData1` and `pData2` are documented by the system service for each service code it handles.

`cbData1`—The size of the memory area in `pData1` being sent to, or received from, the service. This *must* be 0 if `pData1` is not used.

`pData2` —Same as `pData1`. This allows a service to access a second memory area in your space if needed.

`cbData2`—The size of the data being sent to the service. This *must* be 0 if `pData2` is not used.

`dData0`, `dData1`, `dData2`—These are dwords used to pass data to a service. These are strictly for data and *cannot* contain pointers to data. The reason for this is that the operating system does not provide alias memory addresses to the service for them. The use of `dData0`, `dData1`, and `dData2` is defined by each service code handled by a system service.

Respond

```
Respond(dRqHndl, dStatRet): dError
```

`Respond()` is used by message-based system services to respond to the caller that made a request to it. A service must respond to all requests using `Respond()`. Do not use `SendMsg()` or `ISendMsg()`.

With all kernel primitives, `dError`, the value returned from the `Respond()` call itself, is an indication of whether or not the `Respond` primitive functioned normally. Zero will be returned if all was well, otherwise an operating-system status code will be returned.

Parameters:

`dRqHndl`—This is the Request handle that was received by the service at `WaitMsg()` or `CheckMsg()`. It identifies a Request Block resource that was allocated by the operating system.

`dStatRet`—This is the status or error code that will be placed in the second dword, `dMsgLo`, of the message sent back to the requester of the service. This is the *error/status* that the service passes back to you to let you know it has handled your request properly.

ScrollVid

```
ScrollVid(dULCol,dULline,dnCols,dnLines, dfUp)
```

This scrolls the described square area on the screen either *up* or *down* `dnLines`. If `dfUp` is *non-zero* the scroll will be *up*. The line(s) left blank is filled with character 20h and the normal video attribute stored in the JCB.

Parameters:

`dULCol`— The *upper left* column on which to start (0–79).

`dULLine`—The *upper left* line (0–24).

`dnCols`—The number of columns to be scrolled.

`dnLines`—The count of lines to be scrolled.

`dfUp`—This *non-zero* to cause the scroll to be up instead of down.

To scroll the entire screen *up* one line, the parameters are `ScrollVid(0,0,80,25,1)`. In this case the top line is lost, and the bottom line would be blanked.

SendMsg

```
SendMsg (dExch, dMsgHi, dMsgLo): dError
```

`SendMsg()` allows a task to send information to another task. The two-dword message is placed on the specified exchange. If there was a task waiting at that exchange, the message is associated, linked with that task, and it is moved to the Ready queue.

Users of `SendMsg()` should be aware that they can receive responses from services *and* messages at the same exchange. The proper way to ensure the receiver of the message doesn't assume it is a Response, is to set the high order bit of `dMsgHi`, which is the first dword pushed. If it were a response from a service, the first DWord would be less than 80000000h, which is a positive 32-bit signed value. This is an agreed-upon convention, and is not enforced by the operating system. If you never make a request that indicates exchange X is the response exchange, you will not receive a response at the X exchange unless the operating system has gone into an undefined state—a polite term for *crashed.*

The message is two double words that must be pushed onto the stack independently. When `WaitMsg()` or `CheckMsg()` returns these to your intended receiver, it will be into an array of dwords. In C this is unsigned long `msg[2]`. `dMsgLo` will be in the lowest array index, `msg[0]`, and `dMsgHi` will be in the highest memory address, `msg[1]`.

Parameters:

`dExch`—A dword (4 bytes) containing the exchange where the message should be sent.

`dMsgHi`—The upper dword of the message.

`dMsgLo`—The lower dword of the message.

SetCmdLine

```
SetCmdLine(pCmdLine, dcbCmdLine): dError
```

Each Job Control Block has a field to store the command line that was set by the previous application or command line interpreter. This field is not directly used or interpreted by the operating system. This call stores the command line in the JCB. The complimentary call, `GetCmdLine()`, is provided to retrieve the command line. The `CmdLine` is preserved across `ExitJobs`, and while chaining to another application.

Command line interpreters may use this call to set the command line text in the JCB prior to `ExitJob()` or `Chain()` calls running an application.

Parameters:

`pCmdLine`—Points to an array of bytes, the new command line string.

`dcbCmdLine`—The length of the command line string to store. The command line field in the JCB is a maximum of 79 characters. Longer values will cause `ErcBadJobParam` to be returned.

SetExitJob

```
SetExitJob(pabExitJobRet, pdcbExitJobRet): dError
```

Each Job Control Block has a field to store the `ExitJob` filename. This call sets the `ExitJob` string in the JCB. This field is used by the operating system when an application exits to see if

there is another application to load in the context of this job (JCB). Applications may use this call to set the name of the next job they want to execute in this JCB. This is typically used by a command line interpreter.

The complimentary call `GetExitJob()` is provided. The `ExitJob` remains set until it is set again. Calling `SetExitJob` with a zero-length value clears the exit job filename from the JCB. When a job exits, and its `ExitJob` is zero length, the job is terminated and all resources are reclaimed by the operating system.

Parameters:

`pExitJob`—Points to an array of bytes containing the new `ExitJob` filename. The filename must not be longer than 79 characters.

`dcbExitJob`—A dword with the length of the string the `pExitJob` points to.

SetIRQVector

```
SetIRQVector(dIRQnum, pVector)
```

The Set Interrupt Vector call places the address, `pVector`, into the interrupt descriptor table and marks it as an interrupt procedure. `pVector` *must* be a valid address in the operating-system address space because the IDT descriptor makes a protection level transition to *system* level if the interrupt occurred in a user level task, which will happen most often. Initially, all unassigned hardware interrupts are masked at the priority interrupt controller units (PICUs). After the interrupt vector is set, the caller should then call `UnMaskIRQ()` to allow the interrupts to occur.

Parameters:

`dIRQnum`—The hardware interrupt request number for the IRQ to be set. This will be 0–7 for interrupts on 8259 #1, and 8–15 for interrupts on 8259 #2.

Table 15.4 shows the predetermined IRQ uses.

Table 15.4. Hardware IRQs.

IRQ 0	8254 Timer
IRQ 1	Keyboard (8042)
IRQ 2	Cascade from PICU2 (handled internally)
IRQ 3	COM2 serial port *
IRQ 4	COM1 serial port *
IRQ 5	LPT2 *
IRQ 6	Floppy disk controller *
IRQ 7	LPT1 *

IRQ 8	CMOS clock
IRQ 9	Not pre-defined
IRQ 10	Not pre-defined
IRQ 11	Not pre-defined
IRQ 12	Not pre-defined
IRQ 13	Math coprocessor *
IRQ 14	Hard disk controller *
IRQ 15	Not pre-defined

*If installed, these should follow ISA hardware conventions. Built-in device drivers assume these values.

`pVector`—The 32-bit address in operating system protected address space, of the Interrupt Service Routine (ISR) that handles the hardware interrupt.

SetJobName

```
SetJobName(pJobName, dcbJobName): dError
```

This sets the job name in the job control block. This is a maximum of 13 characters. It is used only for the display of job names and serves no other function. If no job name is set by the application, the last 13 characters of the complete path for the run file that is executing are used.

Parameters:

`pJobName`—Pointer to the job name you want to store in the job control block.

`dcbJobName`—Count of bytes in the `pJobName` parameter. The maximum length is 13 bytes. Longer values will be truncated. A zero length, will clear the job name from the JCB.

SetNormVid

```
SetNormVid(dNormVidAttr): dError
```

Each job is assigned a video screen, either virtual or real. The normal video attributes used to display characters on the screen can be different for each job. The default foreground color, the color of the characters themselves, and background colors may be changed by an application. The normal video attributes are used with the `ClrScr()` call, and also used by stream output from high level language libraries. If you `SetNormVid()` to Black On White, then call `ClrScr()`, the entire screen will become black characters on a white background.

Parameters:

dNormVidAttr—This is a dword containing the value of the video attribute to become the normal or default. The attribute value is always a byte, but passed in as a dword for this, and some other operating system video calls.

See Chapter 9 for a table containing values for video attributes.

SetPath

```
SetPath(pPath, dcbPath): dError
```

Each job control block has a field to store the application's current path, a text string. The path is interpreted and used by the file system. Refer to file system documentation. This call allows applications to set their path string in the JCB.

Typically, a command line interpreter would use this call to set the current path.

Parameters:

pPath—This points to an array of bytes containing the new path string.

dcbPath—This is a dword containing the length of the new path for the current job. The path is 79 characters maximum.

SetPriority

```
SetPriority(bPriority): dError
```

This changes the priority of the task that is currently executing. Because the task that is running is the one that needs its priority changed, you can ensure it is rescheduled at the new priority by making any operating-system call that causes this task to be suspended. The `Sleep()` function with a value of 1 would have this effect.

Parameter:

bPriority—This is the new priority (0–31).

SetSysIn

```
SetSysIn(pFileName, dcbFileName): dError
```

This sets the name of the standard input file stored in the job control block. The default name, if none is set by the application, is KBD. The keyboard system service bypasses this mechanism when you use the request/respond interface. If you desire to read data from a file to drive your application, you should read stream data from the file system using the procedural

interface call `ReadBytes()`. Only ASCII data is returned from this call. If the `SysIn` name is set to KDB, it has the same effect as calling `ReadKbd()` with `fWait` set to true, and only receiving the low-order byte which is the character code.

Parameters:

`pFileName`—A maximum of 30 characters with the file or device name.

`dcbFileName`—The count of characters `pFileName` is pointing to.

SetSysOut

```
SetSysOut(pFileName, dcbFileName): dError
```

This sets the name of the standard output file stored in the job control block. The default name, if none is set, is VID. The direct access video calls bypass this mechanism. If you desire to write data from your application to a file, you should write stream data using the file system procedural interface call, `WriteBytes()`. Only ASCII data is saved to the file. If the `SysOut` name is VID, writing to the file VID with `WriteBytes()` is the same as calling `TTYOut()`.

Parameters:

`pFileName`—A maximum of 30 characters with the file or device name.

`dcbFileName`—The count of characters `pFileName` is pointing to.

SetUserName

```
SetUserName(pUserName, dcbUserName): dError
```

The job control block for an application has a field to store the current user's application name. This field is not used or interpreted by the operating system. The complimentary call, `GetUserName()`, is provided to retrieve it from the JCB. The name is preserved across `ExitJobs`, and chaining to another application.

Parameters:

`pUsername`—Points to a text string, which is the user's name.

`dcbUsername`—The length of the string to store.

SetVidOwner

```
SetVidOwner(dJobNum): dError
```

This selects which job screen will be displayed on the screen. This is used by the MMURTL Monitor and can be used by another user written program manager.

Parameter:

dJobNum—The job number for the new owner of the active screen, which is the one to be displayed.

SetXY

```
SetXY(dNewX, dNewY): dError
```

This positions the cursor for the caller's screen to the X and Y position specified. This applies to the virtual screen, or the real video screen if currently assigned.

Parameters:

dNewX—The new horizontal cursor position column.

dNewY—The new vertical cursor position row.

Sleep

```
Sleep(nTicks): dError
```

A public routine that delays the calling process by putting it into a waiting state for the amount of time specified. `Sleep()` should not be used for small, critical timing in applications less than 20 ms. The overhead included in entry and exit code makes the timing a little more than 10 ms, and is not precisely calculable. An example of the use of `Sleep()` is inside a loop that performs a repetitive function, and you do not want to create a *busy-wait* condition, which may consume valuable CPU time causing lower priority tasks to become starved. If your task has a high enough priority, a busy-wait condition may cause the CPU to appear locked up, when it's actually doing exactly what it's told to do.

`Sleep` sends `-1` (0FFFFFFFF hex) for both dwords of the message.

`Sleep` is accomplished internally by setting up a timer block with the countdown value, and an exchange to send a message when the countdown reaches zero. The timer interrupt sends the message, and clears the block when it reaches zero. The default exchange in the TSS is used so the caller doesn't need to provide a separate exchange.

Parameter:

dnTicks—Dword with the number of 10-millisecond periods to delay execution of the task, or make it sleep.

SpawnTask

```
SpawnTask(pEntry, dPriority, fDebug, pStack, fOSCode):dError
```

This creates a new task, or thread of execution, with the instruction address `pEntry`. This allocates a new task state segment (TSS), then fills in the TSS from the parameters you passed. Many of the TSS fields will be inherited from the task that called this function. If the priority of this new task is greater than the running task, it is made to run; otherwise, it's placed on the Ready queue in priority order.

Parameters:

`pEntry`—This is a pointer to the function, procedure, or code section that will become an independent thread of execution from the task that spawned it.

`dPriority`—This is a number from 0 to 31. Application programs should use priority 25. System services will generally be in the 12–24 region. Secondary tasks for device drivers will usually be lower than 12. Device drivers such as network frame handlers may even require priorities as high as 5, but testing will be required to ensure harmony. Print spoolers and other tasks that may consume a fair amount of CPU time, and may cause applications to appear sluggish, should use priorities above 25. Certain system services are higher priority, lower numbers, such as the file system and keyboard.

`fDebug`—A *non-zero* value will cause the task to immediately enter the debugger upon execution. The instruction displayed in the debugger will be the first instruction that is to be executed for the task.

`pStack`—A memory area in the data segment that will become the stack for this task. Operating system operation requires 512 bytes. Add the number of bytes your task will require to this value.

`fOSCode`—If you are spawning a task from a device driver, or within the operating system code segment, you should specify a non-zero value, 1 is accepted.

Tone

```
Tone(dHz, dTicks10ms): dError
```

A public routine that sounds a tone of the specified frequency on the system hardware speaker for the duration specified by `dTicks`. Each `dTick10ms` is 10 milliseconds.

Parameters:

dHz—Dword that is the frequency in Hertz, from 30 to 10,000.

dTicks10ms—The number of 10-millisecond increments to sound the tone. A value of 100 is one second.

TTYOut

```
TTYOut (pTextOut, dTextOut, dAttrib)
```

This places characters on the caller's screen in a TTY (teletype) fashion at the current X and Y coordinates for the calling job.

The following characters in the stream are interpreted as follows:

0A Line Feed—The cursor, next active character placement, will be moved down one line. If this line is below the bottom of the screen, the entire screen will be scrolled up on line, and the bottom line will be blanked

0D Carriage Return—The cursor will be moved to column zero on the current line.

08 BackSpace—The cursor will be moved one column to the left. If it is already at column 0, backspace will have no effect. The backspace is non-destructive, no characters are changed.

For character placement, if the cursor is in the last column of a line, it will be moved to the first column of the next line. If the cursor is in the last column on the last line, the screen will be scrolled up one line, and the bottom line will be blank. The cursor will then be placed in the first column on the last line.

Parameters:

pTextOut—A near pointer to the text.

dTextOut—The number of chars of text.

dAttrib—The attribute or color you want.

UnMaskIRQ

```
UnMaskIQR (dIRQnum) : dError
```

This *unmasks* the hardware interrupt request specified by dIRQnum. When an IRQ is masked in hardware, the CPU will not be interrupted by the particular piece of hardware even if interrupts are enabled.

Parameter:

`dIRQnum`—The hardware interrupt request number for the IRQ you want to unmask. For a description of IRQs see the call description for `SetIRQVector()`.

UnRegisterSvc

```
UnRegisterSvc(pSvcName) : dError
```

This procedure removes the service name from the operating system name registry. The service with that name will no longer be available to callers. They will receive an error.

Parameter:

`pSvcName`—A pointer to an 8-byte string. The string must be left justified and padded with spaces (20h). Service names *are* case sensitive. Examples: "KEYBOARD" and "FILESYS "

This name must exactly match the name supplied by the service when it was registered.

WaitMsg

```
WaitMsg(dExch,pqMsgRet):dError
```

This is the kernel `WaitMsg()` primitive. This procedure provides access to operating system messaging by allowing a task to receive information from another task. If no message is available, the task will be placed on the exchange specified. Execution of the process is suspended until a message is received at that exchange. At that time, the caller's task will be placed on the Ready queue and executed in priority order with other ready tasks. If the first 4-byte value is a valid pointer, less than 8000000h, it is possibly a pointer to a request block that was responded to. This is an agreed upon convention for the convenience of the programmer. Of course, you would have had to make the request in the first place, so this would be obvious to you. You've got to send mail to get mail.

Parameters:

`dError`—Returns 0 unless a fatal kernel error occurs.

`dExch`—The exchange to wait for a message.

`pqMsg`—A pointer to where the message will be returned, an array of two dwords.

16

MMURTL Sample Software

While working with MMURTL, most of my efforts have been directed at the operating system itself. I wrote literally hundreds of little programs to test detailed aspects of MMURTL. Only in the last year or so have I made any real effort to produce even the simplest of applications or externally executable programs.

One of the first programs was a simple command-line interpreter. It has since been expanded, and its real purpose is to illustrate certain job-management aspects of MMURTL. However, it does work and it's useful in its own right. It's still missing some important pieces (at least, some that are important to me), but it's a good start.



Other test programs that may be useful to you have been included on the CD-ROM that accompanies this book. Some of these programs have their source code listed and their operation discussed in this chapter. To see what has been included on the CD-ROM, see "What's On the CD-ROM."

Command-Line Interpreter (CLI)

The MMURTL command-line interpreter (called CLI) is a program that accepts commands and executes them for you.

The CLI screen is divided into two sections. The top line is the status line. It provides information such as date, time, CLI version, job number for this CLI, and your current path.

The rest of the screen is an interactive display area that will contain your command prompt. The command prompt is a greater-than symbol (>) followed by a reverse video line. This line is where you enter commands to be executed by the CLI.



The CLI contains some good examples of using a wide variety of the job-management functions of the MMURTL operating system. The source code to the CLI is listed in this chapter and on the CD-ROM.

Internal Commands

The CLI has several built-in commands and can also execute external commands (load and execute .RUN files).

The internal commands are as follows:

`Cls`	Clear screen
`Copy`	Copy a file
`Dir`	Directory listing
`Debug`	Enter debugger
`Del`	Delete a file
`Dump`	Hex dump of a file

`Exit`	Exit and terminate CLI (return to monitor)
`Help`	Display list of internal commands
`Monitor`	Return to monitor (leave this CLI running)
`MD`	Make directory
`Path`	Set file access path (new path, for example, D:\DIR\)
`RD`	Remove directory
`Rename`	Rename a file
`Run`	Execute a RUN file (replacing this CLI)
`Type`	Type the contents of text file to the screen

Each of these commands, along with an example of its use, is described in the following sections.

Cls (Clear Screen)

The `Cls` command removes all the text from the CLI screen and places the command prompt at the top of the screen.

Copy (Copy a File)

The `Copy` command makes a copy of a file with a different name in the same directory, or with the same name in a different directory. If a file exists with the same name, you are prompted to confirm that you want to overwrite it. The command format is as follows:

```
>Copy THISFILE.TXT THATFILE.TXT <Enter>
```

Wildcards (pattern matching) and default names (leaving one parameter blank) are not supported in the first version of the CLI. Both names must be specified. The current path will be properly used if a full file specification is not used.

Dir (Directory Listing)

The `Dir` command lists all the files for your current path or a path that you specify, as follows:

```
>Dir C:\SOURCE\ <Enter>
```

The listing fills a page and prompts you to press Enter for the next screen full. Pressing Esc will stop the listing. The listing is in multiple columns, as follows:

```
FILENAME SIZE DATE TIME TYPE FIRST_CLUSTER
```

The first-cluster entry is a hexadecimal number used for troubleshooting the file system. The Type is the attribute value taken directly from the MS-DOS directory structure. The values are as follows:

00 or 20 is a normal file.

01 or 21 is a read-only file.

02 or 22 is a hidden file.

04 or 24 is a system file.

10 is a subdirectory entry.

08 is a volume name entry (found only in the root).

File values greater than 20 indicate that the archive attribute is set. It means the file has been modified or newly created since the archive bit was last reset.

Debug (Enter Debugger)

The `Debug` command enters MMURTL's built-in Debugger. To exit the Debugger, press the Esc key. For more information on the Debugger, see Chapter 12, "The Debugger."

Del (Delete a File)

The `Del` command deletes a single file from a disk device. Here's an example:

```
>Del C:\TEST\FILE.TXT <Enter>
```

Dump (Hex Dump of a File)

The `Dump` command dumps the contents of a file to the screen in hexadecimal and ASCII format, as follows (16 bytes per line):

```
ADDRESS  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 xxxxxxxxxxxxxxxx
```

The address is the offset in the file. A new line is displayed for each 16 bytes. The text from the line is displayed following the hexadecimal values. If the character is not ASCII text, a period is displayed instead. `Dump` pauses after each screen full of text.

Exit (Exit the CLI)

The `Exit` command exits this CLI and terminates this job. The keyboard and video are assigned to the monitor if this was the active job.

Help (List Internal Commands)

The `Help` command opens the text file called Help.CLI (if it exists) and displays it on the screen. The file should be located in the MMURTL system directory. This is an ASCII text file, and you may edit it as needed.

Monitor (Return to Monitor)

If this is the active CLI (the one displayed), the `Monitor` command will assign the keyboard and video back to the monitor. This CLI will still be executing.

MD (Make Directory)

The `MD` command creates an empty directory in the current path or the path specified on the command line, for example:

```
>MD OLDSTUFF <Enter>

>MD C:\SAMPLES\OLDSTUFF <Enter>
```

The directory tree up to the point of the new directory must already exist. In other words, the tree will be extended only one level.

Path (Set File-Access Path)

The `Path` command sets the file-access path for this job.

Do not confuse the term *path* with the MS-DOS version of the `Path` command. In MMURTL, a job's path is simply a prefix used by the file system to make complete file specifications from a filename. Unlike a single-tasking operating system, no system-wide concept of a "current drive" or "current directory" can be assumed. Each job control block maintains a context for its particular job.

With any file-system operation (opening, renaming, listing, and so on), if you don't specify a full file specification (including the drive), the file system appends your specification to the current path for this job.

The path you set in the CLI will remain the path for any job that is run or any external command that is executed for this CLI. The path may also be set programmatically (by a program or utility). Don't be surprised if you return from a program you have executed to the CLI with a different path. The current path is displayed on the status line in the CLI. Each CLI (job) has its own path, which is maintained by the operating system in the job control block.

The path must end with the backslash. Here are some examples:

```
>Path D:\DIR\ <Enter>

>PATH C:\ <Enter>

>C:\MMSYS\ <Enter>
```

RD (Remove Directory)

The `RD` command removes an empty directory in the current path or in the path specified on the command line:

```
>RD OLDSTUFF <Enter>

>RD C:\MMURTL\OLDSTUFF <Enter>
```

The directory must be empty. The only entries allowed are the two default entries (. and ..)

Rename (Rename a File)

The Rename command changes the name of a file. If the new filename exists, you are prompted to either overwrite existing file or cancel the command. The file must not be open under any other job. The command format is as follows:

```
>Rename OLDNAME.TXT NEWNAME.TXT <Enter>
```

Run (Execute a .RUN File)

The Run command executes the named RUN file in place of this CLI (same job number). The path remains the same as was set in the CLI. Here are some examples:

```
>Run C:\NewEdit\NewEdit.run New.Txt <Enter>

>Run D:\MMDEV\Test.run <Enter>
```

Any parameters specified after the RUN-file name are passed to the RUN file.

Type (View a Text File on the Screen)

The Type command displays the contents of a text file to the screen one page at a time, pausing after each page. Here's an example:

```
>Type C:\MMSYS\Commands.CLI <Enter>
```

External Commands

External commands are those not built into the CLI. These are RUN files (MMURTL executable files with the file suffix .RUN).

MMURTL looks for external commands in three places (in this order):

1. Your current path (shown on the status line)
2. The MMSYS directory on the system disk
3. The file COMMANDS.CLI

Commands.CLI File

The file COMMANDS.CLI must be located in the \MMSYS directory on your system disk. The format of this ASCII text file is discussed forthwith.

The contents of the file COMMANDS.CLI tells the CLI what RUN file to execute for a command name. It is a simple text file with one command per line. The first item on the line is the command name. No spaces or tabs are allowed before the command name, which is followed with the full file specification for the RUN file to execute. An example of COMMANDS.CLI follows:

```
;Any line beginning with semicolon is a comment.
EDIT     C:\MSAMPLES\EDITOR\Edit.run
Print    C:\MSamples\Print\Print.run
CM32     C:\CM32M\CM32.run
DASM     D:\DASMM\DASM.run
;End of Command.cli
```

At least one space or tab must be between the command name and the RUN file. There must also be at least one space or tab between the RUN-file name and the arguments (if they exist). Additional spaces and tabs are ignored.

Any parameters you specify on the command line in the CLI are passed to the RUN file that is executed. You may also add parameters to the line in COMMANDS.CLI. In this case, the parameters you specified on the command line are appended to those you have in the COMMANDS.CLI file.

Global "Hot Keys"

The system recognizes certain keys that are not passed to applications to perform system-wide functions. Global hot keys are those that are pressed while the Ctrl and Alt keys are held down. The CLI doesn't recognize any hot keys, but other applications might.

Ctrl-Alt-Page Down Switches video and keyboard to next job or the monitor. This key sequence is received and acted upon by the Monitor program.

Ctrl-Alt-Del Terminates the currently displayed job. The monitor and the Debugger will not terminate using this command. This key sequence is received and acted upon by the Monitor program.

CLI Source Listing

The following is the source code for version 1.0 of the Command Line Interpreter. The Include files from the OS Source directory contain the ANSI C prototype for the operating system functions.

Listing 16.1. The source code for the Command Line Interpreter Version 1.0.

```
#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char
#define TRUE 1
#define FALSE 0


#include <stdio.h>
```

continues

Listing 16.1. continued

```
#include <string.h>
#include <ctype.h>

/* Includes for OS public calls and structures */

#include "\OSSOURCE\MKernel.h"
#include "\OSSOURCE\MMemory.h"
#include "\OSSOURCE\MData.h"
#include "\OSSOURCE\MTimer.h"
#include "\OSSOURCE\MVid.h"
#include "\OSSOURCE\MKbd.h"
#include "\OSSOURCE\MJob.h"
#include "\OSSOURCE\MFiles.h"
#include "\OSSOURCE\Status.h"

/******************* BEGIN DATA *******************/

/* Define the Foreground and Background colors */

#define EDVID   BLACK¦BGWHITE
#define NORMVID WHITE¦BGBLACK
#define STATVID BLACK¦BGCYAN

long iCol, iLine;

/* aStatLine is exactly 80 characters in length */

char aStatLine[] =
"mm/dd/yy  00:00:00  CLI  V1.0    Job     Path:                                  ";

char aPath[70];
long cbPath;
char fUpdatePath;

char sdisk, syspath[50];    /* holds system disk and system path */
char hlppath[60];           /* Path to help file */
char clipath[60];           /* Path to CLI */
char cmdpath[60];           /* Path to command file */

char aCmd[80];              /* Command line */
long cbCmd = 0;

unsigned char Buffer[512];
unsigned char bigBuf[4096];
char text[70];

char ExitChar;

unsigned long GPExch;       /* Messaging for Main */
unsigned long GPMsg[2];
unsigned long GPHnd1;

unsigned long StatExch; /* Messaging for status task */
long StatStack[256];    /* 1024 byte stack for stat task */

long time, date;
```

```
long JobNum;

/* array of internal commands for parsing */

#define NCMDS 16

char paCmds[NCMDS+1][10] = {
    "",                 /* 0 external */
    "xxxxx",            /* 1 - not used */
    "Cls",              /* 2  */
    "Copy",             /* 3  */
    "Del",              /* 4  */
    "Dir",              /* 5  */
    "Debug",            /* 6  */
    "Dump",             /* 7  */
    "Exit",             /* 8  */
    "Help",             /* 9  */
    "MD",               /* 10 */
    "Monitor",          /* 11 */
    "Path",             /* 12 */
    "RD",               /* 13 */
    "Ren",              /* 14 */
    "Run",              /* 15 */
    "Type",             /* 16 */
    };

#define EXTCMD 0        /* External Command */
#define RESVDCMD 1
#define CLSCMD  2
#define COPYCMD 3
#define DELCMD  4
#define DIRCMD  5
#define DBGCMD  6
#define DUMPCMD 7
#define EXITCMD 8
#define HELPCMD 9
#define MAKEDIR 10
#define MONCMD  11
#define PATHCMD 12
#define REMDIR  13
#define RENCMD  14
#define RUNCMD  15
#define TYPECMD 16


long CmdNum = 0;

char *apParam[13];      /* Param 0 is cmd name */
long acbParam[13];
#define nParamsMax 13

#define ErcBadParams 80

/* Directory Entry Records, 16 records 32 bytes each */

struct dirstruct {
    U8  Name[8];
    U8  Ext[3];
```

continues

Listing 16.1. continued

```
    U8  Attr;
    U8  Rsvd[10];
    U16 Time;
    U16 Date;
    U16 StartClstr;
    U32 FileSize;
    } dirent[16];

/************************ BEGIN CODE ******************/

/*********************************************************
    This is the status task for the CLI.  It updates
    the status line, which has the time and path.
    It runs as a separate task and is started each
    time the CLI is executed.
*********************************************************/

void StatTask(void)
{
    for(;;)
    {
        if (fUpdatePath)
        {
            if (cbPath > 30)
                cbPath = 30;
            FillData(&aStatLine[47], 30, ' ');
            CopyData(aPath, &aStatLine[47], cbPath);
            fUpdatePath = 0;
        }


        GetCMOSDate(&date);
        aStatLine[0] = '0' + ((date >> 20) & 0x0f); /* month */
        aStatLine[1] = '0' + ((date >> 16) & 0x0f);
        aStatLine[3] = '0' + ((date >> 12) & 0x0f); /* Day */
        aStatLine[4] = '0' + ((date >> 8)  & 0x0f);
        aStatLine[6] = '0' + ((date >> 28) & 0x0f); /* year */
        aStatLine[7] = '0' + ((date >> 24) & 0x0f);

        GetCMOSTime(&time);
        aStatLine[10] = '0' + ((time >> 20) & 0x0f);
        aStatLine[11] = '0' + ((time >> 16) & 0x0f);
        aStatLine[13] = '0' + ((time >> 12) & 0x0f);
        aStatLine[14] = '0' + ((time >> 8) & 0x0f);
        aStatLine[16] = '0' + ((time >> 4) & 0x0f);
        aStatLine[17] = '0' + (time & 0x0f);

        PutVidChars(0,0, aStatLine, 80, STATVID);

        Sleep(100); /* sleep 1 second */
    } /* forEVER */
}

/*********************************************************
  This displays error code text string if listed and
  NON zero.
*********************************************************/
```

```
void CheckErc (unsigned long erc)
{
char *pSt;
char st[40];

FillData(st, 40, 0);
switch (erc)
{
    case 0: return;
    case 1:   pSt = "End of file";      break;
    case 4:   pSt = "User cancelled";       break;
    case 80:  pSt = "Invalid parameters";    break;
    case 101: pSt = "Out of memory (need more for this)";    break;
    case 200: pSt = "Invalid filename (not correct format)";     break;
    case 201: pSt = "No such drive";    break;
    case 202: pSt = "The name is not a file (it's a directory)"; break;
    case 203: pSt = "File doesn't exist"; break;
    case 204: pSt = "Directory doesn't exist";  break;
    case 205: pSt = "File is ReadOnly"; break;
    case 208: pSt = "File in use"; break;
    case 222: pSt = "Can't rename across drives"; break;
    case 223: pSt = "Can't rename across directories"; break;
    case 226: pSt = "File Already Exists (duplicate name)"; break;
    case 228: pSt = "Root directory is full"; break;
    case 230: pSt = "Disk is full (bummer)"; break;
    case 231: pSt = "Directory is full"; break;

    default:
        sprintf(st, "Error %05d on last command", erc);
        pSt = st;
        break;
}
    printf("%s\r\n", pSt);
}

/*******************************************************
  This simply does a software interrupt 03 (Debugger).
*******************************************************/

void GoDebug(void)
{
;
#asm
    INT 03
#endasm
return;
}

/*******************************************************
    This is called to initialize the screen.
    If we are returning from an external command,
    we do not reset the cursor position. fNew is
    used to determine if we do or not.
*******************************************************/

void InitScreen(int fNew)
```

continues

Listing 16.1. continued

```
{
    SetNormVid(NORMVID);
    GetXY(&iCol, &iLine);
    if (fNew)
    {
        iCol = 0;
        iLine = 2;
        ClrScr();
    }
    SetXY(iCol, iLine);
    PutVidChars(0,0, aStatLine, 80, STATVID);
    return;
}

/*******************************************
 This does a hex dump to the screen of a
 memory area passed in by "pb"
*******************************************/

U32 Dump(unsigned char *pb, long cb, unsigned long addr)
{
U32 erc, i, j, KeyCode;
unsigned char buff[17];

    erc = 0;
    GetXY(&iCol, &iLine);

    while (cb)
    {
        printf("%08x ", addr);
        if (cb > 15) j=16;
        else j = cb;
        for (i=0; i<j; i++)
        {
            printf("%02x ",*pb);
            buff[i] = *pb++;
            if (buff[i] < 0x20)
                buff[i] = 0x2E;
            if (buff[i] > 0x7F)
                buff[i] = 0x2E;
        }
        buff[i+1] = 0;
        printf("%s\r\n", &buff[0]);
        ++iLine;
        if (iLine >= 22)
        {
            SetXY(0,23);
            printf("ESC to cancel, any other key to continue...");
            ReadKbd(&KeyCode, 1);  /* ReadKbd, wait */
            if ((KeyCode & 0xff) == 0x1b)
            {
                return 4;
            }
            InitScreen(TRUE);
            SetXY(0,1);
            iLine =1;
        }
```

```
        if (cb > 15) cb-=16;
        else cb=0;
        addr+=j;
    }
    return erc;
}

/*******************************************************
  This sets up to call the dump routine for each page.
  This expects to find the filename in param[1].
*******************************************************/

long DoDump(void)
{
unsigned long j, k, l, erc, dret, fh;

erc = 0;
if ((apParam[1]) && (acbParam[1]))
{
    if (iLine >= 23)
    {
        ScrollVid(0,1,80,23,1);
        SetXY(0,23);
    }
    erc = OpenFile(apParam[1], acbParam[1], ModeRead, 1, &fh);
    if (!erc)
    {
        j=0;                /* file lfa */
        GetFileSize(fh, &k);
        while ((j<k) && (!erc))
        {
            FillData(Buffer, 512, 0);
            erc = ReadBytes (fh, Buffer, 512, &dret);
            if (k-j > 512)
                l = 512;
            else l=k-j;
            if (erc < 2)
                erc = Dump(Buffer, dret, j);
            j+=512;
        }
        CloseFile (fh);
    }
}
else
    printf("Filename not given\r\n");

return erc;
}

/*******************************************************
  This types a text file to the screen.
*******************************************************/

long DoType(char *pName, long cbName)
{
long i, j, lfa, erc, dret, fh, KeyCode;

erc = 0;
```

continues

Listing 16.1. continued

```
if ((pName) && (cbName))
{

    if (iLine >= 23)
    {
        ScrollVid(0,1,80,23,1);
        SetXY(0,23);
    }
    erc = OpenFile(pName, cbName, 0, 1, &fh);
    if (!erc)
    {
        FillData(Buffer, 512, 0);
        dret = 1;
        while ((erc<2) && (dret))
        {
            GetFileLFA(fh, &lfa);
            erc = ReadBytes (fh, Buffer, 78, &dret);
            i = 1;
            while ((Buffer[i-1] != 0x0A) && (i < dret))
            {
                i++;
            }
            for (j=0; j<=i; j++)
            {
                if ((Buffer[j] == 0x09) || (Buffer[j] == 0x0d) ||
                    (Buffer[j] == 0x0a))
                   Buffer[j] = 0x20;
            }
            if (dret)
            {
                PutVidChars(0, iLine, Buffer, i, NORMVID);
                iLine++;
            }
            SetXY(0,iLine);
            SetFileLFA(fh, lfa+i);
            if (iLine >= 22)
            {
                SetXY(0,23);
                printf("ESC to cancel, any other key to continue...");
                ReadKbd(&KeyCode, 1);  /* ReadKbd, wait */
                if ((KeyCode & 0xff) == 0x1b)
                  return(4);
                InitScreen(TRUE);
                SetXY(0,1);
                iLine =1;
            }
        }
        printf("\r\nError: %d\r\n", erc);
        CloseFile (fh);
    }
}
else
    printf("Filename not given\r\n");

return erc;
}
```

```
/*****************************************************
  This reads DOS FAT date & time and converts it into
  strings with the format  MM/DD/YY  HH/MM/SS.
  This is used by the directory listing.
*****************************************************/
void CnvrtFATTime(U16 time,
                  U16 date,
                  char *pTimeRet,
                  char *pDateRet)
{
U32 yr,mo,da,hr,mi,se;
char st[15];

    yr = ((date & 0xFE00) >> 9) + 1980 - 1900;
    if (yr > 99) yr -=100;
    mo = (date & 0x01E0) >> 5;
    da = date & 0x001F;
    hr = (time & 0xF800) >> 11;
    mi = (time & 0x07E0) >> 5;
    se = (time & 0x001F) * 2;
    sprintf(st, "%02d:%02d:%02d",hr,mi,se);
    CopyData(st, pTimeRet, 8);
    sprintf(st, "%02d-%02d-%02d",mo,da,yr);
    CopyData(st, pDateRet, 8);

}

/*****************************************************
  Copy a single file with overwrite checking.
  Use block mode for speed. This uses a conservative
  4KB buffer. This could be made a good deal faster by
  allocating a large chunk of memory and using it
  instead of the 4KB static buffer.
*****************************************************/

U32 CopyFile(char *pFrom, U32 cbFrom, char *pTo, U32 cbTo)
{
U32 fhTo, fhFrom, bytesWant, bytesGot, bytesLeft, erc,
    bytesOut, size, dLFA, KeyCode;

    erc = OpenFile(pFrom, cbFrom, ModeRead, 0, &fhFrom);
    if (!erc)
    {
        /* Check to see if it exists already */

        erc = CreateFile(pTo, cbTo, 0);
        if ((!erc) || (erc==ErcDupName))
        {
            if (erc == ErcDupName)
            {
                printf("File already exists. Overwrite? (Y/N)\r\n");
                ReadKbd(&KeyCode, 1);  /* ReadKbd, wait */
                if (((KeyCode & 0xff) == 'Y') || ((KeyCode & 0xff) == 'y'))
                {
                    erc = 0;
                }
```

continues

Listing 16.1. continued

```
            }

            if (!erc)
            {
                erc = OpenFile(pTo, cbTo, ModeModify, 0, &fhTo);
                if (!erc)
                {
                    /* both files are open in block mode */

                    erc = GetFileSize(fhFrom, &size);
                    if (!erc)
                        erc = SetFileSize(fhTo, size);

                    dLFA = 0;
                    bytesLeft = size;
                    while ((!erc) && (bytesLeft))
                    {   if (bytesLeft >= 4096)
                            bytesWant = 4096;
                        else
                            bytesWant = bytesLeft;
                        if (bytesWant & 511)    /* handle last sector */
                            bytesWant += 512;
                        bytesWant = (bytesWant/512) * 512;
                        erc = ReadBlock(fhFrom, bigBuf, bytesWant,
                                        dLFA, &bytesGot);
                        if (bytesGot)
                        {
                            erc = WriteBlock(fhTo, bigBuf, bytesGot,
                                             dLFA, &bytesOut);
                            if (bytesLeft < bytesOut)
                                bytesLeft = 0;
                            else
                                bytesLeft-=bytesOut;
                        }
                        dLFA += bytesGot;
                    }
                    CloseFile(fhTo);
                }
            }
        }
        CloseFile(fhFrom);
    }
    return(erc);
}

/*******************************************************
  Does a directory listing for param [1] or current path.
*******************************************************/

U32 DoDir(void)
{
U8 fDone;
U32 SectNum, erc, KeyCode, i;
char st[78];

if (iLine >= 23)
{
```

```
    ScrollVid(0,1,80,23,1);
    SetXY(0,23);
}
fDone = 0;
SectNum = 0;
while (!fDone)
{
    erc = GetDirSector(apParam[1], acbParam[1], &dirent[0], 512, SectNum);
    if (!erc)
    {
        for (i=0; i<16; i++)
        {
            if (!dirent[i].Name[0])
            {
                erc = 1;
                fDone = 1;
            }
            if ((dirent[i].Name[0]) && (dirent[i].Name[0] != 0xE5))
            {
                sprintf(st, "%8s %3s  %8d  xx/xx/xx xx/xx/xx  %2x   %04x\r\n",
                        dirent[i].Name,
                        dirent[i].Ext,
                        dirent[i].FileSize,
                        dirent[i].Attr,
                        dirent[i].StartClstr);
                CnvrtFATTime(dirent[i].Time, dirent[i].Date, &st[33], &st[24]);
                printf("%s", st);
                iLine++;
                if (iLine >= 22)
                {
                    SetXY(0,23);
                    printf("ESC to cancel, any other key to continue...");
                    ReadKbd(&KeyCode, 1);  /* ReadKbd, wait */
                    if ((KeyCode & 0xff) == 0x1b)
                    {
                        erc = 4;
                        fDone = TRUE;
                    }
                    InitScreen(TRUE);
                    SetXY(0,1);
                    iLine =1;
                }
            }
        }
    }
    else
        fDone = TRUE;
    SectNum++;
}

return(erc);
}

/*******************************************************
  This fills in the pointers to and sizes of each parameter
  found on the command line in the arrays apParams and
  acbParams.
*******************************************************/
```

continues

Listing 16.1. continued

```
void ParseCmdLine(void)
{
long iCmd, iPrm, i;     /* iCmd is index to aCmd */
    iCmd = 0;
    for (iPrm=0; iPrm<nParamsMax; iPrm++)
    {
        acbParam[iPrm] = 0;     /* default the param to empty */
        apParam[iPrm] = 0;      /* Null the ptr */
        if (iCmd < cbCmd)
        {
            if (!isspace(aCmd[iCmd]))
            {
                apParam[iPrm] = &aCmd[iCmd++];  /* ptr to param */
                i = 1;
                while ((iCmd < cbCmd) && (!isspace(aCmd[iCmd])))
                {
                    i++;
                    iCmd++;
                }
                acbParam[iPrm] = i;     /* size of param */
            }
            while ((iCmd < cbCmd) && (isspace(aCmd[iCmd])))
                iCmd++;
        }
    }
}

/***********************************************
  This opens and parses Commands.CLI in the system
  directory and looks for the command name in
  apParam[0]. If it finds it, it places the full
  filespec for the run file in "runfile" for the
  caller, then fixes up the aCmd command line
  just as if it had come from the user.
  The format for a line entry in Commands.CLI is:
  name fullspec param param param...
***********************************************/

long GetCLICommand(char *runfile)
{
long i, j, k;
FILE *f;
char rawline[90];   /* used to build a pseudo command line */
char cmdline[90];   /* used to build a pseudo command line */
char fdone;

    cmdline[0] = 0;

    f = fopen(cmdpath, "r");
    if (f)
    {
        fdone = 0;
        while (!fdone)
        {
            if (fgets(rawline, 89, f))
            {
```

```
            if (rawline[0] == ';')  /* a comment line */
                continue;
            j = CompareNCS(apParam[0], rawline, acbParam[0]);
            if (j == -1)
            {   /* we found a match for the command */
                /* Move the command name into the command line */
                i = 0; /* Index into rawline */
                j = 0; /* Index into cmdline we are building */
                k = 0; /* Index into runfile name we return to caller */
                while (!(isspace(rawline[i])))
                    cmdline[j++] = rawline[i++];
                /* follwed by a space */
                cmdline[j++] = ' ';

                /* skip whitespace in rawline */
                while (isspace(rawline[i]))
                    i++;

                /* now move runfile name */
                while (!(isspace(rawline[i])))
                    runfile[k++] = rawline[i++];
                runfile[k] = 0; /* null terminte */

                /* skip whitespace in rawline */
                while (isspace(rawline[i]))
                    i++;
                /* now move arguments if any, and LF */
                while (rawline[i])
                    cmdline[j++] = rawline[i++];
                cmdline[j] = 0;

                /* Move cmd line we built to real cmd line */
                strcpy(aCmd, cmdline);
                cbCmd = strlen(aCmd);
                return(1);      /* tell em we got it! */
            }
        }
        else
            fdone = 1;
    }
  }
  else
      printf("Commands.CLI not found.\r\n");

  return(0);
}


/************************************************
  When a command is specified to the CLI and it
  is not an internal command, we first look for
  a .RUN file in the current path. If we don't
  find one, we go to the system directory, if
  it's not there, we then go to COMMANDS.CLI
  and search it line by line to see if a run
  file is specified there. If so, we set the
  command line with everything after the run file
```

continues

Listing 16.1. continued

```
  specified and try to execute it.
*************************************************/

void FindRunFile(void)
{
char runfile[80];
long i, erc, fh;

    /* Try to find in current path */

    CopyData(apParam[0], runfile, acbParam[0]);
    runfile[acbParam[0]] = 0;
    strcat(runfile, ".RUN");

    erc = OpenFile(runfile, strlen(runfile), ModeRead, 1, &fh);
    if (!erc)
    {          /* found a run file */
        CloseFile(fh);
        SetCmdLine(aCmd, cbCmd);
        SetExitJob(clipath, strlen(clipath));
        erc = Chain(runfile, strlen(runfile), 0);
    }
    else    /* Try to find in System directory */
    {
        strcpy(runfile, syspath);
        i = strlen(runfile);
        CopyData(apParam[0], &runfile[i], acbParam[0]);
        runfile[acbParam[0]+i] = 0; /* null terminate */
        strcat(runfile, ".RUN");
        erc = OpenFile(runfile, strlen(runfile), ModeRead, 1, &fh);
        if (!erc)
        {          /* found a run file */
            CloseFile(fh);
            SetCmdLine(aCmd, cbCmd);
            SetExitJob(clipath, strlen(clipath));
            erc = Chain(runfile, strlen(runfile), 0);
        }
         /* now we call GetCLICommand as a last resort */
        else if (GetCLICommand(runfile))
        {
            erc = OpenFile(runfile, strlen(runfile), ModeRead, 1, &fh);
            if (!erc)
            {          /* found a run file */
                CloseFile(fh);
                SetCmdLine(aCmd, cbCmd);
                SetExitJob(clipath, strlen(clipath));
                erc = Chain(runfile, strlen(runfile), 0);
            }
        }
    }
    printf("Command not found\r\n");
}

/*******************************************
   Main function entry point.
   Note: No params to CLI.
*******************************************/
```

```
void main(void)
{
unsigned long erc, i, j, fh;

    erc = AllocExch(&StatExch);
    erc = AllocExch(&GPExch);

    SpawnTask(&StatTask, 24, 0, &StatStack[255], 0);

    GetJobNum(&JobNum);
    sprintf(&aStatLine[37], "%02d", JobNum);

    SetJobName("CLI V1.0", 8);

    /* Get system disk and set up path names for
    cli, command file, and help file.
    */

    GetSystemDisk(&sdisk);
    sdisk &= 0x7F;
    sdisk += 0x41;      /* 0=A, 1=B, 2=C etc. */
    syspath[0] = sdisk;
    syspath[1] = ':';
    syspath[2] = 0;
    strcat(syspath, "\\MMSYS\\");

    strcpy(clipath, syspath);
    strcat(clipath,"CLI.RUN");

    strcpy(cmdpath, syspath);
    strcat(cmdpath,"COMMANDS.CLI");

    strcpy(hlppath, syspath);
    strcat(hlppath,"HELP.CLI");

    /* If a path was already set we assume that we are reloading
       after an external command has been run. We do not
       want to reset the screen completely so the user can
       see anything the external command left on the screen.
    */

    cbPath = 0;
    GetPath(JobNum, aPath, &cbPath);
    if (cbPath)
    {
        GetXY(&iCol, &iLine);
        if (iLine == 0) /* under status line */
            iLine = 1;
        InitScreen(FALSE);
        SetXY(0, iLine);
    }
    else
    {
        strcpy (aPath, syspath);
        cbPath = strlen(syspath);
        SetPath(syspath, strlen(syspath));
```

continues

Listing 16.1. continued

```
        InitScreen(TRUE);
    }

    fUpdatePath = 1;

    SetExitJob(clipath, strlen(clipath));

    while (1)
    {
        FillData(aCmd, 79, ' ');
        aCmd[79] = 0;
        cbCmd = 0;
        SetXY(0, iLine);
        TTYOut (">", 1, NORMVID);
        EditLine(aCmd, cbCmd, 79, &cbCmd, &ExitChar, EDVID);
        TTYOut ("\r\n", 2, NORMVID);
        GetXY(&iCol, &iLine);

        acbParam[0] = 0;
        apParam[0] = 0;
        CmdNum = 0;

        if (ExitChar == 0x0d)
            ParseCmdLine();         /* set up all params */

        if ((acbParam[0]) && (apParam[0]))
        {   /* It's a command! */

            i = 1;
            while (i <= NCMDS)
            {
                j = strlen(paCmds[i]);
                if ((acbParam[0] == j) &&
                    (CompareNCS(apParam[0], paCmds[i], j) == -1))
                {
                    CmdNum = i;
                    break;
                }
                i++;
            }
            erc = 0;
            switch (CmdNum)
            {
                case EXTCMD:            /* external command */
                    FindRunFile();
                    break;
                case CLSCMD:
                    InitScreen(TRUE);
                    break;
                case COPYCMD:
                    if ((acbParam[1]) && (acbParam[2]))
                        erc = CopyFile(apParam[1], acbParam[1],
                                       apParam[2], acbParam[2]);
                    else erc = ErcBadParams;
                    break;
                case DELCMD:
```

```
        erc = OpenFile(apParam[1], acbParam[1], 1, 0, &fh);
        if (!erc)
            erc = DeleteFile(fh);
        if (!erc)
            printf("Done.\r\n");
        break;
    case DIRCMD:
        erc = DoDir();
        break;
    case DBGCMD:
        GoDebug();
        break;
    case DUMPCMD:
        erc = DoDump();
        break;
    case EXITCMD:
        SetExitJob("", 0);
        ExitJob(0);
        break;
    case HELPCMD:
        erc = DoType(hlppath, strlen(hlppath));
        break;
    case MAKEDIR:
        erc = CreateDir(apParam[1], acbParam[1]);
        break;
    case MONCMD:
        erc = Request("KEYBOARD", 4, GPExch, &GPHndl, 0,
                      0, 0,   0, 0,   1, 0, 0);
        erc = WaitMsg(GPExch, GPMsg);
        SetVidOwner(1);
        break;
    case PATHCMD:
        erc = SetPath(apParam[1], acbParam[1]);
        if (!erc)
            erc = GetPath(JobNum, aPath, &cbPath);
        fUpdatePath = 1;
        break;
    case REMDIR:
        erc = DeleteDir(apParam[1], acbParam[1]);
        break;
    case RENCMD:
        if ((acbParam[1]) && (acbParam[2]))
            erc = RenameFile(apParam[1], acbParam[1],
                             apParam[2], acbParam[2]);
        else erc = ErcBadParams;
        break;
    case RUNCMD:
        if (acbParam[1])
        {
            i = 2;
            while (aCmd[i] != ' ')
                i++;
            SetCmdLine(&aCmd[i], cbCmd-i);
            SetExitJob(clipath, strlen(clipath));
            erc = Chain(apParam[1], acbParam[1], 0);
        }
        break;
    case TYPECMD:
```

continues

Listing 16.1. continued

```
                    erc = DoType(apParam[1], acbParam[1]);
                    break;
                default:
                    break;
            }
            CheckErc(erc);
        }

        GetXY(&iCol, &iLine);
        if (iLine >= 23)
        {
            ScrollVid(0,1,80,23,1);
            SetXY(0,23);
        }
    }
}
```

A Simple Editor



The editor included with MMURTL is a very simple in-memory editor. The editor always word-wraps lines that extend beyond 79 columns. The largest file that can be edited is 128K. The source code is listed forthwith and is also included on the accompanying CD-ROM. The editor contains a good example of extensive use of the keyboard service (translating keycodes).

Editor Screen

The editor screen is divided into three sections. The top line provides continuous status information, such as the position of the cursor in the file (column and line), current filename, and typing mode (Insert or Overtype). The next 23 lines are the text editing area. The last line is used for data entry, status, and error display.

Editor Commands

The following four tables describe all the editor commands.

Table 16.1. File management and general commands.

Keystroke	*Action*
Alt-S	Saves changes to current file
Alt-C	Closes and prompts to save current file
Alt-O	Open a new file
Alt-Q	Quits (exits) the editor
Alt-X	Same as Alt-Q (exit)

Keystroke	*Action*
Insert	Toggles between insert and overtype mode
Esc	Exits filename entry mode for open-file command
Alt-V	Make nontext chars visible/invisible (toggle)

Table 16.2. Cursor and screen-management commands.

Keystroke	*Action*
PageDown	Move one screen down in the text file
PageUp	Move one screen up in the text file
Up	Cursor up one line (scroll down if needed)
Down	Cursor down one line (scroll up if needed)
Left	Cursor left one column (no wrap)
Right	Cursor right one column (no wrap)
Alt-B	Go to beginning of text
Alt-E	Go to end of text
Alt-Up Arrow	Cursor to top of screen
Alt-Down Arrow	Cursor to bottom of screen
Alt-Left Arrow	Cursor to beginning of line
Alt Right Arrow	Cursor to end of line
Home	Cursor to beginning of line
End	Cursor to end of line
Shift Right Arrow	Move cursor right five spaces
Shift Left Arrow	Move cursor left five spaces

Table 16.3. Block-selection and editing commands.

Keystroke	*Action*
F3	Begin block (mark)
F4	End block (bound)
F2	Unmark block (no block defined or highlighted)
F9	Move marked block to current cursor position
F10	Copy marked block to current cursor position
Alt-Del	Delete marked block

Table 16.4. Miscellaneous editing keys.

Keystroke	*Action*
Delete	Delete character at current cursor position
Backspace	Destructive in insert mode (INS)
	Non-destructive in overtype mode (OVR)
Tab	Pseudo tab every four columns (space filled)

Editor Source Listing

The editor source is a good example of a program that uses a lot of functionality from the keyboard service. File system calls are also used in place of equivalent C library functions, which provides good examples of checking for file system errors. Listing 16.2 is the editor source code.

Listing 16.2. Editor source code.

```
/* Edit.c  A simple editor using MMURTL file system and
keyboard services */

#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char

#define TRUE 1
#define FALSE 0

#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* Includes for OS public calls and structures */

#include "\OSSOURCE\MKernel.h"
#include "\OSSOURCE\MMemory.h"
#include "\OSSOURCE\MData.h"
#include "\OSSOURCE\MTimer.h"
#include "\OSSOURCE\MVid.h"
#include "\OSSOURCE\MKbd.h"
#include "\OSSOURCE\MJob.h"
#include "\OSSOURCE\MFiles.h"

#define EDVID   BRITEWHITE¦BGBLUE
#define NORMVID WHITE¦BGBLACK
#define MARKVID WHITE¦BGRED
#define STATVID BLACK¦BGCYAN

#define EMPTY   99999
```

```
#define NLINESMAX 26

struct EditRecType {
    U8      *pBuf;
    U8      *pBufWork;          /* For copy and move */
    U32     Line[NLINESMAX];    /* Offset in buf for 1st char in line */
    U32     iBufMax;            /* sBuf - 1 */
    U32     iColMin;            /* Screen coords */
    U32     iRowMin;
    U32     iColMax;
    U32     iRowMax;
    U32     sLine;
    U8      bSpace;
    U8      fVisible;
    U32     iAttrMark;
    U32     iAttrNorm;
    U32     iTabNorm;
    U32     oBufLine0   /* oBufLine0 */
    U32     iCol;       /* cursor, 0..sLine-1 */
    U32     iLine;      /* cursor, 0..cLines-1 */
    U32     oBufInsert; /* offset of next char in */
    U32     oBufLast;   /* offset+1 of last char */
    U32     oBufMark;
    U32     oBufBound;
    };

struct EditRecType  EdRec;
struct EditRecType  *pEdit;
char *pBuf1, *pBuf2;
unsigned char b, b1;
long  erc, fh;
char fModified;
char fOvertype;
char aStat[80];
char aStat1[80];
char aCmd[80];          /* Get our command line */
long cbCmd = 0;
char *apParam[13];      /* Param 0 is cmd name */
long acbParam[13];
#define nParamsMax 13
char Filename[60];
long cbFilename;
unsigned char filler[100];


void clearbuf(void);    /* prototype for forward usage */

/*******************************************************
  Displays errors if they occur for certain file operations.
*******************************************************/

long CheckErc(long call, long erc)
{
char st[40];
long i;

    if (erc) {
```

continues

Listing 16.2. continued

```
        FillData(st, 40, 0);
        Beep();
        switch (call) {
        case 1:
            sprintf(st, "Error %05d occurred on OpenFile", erc);
            break;
        case 2:
            sprintf(st, "Error %05d occurred on ReadBytes", erc);
            break;
        case 3:
            sprintf(st, "Error %05d occurred on WriteBytes", erc);
            break;
        case 4:
            sprintf(st, "Error %05d occurred on CreateFile", erc);
            break;
        case 5:
            sprintf(st, "Error %05d occurred on SetFileSize", erc);
            break;
        case 6:
            sprintf(st, "Error %05d occurred on SetFileLFA", erc);
            break;
        case 7:
            sprintf(st, "Error %05d occurred on ReadKbd", erc);
            break;
        default:
            sprintf(st, "Error %05d occurred on last command", erc);
            break;
        }
        for (i=0; i<40; i++)
            if (!st[i])
                st[i] = ' ';

        PutVidChars (40, 24, st, 39, STATVID);
    }
    return (erc);
}

/*********************************************************
  Clears the status line with 80 blank chars.
*********************************************************/
void ClearStatus(void)
{
char st[80];
    FillData(st, 80, 0);
    PutVidChars (0, 24, st, 80, NORMVID);
}

2
/*************************************************************
   Saves a file you are editing. If fPrompt is true, this
   will prompt you to save. If fClose is true, the file will be closed
   and the buffer will be closed.
*************************************************************/

void SaveFile(int fPrompt, int fClose)
{
```

```
U32 i, keycode, fYes;
unsigned char  *pBuff;

  pBuff = pEdit->pBuf;

    if ((fh) && (fModified))
    {
        if (pEdit->fVisible)
        {    /* fix visible characters */
             for (i=0; i <=pEdit->iBufMax; i++)
                 if (pBuff[i] == 0x07)
                      pBuff[i] = 0x20;
                 pEdit->fVisible = FALSE;
        }
        fYes = 1;
        if (fPrompt)
        {
            ClearStatus();
            SetXY(0, 24);
            TTYOut("This file has been modified. SAVE IT? (Y/N)", 43,
BLACK¦BGCYAN);
            ReadKbd(&keycode, 1);
            if (((keycode & 0xff) == 'N') ¦¦ ((keycode & 0xff) == 'n'))
            {
                fYes = 0;
                ClearStatus();
            }
        }

        if (fYes)
        {
            erc = CheckErc(6, SetFileLFA(fh, 0));
            if (!erc)
                erc = CheckErc(5, SetFileSize(fh,
                            pEdit->oBufLast));
            if (!erc)
                erc = CheckErc(3, WriteBytes (fh, pBuf1,
                             pEdit->oBufLast, &i));
            fModified = 0;
            ClearStatus();
            PutVidChars (0, 24, "DONE...   ", 10, STATVID);
            Sleep(150);
            ClearStatus();
        }

    }

    if (fh && fClose)
    {
        CloseFile(fh);
        fh = 0;
        cbFilename = 0;
        clearbuf();
    }
}

/*************************************************************
   This prompts for a filename to open and opens it if it
```

continues

Listing 16.2. continued

```
   exists. If not, it will prompt to create.
*************************************************************/

void OpenAFile(char *name)
{
U32 filesize, dret, keycode;

    erc = 0;
    cbFilename = 0;
    if (!name)
    {
        SetXY(0,24);
        PutVidChars (0,24, "Filename: ", 10, BLACK¦BGWHITE);
        SetXY(10,24);
        EditLine(Filename, 0, 60, &cbFilename, &b1, BLACK¦BGCYAN);
        SetXY(0,0);
    }
    else
    {
        b1=0x0d;
        strncpy(Filename, name, 13);
        cbFilename = strlen(Filename);
    }
    if ((b1==0x0d) && (cbFilename)) {
        erc = OpenFile(Filename, cbFilename, ModeModify, 1, &fh);
        if (!erc) {
            GetFileSize(fh, &filesize);
            if (filesize < 131000)  /* Buf is 131071 */
            {
                erc = ReadBytes (fh, pBuf1, filesize, &dret);
                if (erc > 1)
                    erc = CheckErc(2, erc);
                else erc = 0;
                pEdit->oBufLast = dret;      /* offset+1 of last char  */
                pBuf1[pEdit->oBufLast] = 0x0F;      /* the SUN */
            }
            else
            {
                CloseFile(fh);
                fh = 0;
                Beep();
                SetXY(50, 24);
                TTYOut("File is too large to edit.", 26, BLACK¦BGCYAN);
                ReadKbd(&keycode, 1);
            }
        }
        else if (erc == 203) {  /* no such file */
            Beep();
            SetXY(50, 24);
            TTYOut("Doesn't exist. Create?? (Y/N)", 29, BLACK¦BGCYAN);
            ReadKbd(&keycode, 1);
            if (((keycode & 0xff) == 'Y') ¦¦ ((keycode & 0xff) == 'y')) {
                erc = CheckErc(4, CreateFile(Filename, cbFilename, 0));

                if (!erc)
                    erc = CheckErc(1, OpenFile(Filename, cbFilename,
```

```
                                    ModeModify, 1, &fh));

                if (erc) {
                    fh = 0;
                    cbFilename = 0;
                }

            }
            else {
                cbFilename = 0;
                ClearStatus();
            }
        }
        else
            CheckErc(1, erc);
    }

    if (!erc)
        ClearStatus();
}

/*************************************************************
   This counts ABSOLUTE LINES from the begining of the buffer
   up to point of oBufLine0 (which is the first char displayed
   in the window). ABSOLUTE means LFs were found, even though
   we word wrap always.
*************************************************************/

unsigned long CountEols (void)
{
unsigned long  nEols, i;
unsigned char  *pBuff;

  pBuff = pEdit->pBuf;
  nEols = 0;
  i = 0;

  while (i < pEdit->oBufLine0) /* count LFs */
    if (pBuff[i++] == 0x0A)
        nEols++;

  return(nEols);
}



/*************************************************************
   This returns the index to the the last character in a line
   up to a maximum of sLine-1. iBuf points to the beginning
   point in the buffer for which to find the end of line.
*************************************************************/

unsigned long findEol (unsigned long iBuf)
{

unsigned long  iEol, iEolMax;
unsigned char  *pBuff;
```

continues

Listing 16.2. continued

```
  pBuff = pEdit->pBuf;

    /* Calculate the most it could be */

  iEolMax = iBuf + pEdit->sLine-1;

    /* Fix it if EOL is past end of data */

  if (iEolMax > pEdit->oBufLast)
     iEolMax = pEdit->oBufLast;

  iEol = iBuf;
  while ((pBuff[iEol] != 0x0A) && (iEol < iEolMax)) /* Find CR */
    iEol++;
  if ((iEol == iEolMax) && (pBuff[iEol] != 0x0A)) {  /* if no CR... */
    iEol = iEolMax;
    if (iEolMax < pEdit->oBufLast) {

      /* now work back to last space */
      while ((pBuff[iEol] != pEdit->bSpace) && (iEol > iBuf))
         iEol--;

      /*  now find first non-space - allows */
      if ((iEol > iBuf) &&
          (pBuff[iEol] == pEdit->bSpace) &&  /* wrap-around w/ double space */
          (iEol == iEolMax)) {

        if ((pBuff[iEol-1] == pEdit->bSpace) ||
            ((iEol == iEolMax) && (pBuff[iEol+1] == pEdit->bSpace))) {
            while ((pBuff[iEol] == pEdit->bSpace) && (iEol > iBuf))
                iEol--;
            while ((pBuff[iEol] != pEdit->bSpace) && (iEol > iBuf))
                iEol--;
        }
      }
      if ((iEol == iBuf) &&
          (pBuff[iBuf] > 0) &&
          (pBuff[iEolMax] > 0))        /* handles "all-char" of full line */
        iEol = iEolMax;
    }
  }
  return(iEol);
}

/*************************************************************
   This walks back through the buffer looking for the
   logical end of a line.
*************************************************************/


unsigned long findPrevLine (unsigned long oBufStart)
{
unsigned long  i, j;
char *pBuff;
```

```
  pBuff = pEdit->pBuf;

  i = 0;
  if (oBufStart)
     i = oBufStart - 1;
  while ((i) && (pBuff[i] != 0x0A))
     i--;
  if (i > 0)
    i--;
  while ((i > 0) && (pBuff[i] != 0x0A))
     i--;
  if (i)
    i++;            /*  Get to known start of line */
  do {
    j = i;
    i = (findEol (j)) + 1;
  }
  while (i < oBufStart);
  return(j);
}

/*********************************************************
   This executes the BEGIN BLOCK (Mark) command.
*********************************************************/


void doMark (unsigned long iLn)
{
unsigned long  iColStart, iColFinish, iMarkLoc, iBoundLoc;

  if (pEdit->oBufMark < EMPTY) {
      if (pEdit->oBufMark <= pEdit->oBufBound) {
      iMarkLoc = pEdit->oBufMark;
      iBoundLoc = pEdit->oBufBound;
     }
     else {
       iMarkLoc = pEdit->oBufBound;
       iBoundLoc = pEdit->oBufMark;
     }
     if ( ((iMarkLoc >= pEdit->Line[iLn]) && (iMarkLoc < pEdit->Line[iLn+1]))   ||
          ((iBoundLoc >= pEdit->Line[iLn]) && (iBoundLoc < pEdit->Line[iLn+1])) ||
          ((iMarkLoc < pEdit->Line[iLn]) && (iBoundLoc >= pEdit->Line[iLn+1])) )
     {
       if (iMarkLoc >= pEdit->Line[iLn])
         iColStart = pEdit->iColMin + iMarkLoc - pEdit->Line[iLn];
       else
         iColStart = pEdit->iColMin;

       if (iBoundLoc < pEdit->Line[iLn+1])
          iColFinish = pEdit->iColMin + iBoundLoc - pEdit->Line[iLn];
       else
          iColFinish = pEdit->iColMin + pEdit->Line[iLn+1] - pEdit->Line[iLn] - 1;

       if (iColStart > pEdit->iColMin)
           PutVidAttrs (pEdit->iColMin,
                        iLn,
                        iColStart-pEdit->iColMin,
```

continues

Listing 16.2. continued

```
                    pEdit->iAttrNorm);
      PutVidAttrs (iColStart, iLn,  iColFinish - iColStart +1, pEdit->iAttrMark);

      if (iColFinish < pEdit->iColMax)
        PutVidAttrs (iColFinish+1,
                  iLn,
                  pEdit->iColMax - iColFinish,
                  pEdit->iAttrNorm);
    }
    else   /*buf col*/
      PutVidAttrs (pEdit->iColMin,
                iLn,
                pEdit->sLine,
                pEdit->iAttrNorm);
  }
}


/*************************************************************
   This inserts data into the main editing buffer.
*************************************************************/


char putInBuf(  unsigned char bPutIn,
                char fOvertype,
                char fSpecInsert)
{
unsigned long  cb;
char  fOK, *pBuff;

    fModified = 1;
    pBuff = pEdit->pBuf;

  if ((pEdit->oBufInsert < pEdit->iBufMax) &&
    ((pEdit->oBufLast < pEdit->iBufMax) ||
     ((fOvertype) && (!fSpecInsert))))
  {
    fOK = 1;
    if ((fOvertype) && (!fSpecInsert)) {
      pBuff[pEdit->oBufInsert] = bPutIn;
      if (pEdit->oBufLast == pEdit->oBufInsert)
         pEdit->oBufLast++;
      pEdit->oBufInsert++;
    }
    else {
      cb = pEdit->oBufLast - pEdit->oBufInsert + 1;
      CopyData (&pBuff[pEdit->oBufInsert], pEdit->pBufWork, cb);
      pBuff[pEdit->oBufInsert] = bPutIn;
      CopyData (pEdit->pBufWork, &pBuff[pEdit->oBufInsert+1], cb);
      pEdit->oBufLast++;
      pEdit->oBufInsert++;
      if (pEdit->oBufMark < EMPTY) {
        if (pEdit->oBufInsert-1 < pEdit->oBufMark)
          pEdit->oBufMark++;
        if (pEdit->oBufInsert-1 <= pEdit->oBufBound)
          pEdit->oBufBound++;
```

```
      }
    }
  }
  else {
    fOK = 0;
    Beep();
    erc = 40400;
  }
  return (fOK);
}

/*************************************************************
   This executes the MOVE command, which moves a marked
   block to the cursor's current location in the file.
*************************************************************/

void moveData (void)
{
unsigned long  i, iMk, iBd;
char *pBuff, *pBuffWork;

    pBuff = pEdit->pBuf;
    pBuffWork = pEdit->pBufWork;

if (pEdit->oBufMark < EMPTY) {
    fModified = 1;
    if (pEdit->oBufMark <= pEdit->oBufBound) {
        iMk = pEdit->oBufMark;
        iBd = pEdit->oBufBound;
    }
    else {
        iBd = pEdit->oBufMark;
        iMk = pEdit->oBufBound;
    }
    if ((pEdit->oBufInsert < iMk) ¦¦ (pEdit->oBufInsert > iBd)) {
        for (i=0; i <= pEdit->oBufLast; i++)
            pBuffWork[i] = pBuff[i];
        if (pEdit->oBufInsert < iMk) {
          for (i=0; i<=iBd-iMk; i++)                /* Move mk/bd */
             pBuff[pEdit->oBufInsert+i] = pBuffWork[iMk+i];
          for (i=0; i<=iMk - pEdit->oBufInsert - 1; i++) /* Shift overwritten ahead
*/
            pBuff[pEdit->oBufInsert+iBd-iMk+1+i] =
            pBuffWork[pEdit->oBufInsert+i];
2        }
        if (pEdit->oBufInsert > iBd) {
            for (i=0; pEdit->oBufInsert - iBd - 1; i++)
                pBuff[iMk+i] = pBuffWork[iBd+1+i];
            pEdit->oBufInsert = pEdit->oBufInsert - iBd + iMk - 1;
            for (i=0; i <=iBd-iMk; i++)
                pBuff[pEdit->oBufInsert+i] = pBuffWork[iMk+i];
        }
        iBd = pEdit->oBufInsert + iBd - iMk;
        iMk = pEdit->oBufInsert;
        if (pEdit->oBufBound > pEdit->oBufMark) {
            pEdit->oBufBound = iBd;
            pEdit->oBufMark = iMk;
```

continues

Listing 16.2. continued

```
        }
        else {
            pEdit->oBufMark = iBd;
            pEdit->oBufBound = iMk;
        }
    }
}
 else Beep();
}

/*************************************************************
   This sets the characters on the screen to normal attributes.
*************************************************************/

void CopyIt (void)
{
unsigned long iMk, iBd;
char *pBuff, *pBuffWork;

    pBuff = pEdit->pBuf;
    pBuffWork = pEdit->pBufWork;

if (pEdit->oBufMark < EMPTY) {
    fModified = 1;

  if (pEdit->oBufMark <= pEdit->oBufBound) {
    iMk = pEdit->oBufMark;
    iBd = pEdit->oBufBound;
  } else {
    iBd = pEdit->oBufMark;
    iMk = pEdit->oBufBound;
  }
  if (pEdit->oBufLast+iBd-iMk+1 < pEdit->iBufMax) {
    CopyData(pBuff, pBuffWork, pEdit->oBufLast+1);
    CopyData(&pBuffWork[iMk], &pBuff[pEdit->oBufInsert], iBd-iMk+1);
    if (pEdit->oBufLast >= pEdit->oBufInsert)
        CopyData(&pBuffWork[pEdit->oBufInsert],
                 &pBuff[pEdit->oBufInsert+iBd-iMk+1],
                 pEdit->oBufLast - pEdit->oBufInsert+1);
    iBd = pEdit->oBufInsert + iBd - iMk;
    iMk = pEdit->oBufInsert;
    pEdit->oBufInsert = pEdit->oBufInsert + iBd - iMk + 1;
    pEdit->oBufLast = pEdit->oBufLast + iBd - iMk + 1;
    if (pEdit->oBufBound > pEdit->oBufMark) {
      pEdit->oBufBound = iBd;
      pEdit->oBufMark = iMk;
    }
    else {
      pEdit->oBufMark = iBd;
      pEdit->oBufBound = iMk;
    }
  }
}
else Beep();
}
```

```
/*********************************************************
   This executes the COPY command, which copies a marked
   block to the cursor's current location in the file.
*********************************************************/

void normAttr (void)
{
unsigned long  i;

  for (i = pEdit->iRowMin; i <= pEdit->iRowMax; i++)
     PutVidAttrs (pEdit->iColMin, i, pEdit->sLine, pEdit->iAttrNorm);
}

/*********************************************************
   This unmarks a selected block. (hides it).
*********************************************************/

void nullMarkBound (void)
{2
  pEdit->oBufMark = EMPTY;
  pEdit->oBufBound = EMPTY;
  normAttr ();
}

/*********************************************************
   This DELETES a selected block.
*********************************************************/

void deleteData (void)
{
unsigned long  i, iMk, iBd;
char  fProb;
char *pBuff, *pBuffWork;

  pBuff = pEdit->pBuf;
  pBuffWork = pEdit->pBufWork;

if (pEdit->oBufMark < EMPTY) {
  fModified = 1;
  if (pEdit->oBufMark <= pEdit->oBufBound)  {
    iMk = pEdit->oBufMark;
    iBd = pEdit->oBufBound;
  } else {
    iBd = pEdit->oBufMark;
    iMk = pEdit->oBufBound;
  }
  if ((pEdit->oBufLine0 >= iMk) && (pEdit->oBufLine0 <= iBd))
     fProb = TRUE;
  else fProb = FALSE;
  CopyData(&pBuff[iBd+1], &pBuff[iMk], pEdit->oBufLast-iBd);
  pEdit->oBufLast = pEdit->oBufLast - iBd + iMk - 1;
  if (pEdit->oBufInsert > iBd)
      pEdit->oBufInsert = pEdit->oBufInsert - iBd + iMk;
  else if ((pEdit->oBufInsert > iMk) && (pEdit->oBufInsert <= iBd))
      pEdit->oBufInsert = iMk;
  if (pEdit->oBufInsert > pEdit->oBufLast)
     pEdit->oBufInsert = pEdit->oBufLast;
```

continues

Listing 16.2. continued

```
  if (fProb)  {
    i = findPrevLine (pEdit->oBufInsert);
    pEdit->oBufLine0 = i;
  }
  nullMarkBound ();
}
}

/*************************************************************
   After screen movement (such as scrolling), this
   finds the proper location of the cursor in relationship
   to the portion of the file currently displayed.
*************************************************************/

void findCursor (void)
{
    /* locates cursor based on oBufInsert
       - might be off screen - if it is, this
       will adjust screen */

unsigned long  i, j;

  i = pEdit->iRowMin;
  while ((i <= pEdit->iRowMax) && (pEdit->oBufInsert >= pEdit->Line[i]))
       i++;
  pEdit->iLine = i - 1;
  if (pEdit->iLine < pEdit->iRowMin)
     pEdit->iLine = pEdit->iRowMin;

  j = pEdit->iLine;
  if ((pEdit->Line[j+1] < EMPTY) &&
      (pEdit->oBufInsert >= pEdit->Line[j+1]))
        pEdit->iLine = pEdit->iLine + 1;
  j = pEdit->iLine;
  pEdit->iCol = pEdit->oBufInsert - pEdit->Line[j] + pEdit->iColMin;
  if (pEdit->iLine > pEdit->iRowMax + 1)
     pEdit->iLine = pEdit->iRowMax;
}

/*************************************************************
   This readjusts iCol and iLine to get them back in sync with
   oBufInsert if oBufInsert is on screen.  If oBufInsert is
   not on screen, this makes it so it is.
*************************************************************/

void coordCursor_oBuf (void)
{
unsigned long  oBuf, i;

  i = pEdit->iRowMax+1;
  if ((pEdit->oBufInsert >= pEdit->oBufLine0) &&
      (pEdit->oBufInsert < pEdit->Line[i]))  {

     /* if bogus line, guarantee end of good */

     i = pEdit->iLine;
```

```
    if (pEdit->Line[i] == EMPTY)
       pEdit->iCol = pEdit->iColMax;

     /* if bogus line, find last good line */

    while ((pEdit->Line[i] == EMPTY) &&
           (i > pEdit->iRowMin)) {
        pEdit->iLine--;
        i = pEdit->iLine;
    }

    i = pEdit->iLine;
    pEdit->oBufInsert =
        pEdit->Line[i] + pEdit->iCol - pEdit->iColMin;
    if (pEdit->oBufInsert > pEdit->oBufLast)
       pEdit->oBufInsert = pEdit->oBufLast;
    oBuf = pEdit->Line[i+1];
    if (pEdit->oBufInsert > oBuf)
       pEdit->oBufInsert = oBuf;   /* get to potential insert - is, if
                                      prev char <> CR, is not if prev = CR */
    if (pEdit->oBufInsert == oBuf)          /* if at EOL */
       if (pEdit->pBuf[oBuf-1] == 0x0A)
         pEdit->oBufInsert--;
    pEdit->iCol = pEdit->oBufInsert + pEdit->iColMin -
                  pEdit->Line[i];
  }
}


/*********************************************************
 Adjusts oBufLine0 to make sure that oBufInsert is on the screen.
 This also sets all of the Line array values (Line[n])
*********************************************************/

void makeOnScreen (void)
{
unsigned long  i, j, k;

  /* If oBufInsert is not on screen (above current display)
     then find the previous line beginning and make that
     the new first line 1 until it is.
  */

  while (pEdit->oBufInsert < pEdit->oBufLine0)
    pEdit->oBufLine0 = findPrevLine (pEdit->oBufLine0);

  /* Set Line[iRowMin] to match oBufLine0 */

  k = pEdit->iRowMin;
  pEdit->Line[k] = pEdit->oBufLine0;

  /* Set all subsequent Line[s] by calling findEol for each. */

  for (i = k; i <= pEdit->iRowMax; i++) {
    if (pEdit->Line[i] < EMPTY) {
      j = findEol (pEdit->Line[i]);
      if (j < pEdit->oBufLast)       /* j = offset of last char of line */
```

continues

Listing 16.2. continued

```
        pEdit->Line[i+1] = j + 1;
      else
        for (j=i+1; j<NLINESMAX; j++)
              pEdit->Line[j] = EMPTY;
    }
  }

    /* If the InsertPoint (your cursor position) is past
    the last line, then do this junk to fix it */

  j = pEdit->iRowMin;
  k = pEdit->iRowMax;
  while (pEdit->oBufInsert >= pEdit->Line[k+1]) {
    for (i=j; i<=k; i++)
      pEdit->Line[i] = pEdit->Line[i+1];
    pEdit->oBufLine0 = pEdit->Line[j];

    i = findEol (pEdit->Line[k]);          /* EOL of iRowMax */
    if (i < pEdit->oBufLast)               /* i = offset of last char of line */
        pEdit->Line[k+1] = i + 1;
    else pEdit->Line[k+1] = EMPTY;
  }
}

/*********************************************************
 Redisplay all data on the screen.
*********************************************************/

void showScreen (char *pFiller)
{
unsigned long i, iLn, cb, oBuf;
char *pBuff;

    pBuff = pEdit->pBuf;

    makeOnScreen ();    /* oBufInsert on screen - Line correct */

    for (iLn = pEdit->iRowMin; iLn <= pEdit->iRowMax; iLn++) {

        /* i = offset in buf of last char on line */
        /* cb = nchars in line */

        cb = 0;
        oBuf = pEdit->Line[iLn];
        if (oBuf < EMPTY) {
            if (pEdit->Line[iLn+1] < EMPTY)
                i = pEdit->Line[iLn+1] - 1;
            else
                i = pEdit->oBufLast;
            cb = i - oBuf + 1;          /* Make size, not offset */


            if ((!pEdit->fVisible) &&
                (pBuff[i] == 0x0A) &&
                (cb))
                    cb--;
```

```
        }

        if ((cb) && (oBuf < EMPTY))
            PutVidChars (pEdit->iColMin, iLn, pBuff+oBuf, cb,
                         pEdit->iAttrNorm);

        if (cb < pEdit->sLine)
            PutVidChars (pEdit->iColMin+cb, iLn, pFiller, pEdit->sLine-cb,
                         pEdit->iAttrNorm);

        doMark (iLn);
    }
}

/*********************************************************
 Resets all variables for the editor and clears the
 buffers. Called before use and after closure.
*********************************************************/

void clearbuf (void)
{
unsigned long i;
char *pBuff;

    pBuff = pEdit->pBuf;
    FillData(filler, 80, 0x20);
    for (i=0; i<NLINESMAX; i++)
       pEdit->Line[i] = EMPTY;
    i = pEdit->iRowMin;
    pEdit->Line[i] = 0;

    pEdit->iCol = pEdit->iColMin;
    pEdit->iLine = pEdit->iRowMin;
    pEdit->bSpace = 0x20;
    pEdit->fVisible = FALSE;
    fModified = 0;
    fOvertype = FALSE;
    pEdit->oBufLast  = 0;
    pEdit->oBufInsert = 0;
    pEdit->oBufLine0 = 0;
    pBuff[pEdit->oBufLast] = 0x0F;      /* the SUN */
    nullMarkBound();
    normAttr ();
}

/*********************************************************
 This is the main editing function. It is a huge WHILE
 loop that reads keystrokes and processes them.
*********************************************************/

void Editor(char *pbExitRet)
{
unsigned long  i, j, k, key;
char  fSpecInsert;     /* TRUE = insert no matter what fOvertype is */
char  fScreen;         /* TRUE = display entire screen */
char  fDone;
unsigned char b;
```

continues

Listing 16.2. continued

```
char *pBuff, *pBuffWork;
long exch;

  pBuff = pEdit->pBuf;
  pBuffWork = pEdit->pBufWork;
  fDone = FALSE;

  if (pEdit->fVisible)
  {
    pEdit->bSpace = 0x07;
    for (i=0; i <=pEdit->oBufLast; i++)
      if (pBuff[i] == 0x20)
          pBuff[i] = pEdit->bSpace;
  } else
    pEdit->bSpace = 0x20;


  normAttr ();
  fScreen = TRUE;

  pBuff[pEdit->oBufLast] = 0x0F;        /* the SUN */

  erc = AllocExch(&exch);

  while (!fDone)
  {
    if (fScreen)
    {
      showScreen (filler);    /* we know oBufInsert on screen */
      findCursor ();
      fScreen = FALSE;
    }
    SetXY (pEdit->iCol, pEdit->iLine);
    FillData(aStat, 80, 0x20);
    i= CountEols() + pEdit->iLine;
    sprintf(aStat,"C: %02d  L: %05d  nChars: %05d",
          pEdit->iCol, i, pEdit->oBufLast);
    if (cbFilename)
        CopyData(Filename, &aStat[40], cbFilename);
    if (fOvertype)
        CopyData("OVR", &aStat[77], 3);
    else
        CopyData("INS", &aStat[77], 3);
    PutVidChars (0, 0, aStat, 80, STATVID);

    fSpecInsert = FALSE;  /* True if char should be inserted even if fOver */

    CheckErc(7, ReadKbd (&key, 1));   /* Wait for char */

    b = key & 0xff;

    if (key & 0x3000) {     /* ALT key is down */

        switch (b) {

            case 0x42:      /* ALT-B — Beginning of Text */
```

```
case 0x62:      /* ALT-b  */
       pEdit->oBufLine0 = 0;
       pEdit->oBufInsert = 0;
       pEdit->iCol = pEdit->iColMin;
       pEdit->iLine = pEdit->iRowMin;
       fScreen = TRUE;
     break;

case 0x43:      /* ALT-C — CloseFile */
case 0x63:      /* ALT-c  */
    SaveFile(TRUE, TRUE);
    fScreen = TRUE;
    break;
case 0x45:      /* ALT-E — End of Text */
case 0x65:      /* ALT-e  */
       pEdit->oBufInsert = pEdit->oBufLast;
       coordCursor_oBuf ();
       fScreen = TRUE;
     break;
case 0x01:      /* ALT-UP Arrow - Cursor to top of screen */
       pEdit->iLine = pEdit->iRowMin;
       pEdit->iCol = pEdit->iColMin;
     break;
case 0x02:  /* ALT Down Arrow - Cursor to bottom of screen */
       pEdit->iLine = pEdit->iRowMin;
       i = pEdit->iLine;
       while ((pEdit->Line[i+1] < EMPTY) &&
              (i < pEdit->iRowMax)) {
           pEdit->iLine++;
           i = pEdit->iLine;
       }
       pEdit->iCol = pEdit->iColMax;
       coordCursor_oBuf ();
     break;
case 0x03:      /* ALT-Left Arrow - Cursor to BOL */
       pEdit->iCol = pEdit->iColMin;
     break;
case 0x04:      /* ALT-Right Arrow - Cursor to EOL */
       i = pEdit->iLine;
       if (pEdit->Line[i+1] < EMPTY) {
          pEdit->iCol =
            pEdit->iColMin +
            pEdit->Line[i+1] -
            pEdit->Line[i];
            i = pEdit->iLine+1;
          if ((pBuff[pEdit->Line[i]-1] == 0x0A)
             && (pEdit->iCol > pEdit->iColMin))
           pEdit->iCol--;
       }
       else pEdit->iCol = pEdit->iColMin +
            pEdit->oBufLast - pEdit->Line[i];
     break;

case 0x56:      /* ALT-V   Make nontext chars Visible/Invisible */
case 0x76:      /* ALT-v  */
       if (pEdit->fVisible)  {
         for (i=0; i <= pEdit->oBufLast; i++)
```

continues

Listing 16.2. continued

```
                    if (pBuff[i] == 0x07)
                      pBuff[i] = 0x20;
                  pEdit->fVisible = FALSE;
                  pEdit->bSpace = 0x20;
                } else {
                  for (i=0; i<=pEdit->oBufLast; i++)
                    if (pBuff[i] == 0x20)
                      pBuff[i] = 0x07;
                  pEdit->fVisible = TRUE;
                  pEdit->bSpace = 0x07;
                }
                fScreen = TRUE;
            break;
        case 0x4F:      /* ALT-O   OpenFile */
        case 0x6F:      /* ALT-o  */
            if (!fh) {
                clearbuf();
                OpenAFile(0);
                fScreen = TRUE;
            }
            break;
        case 0x51:  /* ALT Q - Quit */
        case 0x71:
            SaveFile(TRUE, TRUE);
            fDone = TRUE;
            break;
        case 0x53:      /* ALT S - SaveFile */
        case 0x73:      /* ALT s  */
            SaveFile(FALSE, FALSE);
            fScreen = TRUE;
            break;
        case 0x7F:      /* ALT Delete (Delete Marked Block)  */
            if (pEdit->oBufMark < EMPTY)  {
               deleteData ();
               fScreen = TRUE;
             }
             break;
        default:
            break;
    }
}

else if (key & 0x0300) {        /* CTRL key is down */


}

/* Standard editing keys including LF */

else if (((b >= 0x20) && (b <= 0x7E)) || (b == 0x0D)) {
        coordCursor_oBuf ();
        if (b == 0x20)
            b = pEdit->bSpace;
        if (b == 0x0D)
            b = 0x0A;
                /* Don't overwrite CR */
```

```
        if (pBuff[pEdit->oBufInsert] == 0x0A)
          fSpecInsert = TRUE;
        if (!(putInBuf (b, fOvertype, fSpecInsert)))
            Beep();
        findCursor ();
        fScreen = TRUE;

}

else if (key & 0x0C00) {    /* SHIFT key is down & NOT letter keys */
    switch (b) {
        case 0x04:  /* SHIFT Right */
             if (pEdit->iCol < pEdit->iColMax - 5)
                   pEdit->iCol += 5;
             else
                if (pEdit->iCol < pEdit->iColMax)
                   pEdit->iCol++;
             break;
        case 0x03:  /* SHIFT Left */
             if (pEdit->iCol > pEdit->iColMin + 5)
                   pEdit->iCol -= 5;
             else
                if (pEdit->iCol > pEdit->iColMin)
                   pEdit->iCol--;
             break;
         default:
            break;
    }
}

else {          /* Unshifted editing keys */
    switch (b) {
        case 0x08:  /* Backspace */
            if (pEdit->oBufLast)  {
               coordCursor_oBuf ();
               if (pEdit->oBufInsert)  {
                 pEdit->oBufInsert = pEdit->oBufInsert - 1;
                 if (!fOvertype)  {
                   CopyData(pBuff,
                            pBuffWork,
                            pEdit->oBufLast+1);
                   CopyData(&pBuffWork[pEdit->oBufInsert+1],
                            &pBuff[pEdit->oBufInsert],
                            pEdit->oBufLast-pEdit->oBufInsert);
                   pBuff[pEdit->oBufLast] = 0;
                   pEdit->oBufLast = pEdit->oBufLast - 1;
                   if ((pEdit->oBufMark == pEdit->oBufBound) &&
                       (pEdit->oBufMark == pEdit->oBufInsert))
                      nullMarkBound ();
                   if (pEdit->oBufMark < EMPTY)  {
                     if (pEdit->oBufInsert <= pEdit->oBufMark)
                        pEdit->oBufMark--;
                     if (pEdit->oBufInsert <= pEdit->oBufBound)
                        pEdit->oBufBound--;
                   }
                 }
               }
```

continues

Listing 16.2. continued

```
            if (pEdit->oBufInsert < pEdit->oBufLine0)
               pEdit->oBufLine0 = findPrevLine (pEdit->oBufLine0);
            fScreen = TRUE;
            fModified = TRUE;
         }
         break;
      case 0x06:      /* Home - Cursor to BOL */
         pEdit->iCol = pEdit->iColMin;
         break;
      case 0x09:      /* Tab */
         if (pEdit->oBufLast + pEdit->iTabNorm < pEdit->iBufMax) {
            coordCursor_oBuf ();
            j = pEdit->iTabNorm - (pEdit->iCol % pEdit->iTabNorm);
            for (i=1; i <=j; i++)
                 putInBuf (pEdit->bSpace, FALSE, FALSE);
            fScreen = TRUE;
         }
         break;
      case 0x10:   /* F2 — UNMARK BLOCK */
            nullMarkBound ();
         break;
      case 0x11:      /* F3 — Begin Block */
         if (pEdit->oBufLast > 0)  {
            coordCursor_oBuf ();
            pEdit->oBufMark = pEdit->oBufInsert;
            i = pEdit->iLine;
            if (pEdit->oBufMark >= pEdit->Line[i+1])
               pEdit->oBufMark = pEdit->oBufMark - 1;
            if (pEdit->oBufMark == pEdit->oBufLast)
               pEdit->oBufMark = pEdit->oBufLast - 1;
            pEdit->oBufBound = pEdit->oBufMark;
            fScreen = TRUE;
         }
         break;
      case 0x12:      /* F4 — End Block */
         if (pEdit->oBufMark < EMPTY)  {
            coordCursor_oBuf ();
            pEdit->oBufBound = pEdit->oBufInsert;
            i = pEdit->iLine;
            if (pEdit->oBufBound >= pEdit->Line[i+1])
               pEdit->oBufBound--;
            if (pEdit->oBufBound == pEdit->oBufLast)
               pEdit->oBufBound = pEdit->oBufLast - 1;
            fScreen = TRUE;
         }
         break;
      case 0x17:  /* F9 - MOVE */
            coordCursor_oBuf ();
            moveData ();
            if (pEdit->oBufInsert < pEdit->oBufLine0)
               pEdit->oBufLine0 = pEdit->oBufInsert;
            fScreen = TRUE;
         break;
      case 0x18:  /* F10 - COPY */
            coordCursor_oBuf ();
            CopyIt ();
```

```
      coordCursor_oBuf ();
      fScreen = TRUE;
    break;
case 0x0C:  /* Page Down */
      coordCursor_oBuf ();
      i = pEdit->iRowMax;
      while ((pEdit->Line[i] == EMPTY) && (i > pEdit->iRowMin))
          i--;
      pEdit->oBufLine0 = pEdit->Line[i];
           /*always keep onScreen*/
      if (pEdit->oBufInsert < pEdit->oBufLine0)
         pEdit->oBufInsert = pEdit->oBufLine0;
      pEdit->iLine = pEdit->iRowMin;
      pEdit->iCol = pEdit->iColMin;
      fScreen = TRUE;
    break;
case 0x05:  /* Page Up */
    if (pEdit->oBufLine0)  {
      coordCursor_oBuf ();
      j = pEdit->iRowMax - pEdit->iRowMin;
      i = pEdit->oBufLine0;
      k = pEdit->iLine;   /*fix for scrolling when iLine=iRowMax */
      do {
        i = findPrevLine (i);
        j--;
        k--;
      }
      while ((j > 0) && (i > 0));
      pEdit->oBufLine0 = i;
          /*fix for scroll when iLine=iRowMax*/
      if (pEdit->iLine == pEdit->iRowMax)
         pEdit->oBufInsert = pEdit->Line[k];
           /*keep on screen*/
      i = pEdit->iRowMax;
      if (pEdit->oBufInsert >= pEdit->Line[i+1])
         pEdit->oBufInsert = pEdit->Line[i];
      fScreen = TRUE;
    }
    break;
case 0x01:  /* Up */
   if (pEdit->iLine > pEdit->iRowMin)  {
      pEdit->iLine--;
    } else {       /* scroll screen down if we can */
       i = pEdit->oBufLine0;
       if (i > 0)  {
           i = findPrevLine (i);
           pEdit->oBufLine0 = i;
           pEdit->oBufInsert = i;
           fScreen = TRUE;
       }
    }
    break;
case 0x02:  /* Down */
   i = pEdit->iLine;
   if ((pEdit->Line[i+1] < EMPTY) &&   /*Down Arrow*/
       (i < pEdit->iRowMax))  {
      pEdit->iLine++;
```

continues

Listing 16.2. continued

```
        }
        else {          /* ELSE scroll screen UP if we can */
            i = pEdit->iRowMax;
            if (pEdit->Line[i+1] < EMPTY)  {
                pEdit->oBufInsert = pEdit->Line[i+1];
                i = pEdit->iCol;
                j = pEdit->iLine;
                coordCursor_oBuf ();
                pEdit->iCol = i;
                pEdit->iLine = j;
                fScreen = TRUE;
            }
        }
        break;
    case 0x03:  /* Left */
        if (pEdit->iCol > pEdit->iColMin)  {          /*Left Arrow*/
           pEdit->iCol--;
         }
         break;
    case 0x04:  /* Right */
        if (pEdit->iCol < pEdit->iColMax)  {
           pEdit->iCol++;
         }
         break;
    case 0x0E:  /* Insert */
           if (fOvertype)
              fOvertype = FALSE;
            else fOvertype = TRUE;
         break;
    case 0x7F:  /* Delete */
           coordCursor_oBuf ();
           if ((pEdit->oBufLast) &&
               (pEdit->oBufLast > pEdit->oBufInsert))  {
             CopyData(pBuff,
                      pBuffWork,
                      pEdit->oBufLast+1);
             CopyData(&pBuffWork[pEdit->oBufInsert+1],
                      &pBuff[pEdit->oBufInsert],
                      pEdit->oBufLast-pEdit->oBufInsert);
             pBuff[pEdit->oBufLast] = 0;
             pEdit->oBufLast--;
             if ((pEdit->oBufInsert == pEdit->oBufMark) &&
                 (pEdit->oBufMark == pEdit->oBufBound))
                nullMarkBound ();
             if (pEdit->oBufMark < EMPTY)  {
               if (pEdit->oBufInsert < pEdit->oBufMark)
                  pEdit->oBufMark--;
               if (pEdit->oBufInsert < pEdit->oBufBound)
                  pEdit->oBufBound--;
               if (pEdit->oBufMark == pEdit->oBufLast)
                  pEdit->oBufMark--;
               if (pEdit->oBufBound == pEdit->oBufLast)
                  pEdit->oBufBound--;
             }
             fScreen = TRUE;
             fModified = TRUE;
```

```
                }
                break;
            default:
                break;
        }
    }
  } /* Not fDone */


  for (i=pEdit->iRowMin; i <=pEdit->iRowMax; i++)
    PutVidAttrs (pEdit->iColMin, i, pEdit->sLine+1, 0); /* REM buffer column */

  if (fh) {
    if (pEdit->fVisible)     /* fix visible characters */
         for (i=0; i <=pEdit->iBufMax; i++)
             if (pBuff[i] == 0x07)
                 pBuff[i] = 0x20;
    pBuff[pEdit->oBufLast] = 0;
    if (fModified) {
            erc = CheckErc(6, SetFileLFA(fh, 0));
            if (!erc)
                erc = CheckErc(5, SetFileSize(fh,pEdit->oBufLast));
            if (!erc)
                erc = CheckErc(3, WriteBytes (fh, pBuf1, pEdit->oBufLast, &i));
            fModified = 0;
    }
    CloseFile(fh);
    cbFilename = 0;
  }

  *pbExitRet = b;
  return;
}

/*************************************************************
 This is the main entry point for the editor. It allocates
 two buffers of equal size (a main and a working buffer),
 and then checks for a single parameter which should be the
 name of the file to edit.
*************************************************************/

void main(U32 argc, U8 *argv[])
{
long i;

    ClrScr();
    SetJobName("Editor", 6);
    fh = 0;

    pEdit = &EdRec;
    erc = AllocPage(32, &pBuf1);    /* 32 pages = 128K */
    erc = AllocPage(32, &pBuf2);

    pEdit->pBuf         = pBuf1;
    pEdit->pBufWork     = pBuf2;
    pEdit->iBufMax      = 131071;    /*sBuf - 1 */
    pEdit->iColMin      = 0;         /*Screen coordinates*/
```

continues

Listing 16.2. continued

```
    pEdit->iColMax      = 79;
    pEdit->iRowMin      = 1;
    pEdit->iRowMax      = 23;
    pEdit->sLine        = 80;       /* iColMax-iColMin+1 */
    pEdit->bSpace       = 0x20;
    pEdit->fVisible     = FALSE;
    pEdit->iAttrMark    = MARKVID;  /* Rev Vid*/
    pEdit->iAttrNorm    = EDVID;    /* Rev Vid Half Bright    */
    pEdit->iTabNorm     = 4;        /* Tabs every 4th column  */
    pEdit->oBufLine0    = 0;        /* oBufLine0 */
    pEdit->iCol         = 0;        /* cursor, 0..sLine-1  */
    pEdit->iLine        = 0;        /* cursor, 0..cLines-1    */
    pEdit->oBufInsert   = 0;        /* offset of next char in */
    pEdit->oBufLast     = 0;        /* offset+1 of last char  */
    pEdit->oBufMark     = EMPTY;
    pEdit->oBufBound    = EMPTY;

    SetNormVid(NORMVID);

    FillData(filler, 80, 0x20);
    for (i=0; i<NLINESMAX; i++)
       pEdit->Line[i] = EMPTY;
    i = pEdit->iRowMin;
    pEdit->Line[i] = 0;

    fModified = 0;
    fOvertype = FALSE;    /* Set Overtype OFF */

    if (argc > 1)
    {
        OpenAFile(argv[1]);
    }

    Editor(&b);
    ExitJob(0);
}
```

DumbTerm

DumbTerm is possibly the "dumbest" communications terminal program in existence. It is included here to demonstrate the interface to the RS-232 communications device driver.

Listing 16.3. DumbTerm source code.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "\OSSOURCE\MDevDrv.h"
#include "\OSSOURCE\MJob.h"
```

```
#include "\OSSOURCE\MKbd.h"
#include "\OSSOURCE\MTimer.h"
#include "\OSSOURCE\MVid.h"
#include "\OSSOURCE\RS232.h"

#define NORMVID BRITEWHITE¦BGBLUE
#define CLIVID WHITE¦BGBLACK

unsigned long key;

struct statRecC com;

/************************************************/

void    main(void)

{

int           erc, i;
unsigned char b, lastb;
char          fOK;


SetNormVid(NORMVID);
ClrScr();

printf("     Terminally DUMB, Dumb Terminal Program\r\n");
printf("      (MMURTL Comms Device Driver demo) \r\n");

/* Get the 64-byte device status block that is specific to the
RS-232 device driver. The structure is defined in commdrv.h
*/

 erc = DeviceStat(6, &com, 64, &i);

 if (erc)
 {
    SetNormVid(CLIVID);
    ClrScr();
    printf("Error on Device Stat: %d\r\n", erc);
    ExitJob(erc);
 }

  /* set the params in the block */

  com.Baudrate = 9600;
  com.parity = NO_PAR;
  com.databits = 8;
  com.stopbits = 1;

/* View other params, which we could set but should already be
   defaulted with standard values when driver was initialized.
*/

  printf("IRQNum: %d\r\n", com.IRQNum);
  printf("IOBase: %d\r\n", com.IOBase);
  printf("sXBuf:  %d\r\n", com.XBufSize);
```

continues

Listing 16.3. continued

```
  printf("sRBuf:  %d\r\n", com.RBufSize);
  printf("RTimeO: %d\r\n", com.RTimeOut);
  printf("XTimeO: %d\r\n", com.XTimeOut);

/* Set the params we changed with a DeviceInit */

 erc = DeviceInit(6, &com, 64);
 if (erc)
 {
    SetNormVid(CLIVID);
    ClrScr();
    printf("Error on Device Init: %d\r\n", erc);
    ExitJob(erc);
 }

   /* If device init went OK, we open the comms port */


              /* device, dOpNum,   dLBA, dnBlocks, pData */
    erc = DeviceOp(6,      CmdOpenC, 0,    0,        &i);

    if (erc)
    {
        SetNormVid(CLIVID);
        ClrScr();
        printf("OpenCommC ERROR: %d \r\n", erc);
        ExitJob(erc);
    }

    printf("Communications Port Initialized.\r\n");

    fOK = 1;

    /* This is it... */

    while (fOK)
    {

        if (!ReadKbd(&key, 0))      /* no wait */
        {
            b = key & 0x7f;

            if (key & 0x3000)
            {       /* ALT key is down */
                switch (toupper(b))
                {
                    case 'Q' :
                     /* device, dOpNum,   dLBA, dnBlocks, pData */
                        erc = DeviceOp(6,    CmdCloseC, 0,  0,  &i);
                        SetNormVid(CLIVID);
                        ClrScr();
                        ExitJob(erc);
                        break;
                    default: break;
                }
            }
            else
```

```
            {
                /* device, dOpNum,   dLBA, dnBlocks, pData */
                erc = DeviceOp(6,    CmdWriteB, 0,  0,  &b);
                if (erc)
                    printf("WriteByteCError: %d \r\n", erc);
                else
                {
                    if (b == 0x0D)
                    {
                        b = 0x0A;
                        erc = DeviceOp(6,    CmdWriteB, 0,  0,  &b);
                    }
                }
            }
        }
           /* device, dOpNum,   dLBA, dnBlocks, pData */
        erc = DeviceOp(6,    CmdReadB, 0,  0,  &b);
        if (!erc)
        {
                TTYOut (&b, 1, NORMVID);
                /* add a LF if it's not there after a CR... */
                if ((lastb == 0x0D) && (b != 0x0A))
                    TTYOut ("\n", 1, NORMVID);
                lastb = b;
        }
    }

}
```

Print

The Print program is an example of using the device-driver interface for the parallel LPT device driver. The Print program formats a text file and sends it to the LPT device. It recognizes the following command-line switches:

`\n` Expand tabs to *n* space columns. (n = 1, 2, 4, or 8)

`\D` Display the file while printing.

`\F` Suppress the form feed sent by the Print command.

`\B` Binary print. Send with no translations.

Print also converts single-line feeds to CR/LF.

Print Source Listing

In Listing 16.4, notice that the device-driver interface is almost identical for each device. (Compare DumbTerm and this program.) That's the intention of the design. The differences in handling different devices will be the values in the status record and also the commands that are specific to a driver.

Listing 16.4. Print source code.

```
/* A simple program that prints a single file directly using
   the Parallel Device Driver in MMURTL (Device No. 3 "LPT") .
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "\OSSource\MDevDrv.h"
#include "\OSSource\MJob.h"
#include "\OSSource\MKbd.h"
#include "\OSSource\MTimer.h"
#include "\OSSource\MVid.h"
#include "\OSSource\Parallel.h"

#define     FF              0x0C
#define     LF              0x0A
#define     CR              0x0D
#define     TAB             0x09

unsigned long key;

long tabstops = 4;
long NoFF = 0;
long fDisplay = 0;
long fBinary = 0;
long col = 0;

char name[80];
FILE *f;

struct statRecL lpt;

/*****************************************************/

void main(long argc, unsigned char *argv[])
{
long      erc, erck, i, cl;
unsigned char b, lastb;
char          fdone, *ptr;

    SetJobName("Printing", 8);

    name[0] = 0;

    for(i=1; i < argc; ++i)     /* start at arg 1 */
    {
        ptr = argv[i];
        if (*ptr == '/')
        {
            ptr++;
            switch(*ptr)
            {
                case '1' :          /* Tab Translation Width */
```

```
            case '2' :
            case '4' :
            case '8' :
                tabstops = *ptr - 0x30;
                break;
            case 'F' :          /* No FF at end of file */
            case 'f' :
                NoFF = 1;
                break;
            case 'D' :          /* Display while printing */
            case 'd' :
                fDisplay = 1;
                break;
            case 'B' :          /* BINARY - No translation at all! */
            case 'b' :
                fBinary = 1;
                break;
            default:
                printf("Invalid switch");
                exit(1);
                break;
        }
    }
    else if(!name[0])
        strncpy (name, argv[i], 79);
}

if (!name[0])
{
    /* Input file not explicitly named errors out */

    printf("Print File, Version 1,0\r\n");
    printf("Usage: Filename /1 /2 /4 /8 /F /D /B\r\n");
    printf("/1 /2 /4 /8 - Tab stop translation value\r\n");
    printf("/F  no FormFeed at end of file\r\n");
    printf("/D  Display file while printing\r\n");
    printf("/B  Binary print. NO translation, no FF\r\n\n");
    printf("Error: Source filename required\r\n");
    exit(1);
}

/* Get the 64-byte device status block which is specific to the
   parallel device driver. The structure is defined in parallel.h
   We do this just to see if it's a valid device.
*/

erc = DeviceStat(3, &lpt, 64, &i);

if (erc)
{
    printf("Error getting LPT Device Status: %d\r\n", erc);
    ExitJob(erc);
}

/* If device status went OK, we open the printer port */

            /* device, dOpNum,   dLBA, dnBlocks, pData */
```

continues

Listing 16.4. continued

```
    erc = DeviceOp(3,       CmdOpenL, 0,    0,        &i);

    if (erc)
    {
        printf("OpenLPT ERROR: %d \r\n", erc);
        ExitJob(erc);
    }

    printf("Printing %s ...\r\n", name);

    /* This is it... */

    f = fopen(name, "r");

    if (!f)
    {
              /* device, dOpNum,   dLBA, dnBlocks, pData */
        erc = DeviceOp(3,  CmdCloseLU, 0,    0,        &i);
        printf("Can't open: %s\r\n", name);
        ExitJob(erc);
     }

    col = 0;
    i = 0;
    b = 0;
    fdone = 0;

    while ((!fdone) && (!erc))
    {
        i++;
        cl = fgetc(f);
        lastb = b;
        b = (cl & 0xff);

        if (cl == EOF)
        {
            fdone = 1;
        }
        else if (fBinary)
        {
            erc = DeviceOp(3, CmdWriteB, 0,  1, &lastb);
        }
        else
        {
            switch (b)
            {           /* print/translate the char */
                case CR:
                    erc = DeviceOp(3, CmdWriteB, 0,  1, &b);
                    break;
                case LF:
                    if (lastb != CR)
                    {
                        lastb = CR;
                        erc = DeviceOp(3, CmdWriteB, 0,  1, &lastb);
```

```
                }
                erc = DeviceOp(3, CmdWriteB, 0,  1, &b);
                if (fDisplay)
                    printf("\r\n", lastb);
                col = 0;  /* reset */
                break;
            case TAB:
                do
                {
                    erc = DeviceOp(3, CmdWriteB, 0,  1, " ");
                    col++;
                    if (fDisplay)
                        printf(" ");
                } while (col % tabstops);
                break;
            default:
                if (fDisplay)
                    printf("%c", b);
                col++;
                erc = DeviceOp(3, CmdWriteB, 0,  1, &b);
                    if (erc)
                        printf("Error Writing Byte: %d\r\n", erc);
                break;
        }
    }

    if (i%100==0)       /* every 100 chars see if they want to abort */
    {
        erck = ReadKbd(&key, 0);
        /* no wait */
        if (!erck)
        {
            if (key & 0xff == 0x1b)
            {
                fdone = 1;
                erc = 4;
            }
        }
    }
}

if ((!fBinary) && (!NoFF))
{
    erc = DeviceOp(3, CmdWriteB, 0,  1, "\f");
}

fclose(f);
          /* device, dOpNum, dLBA, dnBlocks, pData */
erc = DeviceOp(3,  CmdCloseL,   0,   0,        &i);
if (erc)
    printf("Can't close LPT. Error: %d\r\n", erc);
printf("Done\r\n");
ExitJob(erc);

}
```

System Service Example

Installable system services are the easiest way to coordinate resource usage and provide additional functionality in MMURTL.

The following two listings show a simple system service and a client of that service. The program is similar to the example shown in Chapter 10, "Systems Programming." It has been expanded some, and it also allows deinstallation with proper termination and recovery of operating-system resources.

Service.C Listing

Execute this from a command-line interpreter. It becomes a system service while keeping its virtual video screen. Pressing any key will terminate the program properly. While this program is running, run the TestSvc.RUN program to exercise the service.

Listing 16.5. Simple service source code.

```
/* Super Simple System Service.
   This is expanded from the sample in the System Programmer
   chapter to show how to properly deinstall a system service.
   The steps to deinstall are:
   1) UnRegister the Service
   2) Serve all remaining requests at service exchange
   3) Deallocate all resources
   4) Exit
*/

#include <stdio.h>
#include "\OSSource\MKernel.h"
#include "\OSSource\MJob.h"
#include "\OSSource\MVid.h"

#define ErcOK         0
#define ErcOpCancel   4
#define ErcNoSuchSvc  30
#define ErcBadSvcCode 32

struct RqBlkType *pRqBlk;        /* A pointer to a Reqeust Block */
unsigned long NextNumber = 0;    /* The number to return */
unsigned long MainExch;          /* Where we wait for Requests */
unsigned long Message[2];        /* The Message with the Request */
long rqHndl;                  /* Used for keyboard request */

void main(void)
{
unsigned long OSError, ErrorToUser, keycode;
long *pDataRet;

  OSError = AllocExch(&MainExch);     /* get an exchange */

  if (OSError)                        /* look for a kernel error */
    ExitJob(OSError);
```

```
  OSError = RegisterSvc("NUMBERS ", MainExch);

  if (OSError)                        /* look for a system error */
    ExitJob(OSError);

  SetNormVid(WHITE|BGBLACK);
  ClrScr();

  printf("NUMBERS Service Installed.\r\n");
  printf("ANY valid keystroke will terminate the service.\r\n");

  OSError = Request("KEYBOARD", 1, MainExch, &rqHndl, 0, &keycode,
                    4, 0, 0, 1, 0, 0);  /* 1 in dData0 = WAIT for key */
  if (OSError)
      printf("Error on Keyboard Request:\r\n", OSError);

  while (1)         /* WHILE forever (almost...) */
  {

    /* Now we wait for a client or for a keystroke to come back */

    OSError = WaitMsg(MainExch, Message); /* Exch & pointer */

    if (!OSError)
    {

      if (Message[0] == rqHndl)  /* it was a keystroke and NOT a client */
      {
        UnRegisterSvc("NUMBERS ");
        while (!CheckMsg(MainExch, Message))
        {
            pRqBlk = Message[0];
            Respond(pRqBlk, ErcNoSuchSvc);
        }
        DeAllocExch(MainExch);
        ExitJob(ErcOpCancel);
      }

      pRqBlk = Message[0];   /* First DWORD contains ptr to RqBlk */

      if (pRqBlk->ServiceCode == 0)          /* Abort request from OS */
          ErrorToUser = ErcOK;

      else if (pRqBlk->ServiceCode == 1)     /* User Asking for Number */
      {
         pDataRet = pRqBlk->pData1;
         *pDataRet = NextNumber++;       /* Give them a number */
         ErrorToUser = ErcOK;                 /* Respond with No error */

        printf("NUMBERS Service gave out number: %d.\r\n", NextNumber-1);

      }
      else ErrorToUser = ErcBadSvcCode;       /* Unknown Service code! */

      OSError = Respond(pRqBlk, ErrorToUser); /* Respond to Request */

    }
  }  /* Loop while(1) */
}
```

TestSvc.C Listing

The program in Listing 16.6 exercises the sample system service listed previously. Execute this from a CLI while the service is running. (Also try it when the service is not running. You should receive a proper error code indicating that the service is not available.)

Listing 16.6. TestSvc.C.

```
/* Test client for the NUMBERS System Service.
   Run this program from another CLI when Service is running
*/
#include <stdio.h>
#include "\OSSOurce\MKernel.h"

unsigned long Number;           /* The number to return */
unsigned long Exch;             /* Where the service will respond */
unsigned long Message[2];       /* The Message from the service */

void main(void)
{
unsigned long Error, rqhndl;

    Error = AllocExch(&Exch);     /* get an exchange */

    if (Error)
        printf("Error %d allocating Exchange.\r\n", Error);

    Error = Request("NUMBERS ", 1, Exch, &rqhndl,
                    0,                          /* No Send ptrs */
                    &Number, 4, 0, 0,
                    0,0,0);
    if (Error)
        printf("Error %d from Request.\r\n", Error);

    if (!Error)
    {
        Error = WaitMsg(Exch, Message);

        if (Error)
            printf("Error %d from WaitMsg.\r\n", Error);
        else
        {
            if (Message[1])
                printf("Error %d from NUMBERS Service.\r\n", Message[1]);
            else
                printf("NUMBERS Service gave out number: %d.\r\n", Number);
        }
    }

    DeAllocExch(Exch);  /* it's nice to do this for the OS */
}
```

IV

The Operating System Source Code

17

Introduction to the Source Code



This chapter discusses each of the important source files, some of the things that tie all of the source files together, and how to build the operating system using the tools included on the CD-ROM.

One of my goals was to keep the source code organized in an easy-to-understand fashion. I hope I've met this goal. You may find some of the source files rather large, but most of the dependent functions will be contained in the file you are viewing or reading. You won't have to go searching through hundreds of files to find a three-line function.

Comments in the Code

You'll find out that I make a lot of comments in my source code. By including many comments, I can, a year or two later, go back and figure out what I did. It's not that I couldn't actually figure out the code without the comments; I just expend the time up front instead of later. In fact, sometimes I write the comments describing a small block of code before I actually code it. Doing so helps me think.

Calling Conventions

I bring up the topic of calling conventions here because, as you look through the code, you will see that much of the assembler looks almost as if it were generated by a compiler. When a human writes this much assembler (and the code is to be interfaced with assembler generated by a compiler), the human becomes a compiler of sorts.

Many of the lower-level procedures use registers to pass data back and forth. Quite a few of them, however, use the stack just as a compiler would. Two conventions are used throughout the operating system. The most prominent and the preferred method is pushing parameters left to right, with the called procedure cleaning the stack. The other is a mangled version of what you'd find in most C compilers for functions with an ellipsis (. . .). I call it *mangled* because the arguments are still pushed left to right, but the number of variable arguments is passed in the `EDI` register. The caller cleans the stack in this convention.

Organization

The MMURTL source code is organized in a modular fashion. Each logical section is contained in one file (or more) that is either C or assembler source code. There are also header files for the C source, and there are INCLUDE files for the assembler that tie publics, common variables, and structures together.

All source files are located in one directory. There are 29 main source files (excluding header and include files). A brief description of each of the main source files follows:

MOSIDT.ASM This defines the basic Interrupt Descriptor Table (IDT). The IDT is a required table for the Intel processors. This is really just a data position holder in the data segment, because the entries in the IDT are done dynamically. The position of this file in relationship to the others is critical because the offset to the IDT must be identified to the processor. The IDT is the first file in the source, which means its data will be at the lowest address. In fact, it ends up at physical and linear address 0.

MOSGDT.ASM This defines the Global Descriptor Table (GDT). This table is also a required table for the Intel processors when they run in protected mode. The GDT is 6K in size. Several selector entries are predefined so you can go directly into protected mode. This ends up at address 800h physical and linear. The position of this file is also critical, because the file is identified to and used by the processor. This must be the second source file in the assembler template file.

MOSPDR.ASM This is also a data definition file that sets up the operating system's page directory and first page table. This file should also remain in its current order because I have hard-coded its address in some source modules.

MPublics.ASM This is also a table, but it is not used by the processor. This defines the selector entries for all MMURTL public calls.

MAIN.ASM This is the first assembler file that contains code. The operating system entry point is located here (the first OS instruction executed). Also some data is defined here. This file should be left in the order I have it in. The ordering for the remainder of the modules is not as important. On the other hand, I haven't tried playing "musical source files" either. Leaving them in their current order would probably be a good idea.

Keyboard.ASM The keyboard source goes against my own rule of not combining a device driver with a system service. Keyboard.ASM was one of the first files written, and it works rather well, so I am reluctant to begin rewriting it (although I will, undoubtedly, someday). System services in assembly language are not fun to write or maintain. Chapter 25, "Keyboard Code," explains all the mysteries behind this table-driven piece of work.

Video.ASM Video.ASM code and data was also one of the first pieces I wrote. You have to see what you're doing, even in the earliest stages of a project. Chapter 26, "Video Code," goes into detail on this file's contents.

Debugger.ASM This file contains the bulk of the debugger logic. Debugging is never fun when you have to do it at the assembler level. Debugging a debugger is even less fun.

UASM.C This file is the debugger disassembler. It is a table-driven disassembler used to display instructions in the debugger. It's not very pretty, but it's effective and accurate.

DevDrvr.ASM The device-driver interface code and data are contained in this file. This provides the re-entrancy protection and also the proper vectoring (indirect calling) of device-driver functions.

Floppy.C Floppy disk device drivers are much harder to write than IDE hard disk drivers. This is one complicated driver. It seems to me that hardware manufacturers could be a little more compassionate toward software people. The reasoning behind controlling the drive motors separately from the rest of the floppy electronics still baffles me.

HardIDE.C The IDE hard disk device driver was fairly easy to write. I was surprised. You will still see remnants of MFM drive commands in here because I started with that. I think it still works with MFM drives, but I don't have any more to test it. If you have MFM drives, you're on your own.

RS232.C This file contains the RS-232 UART driver. It drives two channels, but it can be modified for four, if necessary. It supports the 16550 but only to enable the input buffer (which helps).

Parallel.C The simple parallel port driver is contained here.

FSys.C This is an MS-DOS FAT-compatible file system. It has only the necessary calls to get the job done. It provides stream access as well as block access to files.

JobC.C The program loader and most of the job-related functions are handled in this file.

JobCode.ASM Lower-level job functions and supporting code for functions in JobC.C are contained in this file.

TmrCode.ASM The interrupt service routine for the primary system timer—along with all of the timer functions, such as `Sleep()`—are in this file.

IntCode.ASM Interrupt Service Routine (ISR) handling code can be found here.

RQBCode.ASM Request block management code is in this file.

DMACode.ASM In many systems, when device driver writers need DMA, they must handle it themselves. This file has routines that handle DMA for the programmer writing device drivers. They need only to know the channel, how much to move, and the mode. Device driver writers will appreciate this.

NumCnvrt.ASM This is some code used internally to convert numbers to text and vice-versa.

MemCode.ASM The code memory allocation, address aliasing, and management of physical memory can be found in this file.

SVCCode.ASM System service management code such as `RegisterSvc()` is in this file.

MiscCode.ASM This file has all the little things you need, but you never know where to put. High-speed string manipulation and comparison, as well as port I/O support, is in this file.

Kernel.ASM All of the kernel calls and most of the kernel support code is in this file. Chapter 18, "Kernel Code," discusses the kernel and presents the most important pieces of its code along with discussions of the code.

Except.ASM Exception handlers such as those required for processor faults are in this file. Most of them cause you to go directly into the Debugger.

InitCode.ASM This file has all the helper routines that initialize most of the dynamic operating-system functions.

Monitor.C The monitor program is contained in this file. It provides a lot of the OS functionality, but it really isn't part of the operating system. You could replace quite easily.

Building MMURTL



MMURTL requires the CM32 compiler (included on the CD-ROM) for the C source files and the DASM assembler (also included on the CD-ROM). One MS-DOS batch file (MakeALL.BAT) will build the entire operating system. This takes about 1.5 minutes on a 486/33.

Each of the C source files is turned into an assembler file (which is what the CM32 compiler produces). All of the assembly language source files are then assembled with DASM.

DASM produces the operating system RUN file. The RUN file is in the standard MMURTL Run file format, which is described completely in Chapter 28, "DASM: A 32-Bit Intel-Based Assembler."

The only two MS-DOS commands you need are `CM32` and `DASM`. An example of compiling one of the source files is the following:

```
C:\MMURTL> CM32 Monitor.C
```

After all the C source files are compiled, you use the `DASM` command, as follows:

```
C:\MMURTL> DASM MMURTL.ATF
```

It's truly that simple. If you want an error file, specify `/E` on the command line after the name. If you want a complete listing of all processor instructions and addresses, specify `/L` on the command line after the name.

The following is the Assembler Template File (ATF). This file is required to assemble complex programs with DASM. The assembler is discussed in detail in Chapter 28, but you don't need to read all the DASM documentation to build MMURTL.

```
;Assembler Template File for MMURTL V1.0
.DATA
;These first 5 INCLUDES MUST be in this order!!!!
.INCLUDE MOSIDT.ASM     ; Defines the Interrupt Descriptor Table
.INCLUDE MOSGDT.ASM     ; 6K for GDT @ 00000800h Physical (1.5 Pages, 6K)
```

```
.INCLUDE MOSPDR.ASM     ; OS Page Directory & 1st Table (8k)
.INCLUDE MPublics.ASM   ; Indirect calling address table
.INCLUDE MAIN.ASM       ; Main OS code and some data & START address

; Ordering for the remainder of the modules is not as important.
; An asterisk (*) by the include file indicates it originates as
; a C source file and must be compiled first with
; CM32 (no switches needed).
;
.INCLUDE Keyboard.ASM   ;  Keyboard ISR and Service
.INCLUDE Video.ASM      ;  Video code and data
.INCLUDE Debugger.ASM   ;  Code & Data for Debugger
.INCLUDE UASM.ASM       ;* Code & Data for Debugger Disassembler
.INCLUDE DevDrvr.ASM    ;  Code and Data for Device Driver Interface
.INCLUDE Floppy.ASM     ;* Floppy Device Driver
.INCLUDE HardIDE.ASM    ;* IDE Disk Device Driver
.INCLUDE RS232.ASM      ;* Serial Comms Device Driver
.INCLUDE Parallel.ASM    ; Parallel Comms Device Driver
.INCLUDE FSys.ASM       ;* File System Service
.INCLUDE JobCode.ASM    ;  Additional Job management
.INCLUDE JOBC.ASM       ;* Loader/Job handling Code & Data
.INCLUDE TmrCode.ASM    ;  Timer Code
.INCLUDE IntCode.ASM    ;  ISR handling code
.INCLUDE RQBCode.ASM    ;  Request Block Code
.INCLUDE DMACode.ASM    ;  DMA Handling Code
.INCLUDE NumCnvrt.ASM   ;  Maybe Move to DLL later or can it
.INCLUDE MemCode.ASM    ;  Memory management code
.INCLUDE SVCCode.ASM    ;  System Service management code
.INCLUDE MiscCode.ASM   ;  Misc. code (string,array,I/O support)
.INCLUDE Kernel.ASM     ;  Kernel code
.INCLUDE Except.ASM     ;  Exception handlers
.INCLUDE InitCode.ASM   ;  Initialization support code
.INCLUDE Monitor.ASM    ;* The monitor code & data

.END
```

If you've read all of the chapters preceding this one, you'll understand that all MMURTL programs are comprised of only two segments—code and data. The order of the assembler files in the ATF file determines the order of the data and code. As each piece of code is assembled, it is placed into the code segment in that order. The same is true for the data as it is encountered in the files.

Using Pieces of MMURTL in Your OS

You'll find that most of the source code is modular. For instance, you can take the file system code (FSYS.C) and turned it into a free-standing program with little effort. I built each section of code in modular pieces and tried to keep it that way.

Even though I use CM32 to compile the C source files, the code can be compiled with any ANSI C compiler. You will, no doubt, use another compiler for your project. You'll find that many items in MMURTL depend on the memory alignment of some of the structures. This is one thing you should watch. CM32 packs all structure fields completely. In fact, CM32 packs all variables with no additional padding.

Using Pieces of MMURTL in Other Projects

Some of you may find some of the code useful in projects that have absolutely nothing to do with writing an operating system. The same basic warning about the alignment of structures applies.

The assembler INCLUDE files that define all the structures actually define offsets from a memory location. This is because DASM doesn't support structures, but you would have no problem setting them up for TASM or MASM.

The calling conventions (described earlier) may get you into trouble if you want to use some of the assembler routines from a high-level language. If your C compiler allows it, you can declare functions as Pascal functions (some even use the old PLM type modifier), and they will work fine. Otherwise, you can calculate the correct offsets and fix the `EQU` statements for access where a C compiler would put them on the stack. You would also need to remove the numbers following the `RET` and `RETF` statements so that the caller could clean the stack, as is the C convention.

18

Kernel Code

The kernel code is very small. The actual message handling functions, the scheduler, and the associated kernel helper procedures turn out to be less than 3K of executable code.

The code in the kernel is divided into four main sections: data, local helper functions, internal public functions, and kernel publics functions. Being an old "structured" programmer, I have made most of the "called" functions reside above those that call them. If you read through the descriptions of these little "helper" routines before you look at the code for the kernel primitives, I think the code will make more sense to you.

Naming Conventions

For helper routines that are called only from other assembly language routines, the name of the routine is unmodified (no underscores used).

For functions that must be reached by high-level languages (such as C), a single underscore is prepended to the name. For public functions that are reached through call gates from applications, two underscores are prepended to the name.

Kernel Data

The source file for the kernel doesn't really contain any data variable declarations. The bulk of the data variable declarations are in the file Main.ASM. The data that the kernel primitives deal with are usually used in context to the task that is running. These data are things such as the task state segments, job control blocks, and memory management tables. These all change with respect to the task that is running.

The bulk of the other data items are things for statistics: the number of task switches (`_nSwitches`); the number of times the CPU found itself with nothing to do (`_nHalts`); and the number of tasks ready to run (`_nReady`).

Variables named with a leading underscore are accessible from code generated with the C compiler. For the most part, these are statistic-gathering variables, so I could gain access to them with the monitor program.

The INCLUDE files define things such as error codes and also offsets into structures and tables. See Listing 18.1.

Listing 18.1. Kernel data segment source.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE TSS.INC
.INCLUDE RQB.INC
.INCLUDE JOB.INC

dJunk DD 0      ;Used as temp in Service Abort function
```

```
EXTRN TimerTick  DD
EXTRN SwitchTick DD;
EXTRN dfHalted   DD
EXTRN _nSwitches DD
EXTRN _nHalts    DD
EXTRN _nReady    DD
```

Local Kernel Helper Functions

This section has all of the little functions that help the kernel primitives and scheduler code. Most of these functions manipulate all the linked lists that maintain things such as the ready queue, link blocks, and exchanges.

The interfaces to these calls are exclusively through registers. The issue here is strictly speed. It's not that the routines are called so often; it's that they manipulate common data and therefore are *not* re-entrant, which means interrupts must be cleared. I also had to keep a very good track of which registers I used.

Because certain kernel functions may be called from ISRs, and because portions of other kernel functions may be interrupted by a task change that happens as a result of an action that an ISR takes, you must ensure that interrupts are *disabled* prior to the allocation or deallocation of *all* kernel data segment resources. This especially applies when a message is "in transit" (taken, for example, from an exchange but not yet linked to a TSS and placed on the ready queue). This is important. The concept itself should be very important to you if you intend to write your own multitasking operating system.

The functions described have a certain symmetry that you'll notice, such as `enQueueMsg` and `deQueueMsg` along with `enQueueRdy` and `deQueueRdy`. You may notice that there is a `deQueueTSS` but no `enQueueTSS`. This is because the `enQueueTSS` functionality is needed only in two places and, therefore, is coded in-line.

The following lines begin the code segment for the kernel.

```
.CODE

EXTRN LinToPhy NEAR
```

enQueueMsg

The `enQueueMsg` function places a link block containing a message on an exchange. The message can be a pointer to a Request or an 8-byte generic message. Interrupts will already be disabled when this routine is called.

In MMURTL, an exchange is a place where either messages or tasks wait (link blocks that contain the message actually wait there). There can never be tasks *and* messages at an exchange at the same time (unless the kernel is broken!). When a message is sent to an exchange, and if a task

is waiting there, the task is immediately associated with the message and placed on the ready queue in priority order. For this reason, you share the HEAD and TAIL link pointers of an exchange for tasks and messages. A flag tells you whether it's a task on a message. See Listing 18.2.

Listing 18.2. Queueing for messages at an exchange.

```
enQueueMsg:
;
; INPUT : ESI,EAX
; OUTPUT : NONE
; REGISTERS : EAX,EDX,ESI,FLAGS
; MODIFIES : EDX
;
; This routine will place the link block pointed to by EAX onto the exchange
; pointed to by the ESI register. If EAX is NIL then the routine returns.
;
        OR  EAX,EAX             ; if pLBin = NIL THEN Return;
        JZ eqMsgDone            ;
        MOV DWORD PTR [EAX+NextLB], 0   ; pLBin^.Next <= NIL;
        XCHG ESI,EAX            ; pExch => EAX, pLBin => ESI
        CMP DWORD PTR [EAX+EHead], 0    ; if ..MsgHead = NIL
        JNE eqMNotNIL           ; then
        MOV [EAX+EHead],ESI     ;  ..MsgHead <= pLBin;
        MOV [EAX+ETail],ESI     ;  ..MsgTail <= pLBin;
        MOV DWORD PTR [EAX+fEMsg], 1    ; Flag it as a Msg (vice a task)
        XCHG EAX,ESI                    ; Put pExch Back in ESI
        RETN                    ; else
eqMNotNIL:
        MOV EDX,[EAX+ETail]     ;  ..MsgTail^.NextLB <= pLBin;
        MOV [EDX+NextLB],ESI    ;
        MOV [EAX+ETail],ESI     ;  ..MsgTail <= pLBin;
        MOV DWORD PTR [EAX+fEMsg], 1        ; Flag it as a Msg (vice a task)
        XCHG EAX,ESI            ; Put pExch Back in ESI
eqMsgDone:
        RETN                    ;
```

deQueueMsg

The `deQueueMsg` function removes a link block from an exchange if one exists there. If not, `deQueueMsg` returns NIL (0) in the EAX register.

As stated before, exchanges can hold messages or tasks, but not both. You check the flag in the exchange structure to see which is there. If what's there is a message, you remove it from the linked list and return it. If not, you return NIL (0). See Listing 18.3.

Listing 18.3. De-queueing a message from an exchange.

```
deQueueMsg:
;
```

```
; INPUT : ESI
; OUTPUT : EAX
; REGISTERS : EAX,EBX,ESI,FLAGS
; MODIFIES : *prgExch[ESI].msg.head and EBX
;
; This routine will dequeue a link block on the exchange pointed to by the
; ESI register and place the pointer to the link block dequeued into EAX.
;
        MOV EAX,[ESI+fEMsg]     ; Get Msg Flag
        OR EAX, EAX             ; Is it a Msg?
        JZ deMsgDone            ; No! (return 0)
        MOV EAX,[ESI+EHead]     ; pLBout <= ..MsgHead;
        OR EAX, EAX             ; if pLBout = NIL then Return;
        JZ deMsgDone            ;
        MOV EBX,[EAX+NextLB]    ; ..MsgHead <= ..MsgHead^.Next;
        MOV [ESI+EHead],EBX     ;

deMsgDone:
        RETN                    ;
```

deQueueTSS

The deQueueTSS function removes a pointer to a TSS from an exchange if one exists there. If not, it returns NIL (0) in the EAX register. See Listing 18.4.

Listing 18.4. De-queueing a task from an exchange.

```
deQueueTSS:
;
; INPUT : ESI
; OUTPUT : EAX
; REGISTERS : EAX,EBX,ESI,FLAGS
; MODIFIES : EAX,EBX
;
; This routine will dequeue a TSS on the exchange pointed to by the ESI
; register and place the pointer to the TSS dequeued into EAX.
; EAX return NIL if no TSS is waiting at Exch ESI
;
        XOR EAX,EAX             ; Set up to return nothing
        MOV EBX,[ESI+fEMsg]     ; Msg flag (is it a Msg)
        OR EBX, EBX
        JNZ deTSSDone           ; It's a Msg (return leaving EAX 0)
        MOV EAX,[ESI+EHead]     ; pTSSout <= ..TSSHead;
        OR EAX, EAX             ; if pTSSout = NIL then Return;
        JZ deTSSDone            ;
        MOV EBX,[EAX+NextTSS]   ; ..TSSHead <= ..TSSHead^.Next;
        MOV [ESI+EHead],EBX     ;

deTSSDone:
        RETN                    ;
```

enQueueRdy

The enQueueRdy function places a task (actually a TSS) on the ready queue. The ready queue is a structure of 32 linked lists, one for each of the possible task priorities in MMURTL. You dereference the TSS priority and place the TSS on the proper linked list. See Listing 18.5.

Listing 18.5. Adding a task to the ready queue.

```
PUBLIC enQueueRdy:
;
; INPUT : EAX
; OUTPUT : NONE
; REGISTERS : EAX,EBX,EDX,FLAGS
; MODIFIES : EAX,EBX,EDX
;
; This routine will place a TSS pointed to by EAX onto the ReadyQueue. This
; algorithm chooses the proper priority queue based on the TSS priority.
; The Rdy Queue is an array of QUEUES (2 pointers, head & tail per QUEUE).
; This links the TSS to rgQueue[nPRI].
;
        OR  EAX,EAX             ; if pTSS = NIL then return;
        JZ eqRdyDone            ;
        INC _nReady             ;
        MOV DWORD PTR [EAX+NextTSS], 0  ; pTSSin^.Next <= NIL;
        XOR EBX,EBX             ; get the priority
        MOV BL,[EAX+Priority]   ; in EBX
        XCHG EAX,EBX            ; Priority => EAX, pTSSin => EBX
        SHL EAX, 3              ; Times 8 (size of QUEUE)
        LEA EDX,RdyQ            ; Add offset of RdyQ => EAX
        ADD EAX,EDX             ; EAX pts to proper Rdy Queue
        CMP DWORD PTR [EAX+Head], 0      ; if Head = NIL
        JNE eqRNotNIL           ; then
        MOV [EAX+Head],EBX      ;  ..Head <= pTSSin;
        MOV [EAX+Tail],EBX      ;  ..Tail <= pTSSin;
        RETN                    ; else

eqRNotNIL:
        MOV EDX,[EAX+Tail]      ;  ..Tail^.NextTSS <= pTSSin;
        MOV [EDX+NextTSS],EBX   ;
        MOV [EAX+Tail],EBX      ;  ..Tail <= pTSSin;

eqRdyDone:
        RETN                    ;
```

deQueueRdy

The deQueueRdy function finds the highest-priority ready queue (out of 32) that has a task waiting there and returns a pointer to the first TSS in that list. The pointer to this TSS is also removed from the list. If none is found, the function returns NIL (0). Keep in mind that 0 is the highest priority and 31 is the lowest. See Listing 18.6.

Listing 18.6. De-queueing the highest priority task.

```
PUBLIC deQueueRdy:
;
; INPUT : NONE
; OUTPUT : EAX
; REGISTERS : EAX,EBX,ECX,FLAGS
; MODIFIES : RdyQ
;
; This routine will return a pointer in EAX to the highest priority task
; queued on the RdyQ. Then the routine will "pop" the TSS from the RdyQ.
; If there was no task queued, EAX is returned as NIL.
;
        MOV ECX,nPRI            ; Set up the number of times to loop
        LEA EBX,RdyQ            ; Get base address of RdyQ in EBX

deRdyLoop:
        MOV EAX,[EBX]           ; Get pTSSout in EAX
        OR  EAX, EAX            ; IF pTSSout is NIL Then go and
        JNZ deRdyFound          ; check the next priority.
        ADD EBX,sQUEUE          ; Point to the next Priority Queue
        LOOP deRdyLoop          ; DEC ECX and LOOP IF NOT ZERO

deRdyFound:
        OR  EAX, EAX            ; IF pTSSout is NIL Then there are
        JZ deRdyDone            ; No TSSs on the RdyQ; RETURN
        DEC _nReady             ;
        MOV ECX,[EAX+NextTSS]   ; Otherwise, deQueue the process
        MOV [EBX],ECX           ; And return with the pointer in EAX

deRdyDone:
        RETN                    ;
```

ChkRdyQ

MMURTL's preemptive nature requires that you have the ability to check the ready queues to see what the highest priority task is that could be executed, without actually removing the task from the queue. This routine provides that functionality for the piece of code in the timer interrupt routine (in TimerCode.ASM) that provides preemptive task switching. This is why it's specified as PUBLIC. See Listing 18.7.

Listing 18.7. Finding the highest priority task.

```
PUBLIC ChkRdyQ:
;
; INPUT : NONE
; OUTPUT : EAX
; REGISTERS : EAX,EBX,ECX,FLAGS
; MODIFIES : RdyQ
;
; This routine will return a pointer to the highest priority TSS that
```

continues

Listing 18.7. continued

```
; is queued to run. It WILL NOT remove it from the Queue.
; If there was no task queued, EAX is returned as NIL.
;
        MOV ECX,nPRI            ; Set up the number of times to loop
        LEA EBX,RdyQ            ; Get base address of RdyQ in EBX

ChkRdyLoop:
        MOV EAX,[EBX]           ; Get pTSSout in EAX
        OR  EAX, EAX            ; IF pTSSout is NIL Then go and
        JNZ ChkRdyDone          ; check the next priority.
        ADD EBX,sQUEUE          ; Point to the next Priority Queue
        LOOP ChkRdyLoop       ; DEC ECX and LOOP IF NOT ZERO

ChkRdyDone:
        RETN                    ;
```

Internal Public Helper Functions

The routines described in the following sections are "public" to the rest of the operating system but not accessible to outside callers. They are located in Kernel.ASM because they work closely with kernel structures.

They provide things such as resource garbage collection and exchange owner manipulation.

RemoveRdyJob

When a job terminates, either by its own choosing or by being killed off due to some unforgivable protection violation, the `RemoveRdyJob` function recovers the task state segments (TSSs) that belonged to that job. "It's a nasty job, but someone's got to it." See Listing 18.8.

Listing 18.8. Removing a terminated task from the ready queue.

```
; RemoveRdyJob  (NEAR)
;
; This routine searches all ready queue priorities for tasks belonging
; to pJCB. When one is found it is removed from the queue
; and the TSS is freed up.  This is called when we are killing
; a job.
;
; Procedural Interface :
;
;       RemoveRdyJob(char *pJCB):ercType
;
;   pJCB is a pointer to the JCB that the tasks to kill belong to.
;
; pJCB          EQU DWORD PTR [EBP+8]
;
; INPUT :  (pJCB on stack)
; OUTPUT : NONE
```

```
; REGISTERS : All general registers are trashed
; MODIFIES : RdyQ
;
;
PUBLIC _RemoveRdyJob:
;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV ECX,nPRI            ; Set up the number of times to loop
        LEA EBX,RdyQ            ; Get base address of RdyQ in EBX
        MOV EDX, [EBP+8]        ; EDX holds pJCB for comparison

        ;EBX points to beginning of next Priority Queue
RemRdyLoop:
        MOV EAX,[EBX+Head]      ; Get pTSS in EAX
        MOV EDI, EAX            ; EDI points to last TSS by default (or NIL)
        OR  EAX,EAX             ; Is pTSS 0 (none left queued here)
        JNZ RemRdy0             ; Valid pTSS!

RemRdyLoop1:
        MOV [EBX+Tail], EDI     ; EDI always points to last TSS or NIL
        ADD EBX,sQUEUE          ; Point to the next Priority Queue
        LOOP RemRdyLoop         ; DEC ECX and LOOP IF NOT ZERO

        XOR EAX, EAX            ; No error
        POP EBP
        RETN 4                  ; All done (clean stack)

        ;Go here to dequeue a TSS at head of list
RemRdy0:
        CMP EDX, [EAX+TSS_pJCB] ; Is this from the JCB we want?
        JNE RemRdy2             ; No

        MOV EDI, [EAX+NextTSS]  ; Yes, deQueue the TSS
        MOV [EBX+Head], EDI     ; Fix link in Queue list

        PUSH EBX                ; Save ptr to RdyQue (crnt priority)

        ;Free up the TSS (add it to the free list)
        MOV EBX,pFreeTSS        ; pTSSin^.Next <= pFreeTSS;
        MOV [EAX+NextTSS],EBX   ;
        MOV DWORD PTR [EAX+TSS_pJCB], 0 ; Make TSS invalid
        MOV pFreeTSS,EAX        ; pFreeTSS <= pTSSin;
        INC _nTSSLeft           ;

        POP EBX
        MOV EAX, EDI            ; Make EAX point to new head TSS
        OR EAX, EAX             ; Is it Zero?
        JZ RemRdyLoop1          ; Next Queue please
        JMP RemRdy0             ; back to check next at head of list

        ;Go here to dequeue a TSS in middle or end of list

RemRdy2:
        MOV EAX, [EDI+NextTSS]  ; Get next link in list
        OR EAX, EAX             ; Valid pTSS?
        JZ RemRdyLoop1          ; No. Next Queue please
```

continues

Listing 18.8. continued

```
        CMP EDX, [EAX+TSS_pJCB] ; Is this from JCB we want?
        JE RemRdy3              ; Yes. Trash it.
        MOV EDI, EAX            ; No. Next TSS
        JMP RemRdy2

RemRdy3:
        ;EDI points to prev TSS
        ;EAX points to crnt TSS
        ;Make ESI point to NextTSS

        MOV ESI, [EAX+NextTSS]  ; Yes, deQueue the TSS

        ;Now we fix the list (Make Prev point to Next)
        ;This extracts EAX from the list

        MOV [EDI+NextTSS], ESI  ;Jump the removed link
        PUSH EBX                ;Save ptr to RdyQue (crnt priority)

        ;Free up the TSS (add it to the free list)
        MOV EBX,pFreeTSS        ; pTSSin^.Next <= pFreeTSS;
        MOV [EAX+NextTSS],EBX           ;
        MOV DWORD PTR [EAX+TSS_pJCB], 0 ; Make TSS invalid
        MOV pFreeTSS,EAX                ; pFreeTSS <= pTSSin;
        INC _nTSSLeft                   ;

        POP EBX
        ;
        OR  ESI, ESI            ;Is EDI the new Tail? (ESI = 0)
        JZ  RemRdyLoop1         ;Yes. Next Queue please
        JMP RemRdy2             ;back to check next TSS
```

GetExchOwner

The code that loads new jobs must have the capability to allocate default exchanges for the new program. The Exchange allocation routines assume that the caller will own this exchange. This is not so if it's for a new job. This call identifies who owns an exchange by returning a pointer to the job control block. This function is used with the next one, `SetExchOwner()`. These are `NEAR` functions and not available to outside callers (via call gates). See Listing 18.9.

Listing 18.9. Finding the owner of an exchange.

```
; GetExchOwner  (NEAR)
;
; This routine returns the owner of the exchange specified.
; A pointer to the JCB of the owner is returned.
; ErcNotAlloc is returned if the exchange isn't allocated.
; ErcOutofRange is returned is the exchange number is invalid (too high)
;
; Procedural Interface :
;
;       GetExchOwner(long Exch, char *pJCBRet): dErrror
```

```
;
;   Exch is the exchange number.
;   pJCBRet is a pointer to the JCB that the tasks to kill belong to.
;
; Exch      EQU DWORD PTR [EBP+12]
; pJCBRet   EQU DWORD PTR [EBP+8]

PUBLIC _GetExchOwner:           ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;

       MOV EAX, [EBP+12]        ; Get Resp Exchange in EDX
       CMP EAX,nExch            ; Is the exchange out of range?
       JB GE001                 ; No, continue
       MOV EAX,ErcOutOfRange    ; Yes, Error in EAX register
       JMP GEOEnd               ;

GE001:
       MOV EDX,sEXCH            ; Compute offset of Exch in rgExch
       MUL EDX                  ; sExch * Exch number
       MOV EDX,prgExch          ; Add offset of rgExch => EAX
       ADD EDX,EAX              ; EDX -> Exch
       MOV EAX, [EDX+Owner]
       OR EAX, EAX              ; Valid Exch (Allocated)
       JNZ GE002
       MOV EAX, ErcNotAlloc     ; No, not allocated
       JMP SHORT GEOEnd

GE002:
       MOV ESI, [EBP+8]         ;Where to return pJCB of Exchange
       MOV [ESI], EAX           ;
       XOR EAX, EAX

GEOEnd:
       MOV ESP,EBP              ;
       POP EBP                  ;
       RETN 8                   ;
```

SetExchOwner

The `SetExchOwner()` function is the complimentary call to `GetExchOwner()`, mentioned previously. This call sets the exchange owner to the owner of the job control block pointed to by the second parameter of the call. See Listing 18.10.

Listing 18.10. Changing the owner of an exchange.

```
; SetExchOwner  (NEAR)
;
; This routine sets the owner of the exchange specified to the
; pJCB specified. This is used by the Job code to set the owner of
; a TSS exchange to a new JCB (even though the exchange was allocated
; by the OS).  No error checking is done because the job code does it upfront!
;
```

continues

Listing 18.10. continued

```
; Procedural Interface :
;
;       SetExchOwner(long Exch, char *pNewJCB): dErrror
;
;   Exch is the exchange number.
;   pNewJCB is a pointer to the JCB of the new owner.
;
; Exch      EQU DWORD PTR [EBP+12]
; pNewJCB   EQU DWORD PTR [EBP+8]

PUBLIC _SetExchOwner:           ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;
       MOV EAX, [EBP+12]        ; Exchange Number
       MOV EDX,sEXCH            ; Compute offset of Exch in rgExch
       MUL EDX                  ; sExch * Exch number
       MOV EDX,prgExch          ; Add offset of rgExch => EAX
       ADD EAX,EDX              ; EAX -> oExch + prgExch
       MOV EBX, [EBP+8]
       MOV [EAX+Owner], EBX
       XOR EAX, EAX
       POP EBP                  ;
       RETN 8                   ;
```

SendAbort

System services are programs that respond to requests from applications, other services, or the operating system itself. System services may hold requests from many jobs at a time until they can service the requests (known as asynchronous servicing).

If a program sent a request, then exited or was killed off because of nasty behavior, the `SendAbort` function is called to send a message to all active services, telling them that a particular job has died. If a service is holding a request from the job that died, the service should respond to it as soon as it gets the abort message. It should not process any data for the program (because the program is dead). The error it returns is of no consequence. The kernel knows the program is already dead and will reclaim the request block and exchanges that were used. See Listing 18.11.

Listing 18.11. Notifying services of a job's demise.

```
; SendAbort  (NEAR)
;
; This routine sends one abort message to each valid service
; with the jobnum of the aborting job. If we receive a
; kernel error on Request it may be because it is a service
; that is aborting itself. We ignore the kernel errors.
;
; Procedural Interface :
;
```

```
;       SendAbort(long JobNum, ValidExch): dErrror
;
;   JobNum is the job that is aborting
;   ValidExch is any valid exchange so the request will go through
;
; JobNum    EQU DWORD PTR [EBP+12]
; ValidExch EQU DWORD PTR [EBP+8]

PUBLIC _SendAbort:              ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        MOV ESI,OFFSET rgSVC    ; Get the address of rgSVC
        MOV ECX,nSVC            ; Get the number of Service Descriptors

SAB01:
        CMP DWORD PTR [ESI], 0  ; Valid name?
        JE SAB05                ; NO, next service

        PUSH ESI                ;Save count and pointer to SVC name
        PUSH ECX
        ;Push all the params to make the request
        PUSH ESI                ;pName
        PUSH 0                  ;Abort Service Code
        MOV EAX, [EBP+8]        ;Exchange
        PUSH EAX
        PUSH OFFSET dJunk       ;pHandleRet
        PUSH 0                  ;npSend
        PUSH 0                  ;pData0
        PUSH 0                  ;cbData0
        PUSH 0                  ;pData1
        PUSH 0                  ;cbData1
        MOV EAX, [EBP+12]       ;JobNum
        PUSH EAX                ;dData0
        PUSH 0                  ;dData1
        PUSH 0                  ;dData2
        CALL FWORD PTR _Request
SAB02:
        PUSH DWORD PTR [EBP+8]
        PUSH OFFSET dmsg
        CALL FWORD PTR _WaitMsg
        MOV EAX, dmsg
        CMP EAX, dJunk    ;see if we got response back!
        JNE SAB02         ;NO - wait again
        ;Get count and ptr to SVC names back from stack
        POP ECX
        POP ESI

SAB05:
        ADD ESI, sSVC           ;Next Service name
        LOOP SAB01
        XOR EAX, EAX
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETN 8                  ;
```

Public Kernel Functions

The remainder of the code defines the kernel public functions that are called through call gates. The public kernel functions are called with 48-bit (FAR) pointers that include the selector of the call gate. Some of the functions are auxiliary functions for outside callers and are not actually part of the kernel code that defines the tasking model; these are functions such as `GetPriority()`. They're in this file because they work closely with the kernel.

The offsets to the stack parameters are defined in the comments for each of the calls.

Request()

The kernel `Request` primitive sends a message like the `Send()` primitive, except that this function requires several more parameters. A system structure called a *request block* is allocated and some of the required parameters are placed in it. A request block is the basic structure used for client/server communications. The exchange where a request should be queued is determined by searching the system-service array for a matching request service name specified in the request block. The function that searches the array is `GetExchange()`, and code for it is in the file SVCCode.asm. See Listing 18.12.

Listing 18.12. `Request` kernel primitive code.

```
; The procedural interface to Request looks like this:
;
;    Request(  pSvcName      [EBP+56]
;              wSvcCode      [EBP+52]
;              dRespExch     [EBP+48]
;              pRqHndlRet    [EBP+44]
;              dnpSend       [EBP+40]
;              pData1        [EBP+36]
;              dcbData1      [EBP+32]
;              pData2        [EBP+28]
;              dcbData2      [EBP+24]
;              dData0        [EBP+20]
;              dData1        [EBP+16]
;              dData2        [EBP+12]  ) : dError

PUBLIC __Request:               ;
       PUSH EBP                 ; Save the Previous FramePtr
       MOV EBP,ESP              ; Set up New FramePtr

       ;Validate service name from registry and get exchange
       MOV EAX, [EBP+56]        ;pServiceName
       CALL GetExchange         ;Leaves Service Exch in ESI if no errors
       OR  EAX,EAX              ;Any errors?
       JZ SHORT Req02           ;No
       JMP ReqEnd               ;Yes, return error

Req02:
       ;Validate exchange
```

```
        MOV EDX, [EBP+48]       ; Get Resp Exchange in EDX
        CMP EDX,nExch           ; Is the exchange out of range?
        JB Req03                ; No, continue
        MOV EAX,ercOutOfRange   ; Yes, Error in EAX register
        JMP ReqEnd              ;

Req03:
        ;Get them a request block
        CLI
        CALL NewRQB             ;EAX has ptr to new RqBlk (or 0 if none)
        STI
        OR  EAX, EAX            ;Did we get one? (NIL (0) means we didn't)
        JNZ Req04               ;Yes. EAX ptr to new RqBlk
        MOV EAX, ErcNoMoreRqBlks ;No, Sorry...
        JMP ReqEnd              ;

Req04:
        ;ESI still has the exchange for the service
        ;EDX still has the response exchange
        ;EAX has pRqBlk (Handle)

        MOV EBX, EAX                ;EBX now pts to RqBlk
        MOV [EBX+ServiceExch], ESI  ;Put Svc Exch into RqBlk
        MOV EAX, [EBP+52]           ;Get Svc Code
        MOV [EBX+ServiceCode], AX   ;Put Svc Code into RqBlk
        MOV [EBX+RespExch], EDX     ;Put Resp Exch into RqBlk
        CALL GetCrntJobNum          ;Get crnt JCB (Job Num of owner)
        MOV [EBX+RqOwnerJob], EAX   ;put in RqBlk
        MOV EAX, [EBP+20]           ;Get dData0
        MOV [EBX+dData0], EAX       ;put in RqBlk
        MOV EAX, [EBP+16]           ;Get dData1
        MOV [EBX+dData1], EAX       ;put in RqBlk
        MOV EAX, [EBP+20]           ;Get dData2
        MOV [EBX+dData2], EAX       ;put in RqBlk
        MOV EAX, [EBP+36]           ;Get pData1
        MOV [EBX+pData1], EAX       ;put in RqBlk
        MOV EAX, [EBP+32]           ;Get cbData1
        MOV [EBX+cbData1], EAX      ;put in RqBlk
        MOV EAX, [EBP+28]           ;Get pData2
        MOV [EBX+pData2], EAX       ;put in RqBlk
        MOV EAX, [EBP+24]           ;Get cbData2
        MOV [EBX+cbData2], EAX      ;put in RqBlk
        MOV EAX, [EBP+40]           ;Number of Send PbCbs
        CMP EAX, 3                  ;Must be 2 or less
        JB Req06                    ;
        MOV EAX, 2

Req06:
        MOV [EBX+npSend], AL        ;Put nSend PbCbs into RqBlk
        MOV CL, 2                   ;Caculate nRecv (2-nSend)
        SUB CL, AL                  ;Leave in CL
        MOV [EBX+npRecv], CL        ;Put npRecv in RqBlk

        ;At this point the RqBlk is all filled in.
        ;Now we will return the RqBlkHandle to the user.
        ;The handle is actually a ptr to the RqBlk but they can't use
        ;it as one anyway (so no problem)
```

continues

Listing 18.12. continued

```
        MOV EDI, [EBP+44]           ;Ptr to return handle to
        MOV [EDI], EBX              ;Give it to them
        MOV EDX, EBX                ;Save RqBlk in EDX

        CLI                         ; No interruptions from here on
        ;Now we allocate a Link block to use

        MOV EAX,pFreeLB             ; EAX <= pFreeLB;
        OR EAX,EAX                  ; Is pFreeLB NIL? (out of LBs)
        JNZ Req08                   ;
        CALL DisposeRQB             ; NO... free up RqBlk
        MOV EAX,ercNoMoreLBs        ; Move error in the EAX register
        JMP ReqEnd                  ; Go home with bad news

Req08:
        MOV EBX,[EAX+NextLB]        ; pFreeLB <= pFreeLB^.Next
        MOV pFreeLB,EBX             ;
        DEC _nLBLeft                ;

        MOV DWORD PTR [EAX+LBType],REQLB    ; This is a Request Link Block
        MOV DWORD PTR [EAX+NextLB], 0       ; pLB^.Next <= NIL;
        MOV [EAX+DataLo],EDX            ; RqHandle into Lower 1/2 of Msg
        MOV DWORD PTR [EAX+DataHi], 0   ; Store zero in upper half of pLB^.Data
        PUSH EAX                        ; Save pLB on the stack

        ;ESI still has the exchange Number for the service.
        ;The ptr to the exch is required for deQueueTSS so we get it.

        MOV EAX,ESI             ; Exch => EAX
        MOV EDX, sEXCH          ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ESI,EAX             ; MAKE ESI <= pExch

        ;ESI now points to the exchange

        CALL deQueueTSS             ; DeQueue a TSS on that Exch
        OR  EAX,EAX                 ; Did we get one?
        JNZ Req10                   ; Yes, give up the message
        POP EAX                     ; No, Get the pLB just saved
        CALL enQueueMsg             ; EnQueue the Message on Exch
        XOR EAX,EAX                 ; No Error
        JMP SHORT ReqEnd            ; And get out!

Req10:
        POP EBX                     ; Get the pLB just saved into EBX
        MOV [EAX+pLBRet],EBX        ; and put it in the TSS
        CALL enQueueRdy             ; EnQueue the TSS on the RdyQ
        MOV EAX,pRunTSS             ; Get the Ptr To the Running TSS
        CALL enQueueRdy             ; and put him on the RdyQ
        CALL deQueueRdy             ; Get high priority TSS off the RdyQ
        CMP EAX,pRunTSS             ; If the high priority TSS is the
        JNE Req12                   ; same as the Running TSS then return
        XOR EAX,EAX                 ; Return to Caller with erc ok.
        JMP SHORT ReqEnd
```

```
Req12:
        MOV pRunTSS,EAX         ; Make the TSS in EAX the Running TSS
        MOV BX,[EAX+Tid]        ; Get the task Id (TR)
        MOV TSS_Sel,BX          ; Put it in the JumpAddr for Task Switch
        INC _nSwitches          ; Keep track of how many switches for stats
        MOV EAX, TimerTick      ; Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]     ; JMP TSS (This is the task switch)
        XOR EAX,EAX             ; Return to Caller with erc ok.

ReqEnd:
        STI                     ;
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 48                 ; Rtn to Caller & Remove Params from stack
```

Respond()

The `Respond` primitive is used by system services to respond to a `Request()` received at their service exchange. The request block handle must be supplied along with the error/status code to be returned to the caller. This call is very similar to `Send()` except that it de-aliases addresses in the request block and then deallocates it. The exchange to respond to is located inside the request block. See Listing 18.13.

Listing 18.13. `Respond` kernel primitive code.

```
;
;     Respond(dRqHndl, dStatRet): dError
;
;
dRqHndl  EQU DWORD PTR [EBP+16]
dStatRet EQU DWORD PTR [EBP+12]

PUBLIC __Respond:               ;
        PUSH EBP                ; Save Callers Frame
        MOV EBP,ESP             ; Setup Local Frame
        MOV EAX, dRqHndl        ; pRqBlk into EAX
        MOV EBX, dStatRet
        CMP EBX, ErcOwnerAbort  ;
        JNE Resp01
        CLI                     ; No interruptions
        CALL DisposeRQB         ; Return Aborted RQB to pool.
        XOR EAX, EAX            ; No Error
        JMP RespEnd             ; Get out
Resp01:
        MOV ESI, [EAX+RespExch] ; Response Exchange into ESI
        CMP ESI,nExch           ; Is the exchange out of range?
        JNAE Resp02             ; No, continue
        MOV EAX,ercOutOfRange   ; Error into the EAX register.
        JMP RespEnd             ; Get out
```

continues

Listing 18.13. continued

```
Resp02:
        MOV EAX,ESI             ; Exch => EAX
        MOV EDX,sEXCH           ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ESI,EAX             ; MAKE ESI <= pExch
        CMP DWORD PTR [EAX+Owner], 0    ; If the exchange is not allocated
        JNE Resp04              ; return to the caller with error
        MOV EAX,ercNotAlloc     ; in the EAX register.
        JMP RespEnd             ;

Resp04:
        MOV EAX, dRqHndl        ; Get Request handle into EBX (pRqBlk)
        MOV EBX, [EAX+RqOwnerJob]
        CALL GetCrntJobNum
        CMP EAX, EBX
        JE Resp06               ;Same job - no DeAlias needed

        MOV EAX, dRqHndl        ; Get Request handle into EBX (pRqBlk)
        MOV EBX, [EAX+cbData1]  ;
        OR EBX, EBX
        JZ Resp05               ;No need to dealias (zero bytes)
        MOV EDX, [EAX+pData1]
        OR EDX, EDX
        JZ Resp05               ;Null pointer!

        PUSH ESI                ;Save pExch across call
        PUSH EDX                ;pMem
        PUSH EBX                ;cbMem
        CALL GetCrntJobNum
        PUSH EAX
        CALL FWORD PTR _DeAliasMem  ;DO it and ignore errors
        POP ESI                 ;get pExch back

Resp05:
        MOV EAX, dRqHndl        ; Get Request handle into EBX (pRqBlk)
        MOV EBX, [EAX+cbData2]  ;
        OR EBX, EBX
        JZ Resp06               ;No need to dealias (zero bytes)
        MOV EDX, [EAX+pData2]
        OR EDX, EDX
        JZ Resp06               ;Null pointer!
        PUSH ESI                ;Save pExch across call
        PUSH EDX                ;pMem
        PUSH EBX                ;cbMem
        CALL GetCrntJobNum      ;
        PUSH EAX
        CALL FWORD PTR _DeAliasMem  ;DO it and ignore errors
        POP ESI                 ;get pExch back

Resp06:
        MOV EAX, dRqHndl        ; Get Request handle into EBX (pRqBlk)
        CLI                     ; No interruptions
        CALL DisposeRQB         ; Return Rqb to pool. Not needed anymore
```

```
        ; Allocate a link block
        MOV EAX,pFreeLB         ; NewLB <= pFreeLB;
        OR EAX,EAX              ; IF pFreeLB=NIL THEN No LBs;
        JNZ Resp07              ;
        MOV EAX,ercNoMoreLBs    ; caller with error in the EAX register
        JMP RespEnd

Resp07:
        MOV EBX,[EAX+NextLB]    ; pFreeLB <= pFreeLB^.Next
        MOV pFreeLB,EBX         ;
        DEC _nLBLeft            ; Update stats
        MOV DWORD PTR [EAX+LBType], RESPLB ; This is a Response Link Block
        MOV DWORD PTR [EAX+NextLB], 0      ; pLB^.Next <= NIL;
        MOV EBX, dRqHndl        ; Get Request handle into EBX
        MOV [EAX+DataLo],EBX    ; Store in lower half of pLB^.Data
        MOV EBX, dStatRet       ; Get Status/Error into EBX
        MOV [EAX+DataHi],EBX    ; Store in upper half of pLB^.Data
        PUSH EAX                ; Save pLB on the stack
        CALL deQueueTSS         ; DeQueue a TSS on that Exch
        OR  EAX,EAX             ; Did we get one?
        JNZ Resp08              ; Yes, give up the message
        POP EAX                 ; Get the pLB just saved
        CALL enQueueMsg         ; EnQueue the Message on Exch
        XOR EAX, EAX            ; No Error
        JMP SHORT RespEnd       ; And get out!

Resp08:
        POP EBX                 ; Get the pLB just saved into EBX
        MOV [EAX+pLBRet],EBX    ; and put it in the TSS
        CALL enQueueRdy         ; EnQueue the TSS on the RdyQ
        MOV EAX,pRunTSS         ; Get the Ptr To the Running TSS
        CALL enQueueRdy         ; and put him on the RdyQ
        CALL deQueueRdy         ; Get high priority TSS off the RdyQ

        CMP EAX,pRunTSS         ; If the high priority TSS is the
        JNE Resp10              ; same as the Running TSS then return
        XOR EAX,EAX             ; Return to Caller with erc ok.
        JMP SHORT RespEnd       ;

Resp10:
        MOV pRunTSS,EAX         ; Make the TSS in EAX the Running TSS
        MOV BX,[EAX+Tid]        ; Get the task Id (TR)
        MOV TSS_Sel,BX          ; Put it in the JumpAddr
        INC _nSwitches
        MOV EAX, TimerTick      ;Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]     ; JMP TSS
        XOR EAX,EAX             ; Return to Caller with erc ok.

RespEnd:
        STI
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ; Rtn to Caller & Remove Params
```

MoveRequest()

This is the kernel `MoveRequest` primitive. The `MoveRequest` primitive allows a service to move a request to another exchange it owns. This cannot be used to forward a request to another service or job, because the pointers in the request block are not properly aliased. `MoveRequest()` is very similar to `SendMsg()` except that `MoveRequest()` checks to ensure the destination exchange is owned by the sender. See Listing 18.14.

Listing 18.14. `MoveRequest` kernel primitive code.

```
;
; Procedural Interface :
;
;      MoveRequest(dRqBlkHndl, DestExch):ercType
;
;           dqMsg is the handle of the RqBlk to forward.
;           DestExch the exchange to where the Request should be sent.
;
;
;dRqBlkHndl     EQU [EBP+16]
;DestExch       EQU [EBP+12]

PUBLIC __MoveRequest:              ;
        PUSH EBP                  ;
        MOV EBP,ESP               ;
        MOV ESI, [EBP+12]         ; Get Exchange Parameter in ESI
        CMP ESI,nExch             ; Is the exchange out of range
        JNAE MReq02               ; No, continue
        MOV EAX,ercOutOfRange     ; in the EAX register.
        JMP MReqEnd               ; Get out

MReq02:
        MOV EAX,ESI               ; Exch => EAX
        MOV EDX,sEXCH             ; Compute offset of Exch in rgExch
        MUL EDX                   ;
        MOV EDX,prgExch           ; Add offset of rgExch => EAX
        ADD EAX,EDX               ;
        MOV ESI,EAX               ; MAKE ESI <= pExch
        MOV EDX, [EAX+Owner]      ; Put exch owner into EDX (pJCB)
        CALL GetpCrntJCB          ; Leaves it in EAX (uses only EAX)
        CMP EDX, EAX              ; If the exchange is not owned by sender
        JE  MReq04                ; return to the caller with error
        MOV EAX, ErcNotOwner      ; in the EAX register.
        JMP MReqEnd               ; Get out

MReq04:
        CLI                       ; No interruptions from here on
        ; Allocate a link block
        MOV EAX,pFreeLB           ; NewLB <= pFreeLB;
        OR EAX,EAX                ; IF pFreeLB=NIL THEN No LBs;
        JNZ MReq08                ;
        MOV EAX,ercNoMoreLBs      ; caller with error in the EAX register
        JMP MReqEnd               ; Go home with bad news
```

```
MReq08:
        MOV EBX,[EAX+NextLB]    ; pFreeLB <= pFreeLB^.Next
        MOV pFreeLB,EBX         ;
        DEC _nLBLeft            ; Update stats

        MOV DWORD PTR [EAX+LBType], REQLBA  ; This is a Request Link Block Aliased
        MOV DWORD PTR [EAX+NextLB], 0       ; pLB^.Next <= NIL;
        MOV EBX, [EBP+16]                   ; RqHandle
        MOV [EAX+DataLo],EBX            ; RqHandle into Lower 1/2 of Msg
        MOV DWORD PTR [EAX+DataHi], 0   ; Store zero in upper half of pLB^.Data
        PUSH EAX                    ; Save pLB on the stack
        CALL deQueueTSS             ; DeQueue a TSS on that Exch
        OR  EAX,EAX                 ; Did we get one?
        JNZ MReq10                  ; Yes, give up the message
        POP EAX                     ; Get the pLB just saved
        CALL enQueueMsg             ; EnQueue the Message on Exch
        JMP SHORT MReqEnd           ; And get out!

MReq10:
        POP EBX                     ; Get the pLB just saved into EBX
        MOV [EAX+pLBRet],EBX        ; and put it in the TSS
        CALL enQueueRdy             ; EnQueue the TSS on the RdyQ
        MOV EAX,pRunTSS             ; Get the Ptr To the Running TSS
        CALL enQueueRdy             ; and put him on the RdyQ
        CALL deQueueRdy             ; Get high priority TSS off the RdyQ
        CMP EAX,pRunTSS             ; If the high priority TSS is the
        JNE MReq12                  ; same as the Running TSS then return
        XOR EAX,EAX                 ; Return to Caller with erc ok.
        JMP SHORT MReqEnd

MReq12:
        MOV pRunTSS,EAX         ; Make the TSS in EAX the Running TSS
        MOV BX,[EAX+Tid]        ; Get the task Id (TR)
        MOV TSS_Sel,BX          ; Put it in the JumpAddr for Task Switch
        INC _nSwitches          ; Keep track of how many switches for stats
        MOV EAX, TimerTick      ; Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]     ; JMP TSS (This is the task switch)
        XOR EAX,EAX             ; Return to Caller with erc ok.

MReqEnd:
        STI                     ;
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
```

SendMsg()

This is the kernel `SendMsg` primitive. It sends a nonspecific message from a running task to an exchange. The `SendMsg` primitive may cause a task switch, if a task is waiting at the exchange and it is of equal or higher priority than that of the task which sent the message. See Listing 18.15.

Listing 18.15. SendMsg kernel primitive code.

```
;
; Procedural Interface :
;
;       SendMsg(exch, dMsg1, dMsg2):ercType
;
;           exch is a DWORD (4 BYTES) containing the exchange to where the
;           message should be sent.
;
;           dMsg1 & dMsg2 are DWord values defined and understood
;           only by the sending and receiving tasks.
;
SendExchange    EQU [EBP+14h]
MessageHi       EQU DWORD PTR [EBP+10h]
MessageLo       EQU DWORD PTR [EBP+0Ch]

PUBLIC __SendMsg:                       ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV ESI,SendExchange    ; Get Exchange Parameter in ESI
        CMP ESI,nExch           ; If the exchange is out of range
        JNAE Send00             ; the return to caller with error
        MOV EAX,ercOutOfRange   ; in the EAX register.
        JMP SendEnd

Send00:
        MOV EAX,ESI             ; Exch => EAX
        MOV EDX,sEXCH           ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ESI,EAX             ; MAKE ESI <= pExch
        CMP DWORD PTR [EAX+Owner], 0    ; If the exchange is not allocated
        JNE Send01              ; return to the caller with error
        MOV EAX,ercNotAlloc     ; in the EAX register.
        JMP SendEnd             ;

Send01:
        CLI                     ; No interrupts
        ; Allocate a link block
        MOV EAX,pFreeLB         ; NewLB <= pFreeLB;
        OR EAX,EAX              ; IF pFreeLB=NIL THEN No LBs;
        JNZ SHORT Send02        ;
        MOV EAX,ercNoMoreLBs    ; caller with error in the EAX register
        JMP SHORT MReqEnd       ; Go home with bad news

Send02:
        MOV EBX,[EAX+NextLB]    ; pFreeLB <= pFreeLB^.Next
        MOV pFreeLB,EBX         ;
        DEC _nLBLeft            ; Update stats

        MOV DWORD PTR [EAX+LBType], DATALB ; This is a Data Link Block
        MOV DWORD PTR [EAX+NextLB], 0      ; pLB^.Next <= NIL;
        MOV EBX,MessageLo       ; Get lower half of Msg in EBX
        MOV [EAX+DataLo],EBX    ; Store in lower half of pLB^.Data
        MOV EBX,MessageHi       ; Get upper half of Msg in EBX
        MOV [EAX+DataHi],EBX    ; Store in upper half of pLB^.Data
```

```
        PUSH EAX                ; Save pLB on the stack

        CLI                     ; No interrupts
        CALL deQueueTSS         ; DeQueue a TSS on that Exch
        STI
        OR  EAX,EAX             ; Did we get one?
        JNZ Send25              ; Yes, give up the message
        POP EAX                 ; Get the pLB just saved
        CLI                     ; No interrupts
        CALL enQueueMsg         ; EnQueue the Message on Exch
        JMP Send04              ; And get out (Erc 0)!

Send25:
        POP EBX                 ; Get the pLB just saved into EBX
        CLI                     ; No interrupts
        MOV [EAX+pLBRet],EBX    ; and put it in the TSS
        CALL enQueueRdy         ; EnQueue the TSS on the RdyQ
        MOV EAX,pRunTSS         ; Get the Ptr To the Running TSS
        CALL enQueueRdy         ; and put him on the RdyQ
        CALL deQueueRdy         ; Get high priority TSS off the RdyQ
        CMP EAX,pRunTSS         ; If the high priority TSS is the
        JNE Send03              ; same as the Running TSS then return
        JMP SHORT Send04        ; Return with ErcOk

Send03:
        MOV pRunTSS,EAX         ; Make the TSS in EAX the Running TSS
        MOV BX,[EAX+Tid]        ; Get the task Id (TR)
        MOV TSS_Sel,BX          ; Put it in the JumpAddr
        INC _nSwitches
        MOV EAX, TimerTick      ;Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]     ; JMP TSS

Send04:
        XOR EAX,EAX             ; Return to Caller with erc ok.

SendEnd:
        STI                     ;
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;
```

ISendMsg()

This is the kernel `ISendMsg` primitive (Interrupt Send). This procedure allows an ISR to send a message to an exchange. This is the same as `SendMsg()` except that *no* task switch is performed. If a task is waiting at the exchange, the message is associated (linked) with the task and then moved to the `RdyQ`. It will get a chance to run the next time the `RdyQ` is evaluated by the kernel, which will probably be caused by the timer interrupt slicer.

Interrupt tasks can use `ISendMsg()` to send single or multiple messages to exchanges during their execution. Interrupts are *cleared* on entry and will not be set on exit! It is the responsibility of the caller to set them, if desired. `ISendMsg` is intended to be used only by ISRs in device drivers.

You may notice that jump instructions are avoided and code is placed in-line, where possible, for speed. See Listing 18.16.

Listing 18.16. `ISendMsg` kernel primitive code.

```
; Procedural Interface :
;
;       ISendMsg(exch, dMsg1, dMsg2):ercType
;
;           exch is a DWORD (4 BYTES) containing the exchange to where the
;           message should be sent.
;
;           dMsg1 and dMsg2 are DWORD messages.
;
; Parameters on stack are the same as _SendMsg.

PUBLIC __ISendMsg:                      ;
        CLI                     ;INTS ALWAYS CLEARED AND LEFT THAT WAY!
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV ESI,SendExchange    ; Get Exchange Parameter in ESI
        CMP ESI,nExch           ; If the exchange is out of range
        JNAE ISend00            ; then return to caller with error
        MOV EAX,ercOutOfRange   ; in the EAX register.
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;

ISend00:
        MOV EAX,ESI             ; Exch => EAX
        MOV EDX,sEXCH           ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ESI,EAX                     ; MAKE ESI <= pExch
        CMP DWORD PTR [EAX+Owner], 0    ; If the exchange is not allocated
        JNE ISend01                     ; return to the caller with error
        MOV EAX,ercNotAlloc     ; in the EAX register.
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;

ISend01:
        ; Allocate a link block
        MOV EAX,pFreeLB         ; NewLB <= pFreeLB;
        OR EAX,EAX              ; IF pFreeLB=NIL THEN No LBs;
        JNZ SHORT ISend02       ;
        MOV EAX,ercNoMoreLBs    ; caller with error in the EAX register
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;

ISend02:
        MOV EBX,[EAX+NextLB]    ; pFreeLB <= pFreeLB^.Next
        MOV pFreeLB,EBX         ;
        DEC _nLBLeft            ;
```

```
        MOV DWORD PTR [EAX+LBType],DATALB ; This is a Data Link Block
        MOV DWORD PTR [EAX+NextLB],0      ; pLB^.Next <= NIL;
        MOV EBX,MessageLo       ; Get lower half of Msg in EBX
        MOV [EAX+DataLo],EBX    ; Store in lower half of pLB^.Data
        MOV EBX,MessageHi       ; Get upper half of Msg in EBX
        MOV [EAX+DataHi],EBX    ; Store in upper half of pLB^.Data
        PUSH EAX                ; Save pLB on the stack
        CALL deQueueTSS         ; DeQueue a TSS on that Exch
        OR  EAX,EAX             ; Did we get one?
        JNZ ISend03             ; Yes, give up the message
        POP EAX                 ; No, Get the pLB just saved
        CALL enQueueMsg         ; EnQueue the Message on Exch
        JMP ISend04             ; And get out!

ISend03:
        POP EBX                 ; Get the pLB just saved into EBX
        MOV [EAX+pLBRet],EBX    ; and put it in the TSS
        CALL enQueueRdy         ; EnQueue the TSS on the RdyQ

ISend04:
        XOR EAX,EAX             ; Return to Caller with erc ok.
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;
```

WaitMsg()

This is the kernel `WaitMsg` primitive. This procedure allows a task to receive information from another task via an exchange. If no message is at the exchange, the task is placed on the exchange, and the ready queue is reevaluated to make the next highest-priority task run.

The `WaitMsg` code is a very key piece to task scheduling. You will notice that if no task is ready to run, I simply halt the processor with interrupts enabled. A single flag named `fHalted` is set to let the interrupt slicer know that it doesn't have to check for a higher-priority task. If you halted, there wasn't one to begin with.

The `WaitMsg()` and `CheckMsg()` primitives also have the job of aliasing memory addresses inside of request blocks for system services. This is because they may be operating in completely different memory contexts, which means they have different page directories. See Listing 18.17.

Listing 18.17. `WaitMsg` kernel primitive code

```
;
; A result code is returned in the EAX register.
;
; Procedural Interface :
;
;       Wait(dExch,pdqMsgRet):ercType
;
;           dExch is a DWORD (4 BYTES) containing the exchange to
```

continues

Listing 18.17. continued

```
;           where the message should be sent.
;
;           pMessage is a pointer to an 8 byte area where the
;           message is stored.
;
WaitExchange    EQU [EBP+10h]
pMessage        EQU [EBP+0Ch]
;
;
PUBLIC __WaitMsg:                       ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV ESI,WaitExchange    ; Get Exchange Parameter in ESI
        CMP ESI,nExch           ; If the exchange is out of range
        JNAE Wait00               ; the return to caller with error
        MOV EAX,ercOutOfRange   ; in the EAX register.
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;

Wait00:
        MOV EAX,ESI             ; ExchId => EAX
        MOV EBX,sEXCH           ; Compute offset of ExchId in rgExch
        MUL EBX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ESI,EAX             ; Put Exch in to ESI
        CMP DWORD PTR [EAX+Owner], 0   ; If the exchange is not allocated
        JNE Wait01              ; return to the caller with error
        MOV EAX,ercNotAlloc     ; in the EAX register.
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;

Wait01:
        CLI                     ;
        CALL deQueueMsg         ; EAX <= pLB from pExch (ESI)
        OR  EAX,EAX             ; If no message (pLB = NIL) Then
        JZ Wait02               ; Wait for Message Else
        JMP Wait05              ; Get Message and Return

Wait02:
        MOV EAX,pRunTSS         ; Get pRunTSS in EAX to Wait

        ;This next section of code Queues up the TSS pointed to
        ;by EAX on the exchange pointed to by ESI
        ; (i.e., we make the current task "wait")

        MOV DWORD PTR [EAX+NextTSS], 0  ; pTSSin^.Next <= NIL;
        XCHG ESI,EAX                    ; pExch => EAX, pTSSin => ESI
        CMP DWORD PTR [EAX+EHead], 0    ; if ..TSSHead = NIL
        JNE Wait025                     ; then
        MOV [EAX+EHead],ESI     ;  ..TSSHead <= pTSSin;
        MOV [EAX+ETail],ESI     ;  ..TSSTail <= pTSSin;
        MOV DWORD PTR [EAX+fEMsg], 0        ; Flag it as a TSS (vice a Msg)
```

```
        XCHG ESI,EAX            ; Make ESI <= pExch Again
        JMP SHORT Wait03                ; else

Wait025:
        MOV EDX,[EAX+ETail]     ;  ..TSSTail^.NextTSS <= pTSSin;
        MOV [EDX+NextTSS],ESI   ;
        MOV [EAX+ETail],ESI     ;  ..TSSTail <= pTSSin;
        MOV DWORD PTR [EAX+fEMsg], 0        ; Flag it as a TSS (vice a Msg)
        XCHG ESI,EAX            ; Make ESI <= pExch Again

        ;We just placed the current TSS on an exchange,
        ;now we get the next TSS to run (if there is one)

Wait03:
        CALL deQueueRdy         ; Get highest priority TSS off the RdyQ
        OR EAX, EAX             ; Anyone ready to run?
        JNZ Wait035             ; Yes (jump to check pTSS)

        MOV EDI, 1
        MOV dfHalted, EDI
        INC _nHalts
        STI                     ; No, then HLT CPU until ready
        HLT                     ; Halt CPU and wait for interrupt
        CLI                     ; An interrupt has occurred. Clear Interrupts
        XOR EDI,EDI
        MOV dfHalted, EDI
        JMP Wait03              ; Check for a task to switch to

Wait035:
        CMP EAX,pRunTSS         ; Same one as before???
        JE Wait04               ; You bet! NO SWITCH!!!

        ;Now we switch tasks by placing the address of the
        ;new TSS in pRunTSS and jumping to it.  This forces
        ;a hardware task switch.

        MOV pRunTSS,EAX         ; Make high priority TSS Run.
        MOV BX,[EAX+Tid]        ;
        MOV TSS_Sel,BX          ;
        INC _nSwitches
        MOV EAX, TimerTick      ;Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]     ; JUMP TSS (Switch Tasks)

        ; A task has just finished "Waiting"
        ; We are now in the new task with its memory space
        ; (or the same task if he was high pri & had a msg)
        ; If this is a system service it may need RqBlk address aliases
        ; If it is an OS service we alias in OS memory!

Wait04:
        MOV EDX,pRunTSS         ; Put the TSS in EAX into EDX
        MOV EAX,[EDX+pLBRet]    ; Get the pLB in EAX

Wait05:
        ; If we got here, we have either switched tasks
        ; and we are delivering a message (or Req) to the new task,
```

continues

Listing 18.17. continued

```
        ; or there was a message waiting at the exch of
        ; the first caller and we are delivering it.
        ; Either way, the message is already deQueued from
        ; the exch and the critical part of WaitMsg is over.
        ; We can start interrupts again except when we have
        ; to return the Link Block to the pool (free it up)

        STI                                 ; WE CAN RESTART INTERRUPTS HERE
        CMP DWORD PTR [EAX+LBType],REQLB    ; Is the link block a Req Link Block?
        JNE Wait06                          ; No, Treat it as a data link block

        ;pLB.DataLo is RqHandle (pRqBlk)

        PUSH EAX                ; Save ptr to Link Block
        MOV EBX,[EAX+DataLo]    ; Get pRqBlk into EBX

        ;Now we set up to alias the memory for the service
        ; (Alias the 2 Pointers in the RqBlk)
        ;_AliasMem(pMem, dcbMem, dJobNum, ppAliasRet): dError

        MOV ECX, [EBX+cbData1]      ;
        OR ECX, ECX                 ;is cbData1 0?
        JZ Wait051                  ;Yes

        MOV EAX, [EBX+pData1]       ;
        OR EAX, EAX                 ;is pData1 NULL?
        JZ Wait051                  ;Yes

                                    ;Set up params for AliasMem
        PUSH EAX                    ;pMem
        PUSH ECX                    ;cbMem
        MOV EAX, [EBX+RqOwnerJob]
        PUSH EAX                    ;dJobNum
        ADD EBX, pData1             ;Offset to pData1 in RqBlk
        PUSH EBX                    ;Linear Address of pData1
        CALL FWORD PTR _AliasMem
        OR EAX, EAX                 ;Error??
        JZ Wait051                  ;No, continue
        POP EBX                     ;Make stack right
        MOV ESP,EBP                 ;Return Error...
        POP EBP                     ;
        RETF 8                      ;

Wait051:                            ;Second Pointer (pData2)
        POP EAX                     ;Restore ptr to Link Block
        PUSH EAX                    ;Save again
        MOV EBX,[EAX+DataLo]        ; Get pRqBlk into EBX

        MOV ECX, [EBX+cbData2]      ;
        OR ECX, ECX                 ;is cbData2 0?
        JZ Wait052                  ;Yes

        MOV EAX, [EBX+pData2]       ;
        OR EAX, EAX                 ;is pData2 NULL?
        JZ Wait052                  ;Yes
                                    ;Set up params for AliasMem
```

```
        PUSH EAX                    ;pMem
        PUSH ECX                    ;cbMem
        MOV EAX, [EBX+RqOwnerJob]
        PUSH EAX                    ;dJobNum
        ADD EBX, pData2             ;Offset to pData2 in RqBlk
        PUSH EBX                    ;Linear Address of PData1
        CALL FWORD PTR _AliasMem
        OR EAX, EAX                 ;Error??
        JZ Wait052                  ;No, continue
        POP EBX                     ;Make stack right
        MOV ESP,EBP                 ;Return Error...
        POP EBP                     ;
        RETF 8                      ;

Wait052:
        POP EAX                 ;Restore ptr to Link Block

Wait06:
        MOV EBX,[EAX+DataLo]    ; Get pLB^.Data into ECX:EBX
        MOV ECX,[EAX+DataHi]    ;
        MOV EDX,pMessage        ; Get Storage Addr in EDX
        MOV [EDX],EBX           ; Put pLB^.Data in specified
        MOV [EDX+4],ECX         ; memory space (EDX)

        ;Return the LB to the pool
        CLI
        MOV EBX,pFreeLB         ; pLBin^.Next <= pFreeLB;
        MOV [EAX+NextLB],EBX    ;
        MOV pFreeLB,EAX         ; pFreeLB <= pLBin;
        INC _nLBLeft            ;
        STI                     ;
        XOR EAX,EAX             ; ErcOK! (0)
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
```

CheckMsg()

This is the kernel `CheckMsg` primitive. This procedure allows a task to receive information from another task without blocking. In other words, if no message is available, `CheckMsg()` returns to the caller. If a message *is* available it is returned to the caller immediately.

The caller is never placed on an exchange, and the ready queue is not evaluated. As with `WaitMsg()`, address aliasing must also be accomplished here for system services if they have different page directories. See Listing 18.18.

Listing 18.18. `CheckMsg` kernel primitive code.

```
; A result code is returned in the EAX register.
;
```

continues

Listing 18.18. continued

```
; Procedureal Interface :
;
;       CheckMsg(exch,pdqMsg):ercType
;
;           exch is a DWORD (4 BYTES) containing the exchange to where the
;           message should be sent.
;
;           pdqMsg is a pointer to an 8 byte area where the message is stored.
;
ChkExchange   EQU [EBP+10h]
pCkMessage    EQU [EBP+0Ch]

PUBLIC __CheckMsg:                ;
       PUSH EBP                   ;
       MOV EBP,ESP                ;
       MOV ESI,ChkExchange        ; Get Exchange Parameter in ESI
       CMP ESI,nExch              ; If the exchange is out of range
       JNAE Chk01                 ; then return to caller with error
       MOV EAX,ercOutOfRange      ; in the EAX register.
       MOV ESP,EBP                ;
       POP EBP                    ;
       RETF 8                     ;
Chk01:
       MOV EAX,ESI                ; Exch => EAX
       MOV EBX,sEXCH              ; Compute offset of Exch in rgExch
       MUL EBX                    ;
       MOV EDX,prgExch            ; Add offset of rgExch => EAX
       ADD EAX,EDX                ;
       MOV ESI,EAX                        ; Put pExch in to ESI
       CMP DWORD PTR [EAX+Owner], 0       ; If the exchange is not allocated
       JNE Chk02                          ; return to the caller with error

       MOV EAX,ercNotAlloc        ; in the EAX register.
       MOV ESP,EBP                ;
       POP EBP                    ;
       RETF 8                     ;
Chk02:
       CLI                        ; Can't be interrupted
       CALL deQueueMsg            ; EAX <= pLB from pExch (ESI)
       OR  EAX,EAX                ; If pLB = NIL Then
       JNZ Chk03                  ; Go to get msg and return

       STI                        ;
       MOV EAX,ercNoMsg           ; return with erc no msg
       MOV ESP,EBP                ;
       POP EBP                    ;
       RETF 8                     ;
Chk03:
       STI                        ;We can be interrupted again

       CMP DWORD PTR [EAX+LBType],REQLB    ; Is the link block a Req Link Block?
       JNE Chk04                  ; No, Treat it as a data link block

       ;pLB.DataLo is RqHandle (pRqBlk)

       PUSH EAX                   ; Save ptr to Link Block
       MOV EBX,[EAX+DataLo]       ; Get pRqBlk into EBX
```

```
        ;Now we set up to alias the memory for the service
        ; (Alias the 2 Pointers in the RqBlk)
        ;_AliasMem(pMem, dcbMem, dJobNum, ppAliasRet): dError

        MOV ECX, [EBX+cbData1]      ;
        OR ECX, ECX                 ;is cbData1 0?
        JZ Chk031                   ;Yes

        MOV EAX, [EBX+pData1]       ;
        OR EAX, EAX                 ;is pData1 NULL?
        JZ Chk031                   ;Yes
                                    ;Set up params for AliasMem
        PUSH EAX                    ;pMem
        PUSH ECX                    ;cbMem
        MOV EAX, [EBX+RqOwnerJob]
        PUSH EAX                    ;dJobNum
        ADD EBX, pData1             ;Offset to pData1 in RqBlk
        PUSH EBX                    ;Linear Address of pData1
        CALL FWORD PTR _AliasMem
        OR EAX, EAX                 ;Error??
        JZ Chk031                   ;No, continue
        POP EBX                     ;Make stack right
        MOV ESP,EBP                 ;Return Error...
        POP EBP                     ;
        RETF 8                      ;

Chk031:                             ;Second Pointer (pData2)
        POP EAX                     ;Restore ptr to Link Block
        PUSH EAX                    ;Save again
        MOV EBX,[EAX+DataLo]        ; Get pRqBlk into EBX

        MOV ECX, [EBX+cbData2]      ;
        OR ECX, ECX                 ;is cbData2 0?
        JZ Chk032                   ;Yes

        MOV EAX, [EBX+pData2]       ;
        OR EAX, EAX                 ;is pData2 NULL?
        JZ Chk032                   ;Yes
                                    ;Set up params for AliasMem
        PUSH EAX                    ;pMem
        PUSH ECX                    ;cbMem
        MOV EAX, [EBX+RqOwnerJob]
        PUSH EAX                    ;dJobNum
        ADD EBX, pData2             ;Offset to pData2 in RqBlk
        PUSH EBX                    ;Linear Address of PData1
        CALL FWORD PTR _AliasMem
        OR EAX, EAX                 ;Error??
        JZ Chk032                   ;No, continue
        POP EBX                     ;Make stack right
        MOV ESP,EBP                 ;Return Error...
        POP EBP                     ;
        RETF 8                      ;

Chk032:
        POP EAX                     ;Restore Ptr to Link Block
```

continues

Listing 18.18. continued

```
Chk04:
        MOV EBX,[EAX+DataLo]    ; Get pLB^.Data into ECX:EBX
        MOV ECX,[EAX+DataHi]    ;
        MOV EDX,pCkMessage      ; Get Storage Addr in EDX
        MOV [EDX],EBX           ; Put pLB^.Data in specified
        MOV [EDX+4],ECX         ; memory space (EDX)

        ;Return the LB to the pool
        CLI
        MOV EBX,pFreeLB         ; pLBin^.Next <= pFreeLB;
        MOV [EAX+NextLB],EBX    ;
        MOV pFreeLB,EAX         ; pFreeLB <= pLBin;
        INC _nLBLeft            ;
        STI                     ;

        XOR EAX,EAX             ;ErcOK! (0)
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
;
```

NewTask()

This is the kernel `NewTask()` primitive. This creates a new task and schedules it for execution. This primitive is used primarily to create a task for a job other than the one you are in. The operating system job-management code uses this to create the initial task for a newly loaded job.

For MMURTL version 1.x, stack protection is not provided for transitions through the call gates to the OS code. In future versions, `NewTask()` will allocate operating system stack space separately from the stack memory the caller provides as a parameter. The protection in this version of MMURTL is limited to page protection, and any other access violations caused will result in a general protection violation or page fault. The change will be transparent to applications when it occurs. See Listing 18.19.

Listing 18.19. `NewTask` kernel primitive code.

```
;
;  Procedural interface:
;
;   NewTask(JobNum, CodeSeg, Priority, fDebug, Exch, ESP, EIP): dErcRet
;
;
NTS_Job     EQU [EBP+36]        ;Job Num for this task
NTS_CS      EQU [EBP+32]        ;8 for OS, 18h for user task
NTS_Pri     EQU [EBP+28]        ;Priority of this task
NTS_fDbg    EQU [EBP+24]        ;TRUE for DEBUGing
NTS_Exch    EQU [EBP+20]        ;Exchange for TSS
NTS_ESP     EQU [EBP+16]        ;Initial stack pointer
NTS_EIP     EQU [EBP+12]        ;Task start address
```

```
PUBLIC __NewTask:                    ;
       PUSH EBP                      ;
       MOV EBP,ESP                   ;

       MOV EDX, NTS_Pri              ;
       CMP EDX, nPRI-1               ;Priority OK?
       JBE NT0000
       MOV EAX,ercBadPriority
       JMP NTEnd

NT0000:
       MOV ECX, NTS_Exch
       CMP ECX, nExch                ;Exch in range?
       JBE NT0001
       MOV EAX,ercOutOfRange
       JMP NTEnd

NT0001:
       CLI                           ;we can't be interrupted
       MOV EAX,pFreeTSS              ; NewTSS <= pFreeTSS;
       OR EAX,EAX                    ; IF pFreeTSS=NIL THEN Return;
       JNZ NT0002                    ;
       MOV EAX,ercNoMoreTSSs         ;No...
       JMP NTEnd

NT0002:
       MOV EBX,[EAX+NextTSS]   ; pFreeTSS <= pFreeTSS^.Next
       MOV pFreeTSS,EBX        ;
       DEC _nTSSLeft           ;
       STI

       ;EAX now has pNewTSS

       MOV [EAX+Priority],DL         ;put Priority into TSS
       MOV DWORD PTR [EAX+TSS_EFlags],0202h  ;Load the Flags Register
       MOV [EAX+TSS_Exch], ECX       ;Put new Exch in TSS (ECX is free)
       MOV EBX, NTS_EIP              ;mov EIP into TSS (Start Address)
       MOV [EAX+TSS_EIP],EBX
       MOV EBX, NTS_ESP              ;mov ESP into TSS
       MOV [EAX+TSS_ESP],EBX
       MOV [EAX+TSS_ESP0],EBX        ;
       MOV ECX, NTS_CS               ;mov CS into TSS
       MOV [EAX+TSS_CS],CX

       PUSH EAX                      ;Save pNewTSS

       ;Now we get pJCB from JobNum they passed in so we can
       ;get the PD from the JCB

       MOV EAX, NTS_Job              ;Set up to call GetpJCB
       CALL GetpJCB                  ;EAX now has pJCB
       MOV ECX, EAX                  ;ECX now has pJCB

       POP EAX                       ;Restore pNewTSS to EAX

       MOV [EAX+TSS_pJCB],ECX        ;Put pJCB into TSS
       MOV EBX, [ECX+JcbPD]          ;Set up to call LinToPhy
```

continues

Listing 18.19. continued

```
        PUSH EAX                    ;Save pNewTSS again

        MOV EAX, NTS_Job            ;
        CALL LinToPhy               ;Get Physical Address for PD into EAX
        MOV EBX, EAX
        POP EAX                     ;pNewTSS into EAX
        MOV [EAX+TSS_CR3],EBX       ;Put Physical Add for PD into TSS_CR3
        CMP DWORD PTR NTS_fDbg, 0   ;Debug on entry?
        JE NT0004                   ;No
        MOV WORD PTR [EAX+TSS_TrapBit], 1    ;Yes

NT0004:
        MOV EBX, NTS_Pri            ;Get priority of new task

        CLI                         ;We can't be interrupted
        MOV EDX,pRunTSS                 ;Get who's running
        CMP BYTE PTR [EDX+Priority],BL  ;Who got the highest Pri?
        JA NT0005                       ;New guy does (lowest num)
        CALL enQueueRdy             ;Just put new guy on the ReadyQue (EAX)
        XOR EAX,EAX                 ;ercOk
        JMP NTEnd                   ;Return to caller

NT0005:
        XCHG EAX,EDX                ;CrntTSS -> EAX, New TSS -> EDX
        PUSH EDX                    ;Save New TSS
        CALL enQueueRdy             ;
        POP EAX                     ;New TSS -> EAX
        MOV pRunTSS,EAX             ;Move new TSS into pRunTSS
        MOV BX,[EAX+Tid]            ;Put Selector/Offset in "TSS"
        MOV TSS_Sel,BX              ;
        INC _nSwitches
        MOV EAX, TimerTick      ;Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]         ;Jump to new TSS
        XOR EAX,EAX                 ;ErcOk

NTEnd:
        STI                         ;
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 28                     ;
```

SpawnTask()

The `SpawnTask` is the kernel primitive used to create another task (a thread) in the context of the caller's job control block and memory space. It's the easy way to create additional tasks for existing jobs.

As with `NewTask()`, the newly created task is placed on the ready queue if it is not a higher priority than the task that created it. See Listing 18.20.

Listing 18.20. SpawnTask kernel primitive code.

```
;
; Procedural Interface:
; SpawnTask(pEntry, dPriority, fDebug, pStack, fOSCode);
;
;
pEntryST    EQU DWORD PTR [EBP+28]
dPriST      EQU DWORD PTR [EBP+24]
fDebugST    EQU DWORD PTR [EBP+20]
pStackST    EQU DWORD PTR [EBP+16]
fOSCodeST   EQU DWORD PTR [EBP+12]

NewExchST   EQU DWORD PTR [EBP-4]
NewTSSST    EQU DWORD PTR [EBP-8]

PUBLIC __SpawnTask:                 ;
       PUSH EBP                     ;
       MOV EBP,ESP                  ;
       SUB ESP, 8                   ;two local DWORD vars
       CMP dPriST, nPRI-1           ;Priority OK?
       JBE ST0001
       MOV EAX,ercBadPriority
       JMP STEnd

ST0001:
       LEA EAX, NewExchST           ;Allocate exchange
       PUSH EAX
       CALL FWORD PTR _AllocExch
       OR EAX, EAX                  ;see if we got an error
       JNZ STEnd                    ;Yup, bad news

       ;Allocate a new TSS
       CLI                      ;we can't be interrupted
       MOV EAX,pFreeTSS         ; NewTSS <= pFreeTSS;
       OR EAX,EAX               ; IF pFreeTSS=NIL THEN Return;
       JNZ ST0002               ;
       STI

       ;Dealloc Exch if we didn't get a TSS
       PUSH NewExchST
       CALL FWORD PTR _DeAllocExch
       MOV EAX,ercNoMoreTSSs       ;No...
       JMP NTEnd

ST0002:
       MOV EBX,[EAX+NextTSS]    ; pFreeTSS <= pFreeTSS^.Next
       MOV pFreeTSS,EBX         ;
       DEC _nTSSLeft            ;
       STI

       MOV NewTSSST, EAX            ;Save new TSS
       MOV EBX, NewExchST           ;mov exch into TSS
       MOV [EAX+TSS_Exch],EBX
       MOV WORD PTR [EAX+TSS_CS], OSCodeSel    ;Defaults to OS code selector
       CMP fOSCodeST, 0
       JNE ST0003
       MOV WORD PTR [EAX+TSS_CS], JobCodeSel   ;Make OS code selector
```

continues

Listing 18.20. continued

```
ST0003:
        MOV EBX,pEntryST            ;mov EIP into TSS
        MOV [EAX+TSS_EIP],EBX
        MOV EBX, pStackST           ;mov ESP into TSS
        MOV [EAX+TSS_ESP],EBX
        MOV [EAX+TSS_ESP0],EBX
        MOV EBX, pRunTSS
        MOV EDX, [EBX+TSS_pJCB]     ;Get pJCB from Crnt Task
        MOV [EAX+TSS_pJCB],EDX
        MOV EDX, [EBX+TSS_CR3]      ;Get CR3 from crnt task
        MOV [EAX+TSS_CR3],EDX       ; move into new TSS
        MOV DWORD PTR [EAX+TSS_EFlags],0202h  ;Load the Flags Register
        CMP fDebugST, 0                       ;Debug on entry?
        JE ST0004                             ;No
        MOV WORD PTR [EAX+TSS_TrapBit], 1   ;Yes

ST0004:
        MOV EBX, dPriST             ;mov priority into BL
        MOV [EAX+Priority],BL       ;put in TSS

        CLI                         ;we can't be interrupted
        MOV EDX,pRunTSS             ;Get who's running
        CMP [EDX+Priority],BL       ;Who got the highest Pri?
        JA ST0005                   ;If crnt >, New guy does (lowest num)
        CALL enQueueRdy             ;Old guy does, just put new guy on Q.
        XOR EAX,EAX                 ;ercOk
        JMP STEnd                   ;Return to caller

ST0005:
        XCHG EAX,EDX                ;CrntTSS -> EAX, New TSS -> EDX
        PUSH EDX                    ;New TSS -> Stack
        CALL enQueueRdy             ;Place crnt TSS on Q
        POP EAX                     ;New TSS -> EAX
        MOV pRunTSS,EAX             ;Move new TSS into pRunTSS
        MOV BX,[EAX+Tid]            ;Put Selector/Offset in "TSS"
        MOV TSS_Sel,BX              ;
        INC _nSwitches
        MOV EAX, TimerTick      ;Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        JMP FWORD PTR [TSS]         ;Jump to new TSS
        XOR EAX,EAX                 ;ErcOk

STEnd:
        STI                         ;
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 20                     ;
```

AllocExch()

The `AllocExch()` is an auxiliary function to allocate an exchange. The exchange is a "message port" where you send and receive messages. It is a property of a job (not a task). See Listing 18.21.

Listing 18.21. Allocate exchange code.

```
;
; Procedural Interface :
;
;       AllocExch(pExchRet):dError
;
;          pExchRet is a pointer to where you want the Exchange Handle
;          returned.  The Exchange Handle is a DWORD (4 BYTES).
;

PUBLIC __AllocExch:              ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;

       XOR ESI,ESI              ; Zero the Exch Index
       MOV EBX,prgExch          ; EBX <= ADR rgExch
       MOV ECX,nExch            ; Get number of exchanges in ECX

AE000:
       CLI                                ;
       CMP DWORD PTR [EBX+Owner], 0       ; Is this exchange free to use
       JE AE001                           ; If we found a Free Exch, JUMP
       ADD EBX,sEXCH            ; Point to the next Exchange
       INC ESI                  ; Increment the Exchange Index
       LOOP AE000               ; Keep looping until we are done
       STI                      ;
       MOV EAX,ercNoMoreExch    ; There are no more free exchanges
       MOV ESP,EBP              ;
       POP EBP                  ;
       RETF 4                   ;

AE001:
       MOV EDX,[EBP+0CH]        ; Get the pExchRet in EDX
       MOV [EDX],ESI            ; Put Index of Exch at pExchRet
       MOV EDX,pRunTSS          ; Get pRunTSS in EDX
       MOV EAX,[EDX+TSS_pJCB]   ; Get the pJCB in EAX
       MOV [EBX+Owner],EAX      ; Make the Exch owner the Job
       STI                      ;
       MOV DWORD PTR [EBX+EHead],0     ; Make the msg/TSS queue NIL
       MOV DWORD PTR [EBX+ETail],0     ;
       DEC _nEXCHLeft                  ; Stats
       XOR EAX,EAX              ;ercOK (0)
       MOV ESP,EBP              ;
       POP EBP                  ;
       RETF 4                   ;
```

DeAllocExch()

The `DeAllocExch()` function is the compliment of `AllocExch()`. The `DeAllocExch()` function has a lot more to do, however. When an exchange is returned to the free pool of exchanges, it may have left over link blocks hanging on it. Those link blocks may also contain requests. This means that `DeAllocExch()` must also be a garbage collector.

Messages are dequeued, and link blocks, TSSs, and request blocks are freed up, as necessary. See Listing 18.22.

Listing 18.22. Deallocate exchange code.

```
;
; Procedural Interface :
;
;       DeAllocExch(Exch):ercType
;
;           Exch is the Exchange Handle the process is asking to be released.

PUBLIC __DeAllocExch:                   ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        MOV ESI,[EBP+0CH]       ; Load the Exchange Index in ESI
        MOV EAX,ESI             ; Get the Exchange Index in EAX
        MOV EDX,sEXCH           ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ;
        MOV ECX,EAX             ; Make a copy in ECX (ECX = pExch)

        MOV EDX,pRunTSS         ; Get the pRunTSS in EDX
        MOV EBX,[EDX+TSS_pJCB]  ; Get pJCB in EBX
        MOV EDX,[EAX+Owner]     ; Get the Exchange Owner in EDX
        CMP EBX,EDX             ; If the CurrProc owns the Exchange,
        JE DE000                ; yes
        CMP EBX, OFFSET MonJCB  ; if not owner, is this the OS???
        JE DE000                ; yes
        MOV EAX,ercNotOwner     ;
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 4                  ;

DE000:
        CLI                     ;
        CMP DWORD PTR [ECX+fEMsg],0 ; See if a message may be queued
        JE DE001                    ; No. Go check for Task (TSS)
        MOV ESI, ECX                ; ESI must point to Exch for deQueue
        CALL deQueueMsg         ; Yes, Get the message off of the Exchange
        OR EAX, EAX
        JZ DE002                ; Nothing there. Go free the Exch.

        ;Return the LB to the pool
        MOV EBX,pFreeLB         ; pLBin^.Next <= pFreeLB;
        MOV [EAX+NextLB],EBX    ;
        MOV pFreeLB,EAX         ; pFreeLB <= pLBin;
        INC _nLBLeft            ;
        JMP DE000               ; Go And Check for more.

        ; If we find an RqBlk on the exchange we must respond
        ;with ErcInvalidExch before we continue! This will
        ;only happen if a system service writer doesn't follow
        ;instructions or a service crashes!
        ;
```

```
DE001:
        CMP DWORD PTR [ECX+EHead], 0    ; Check to See if TSS is queued
        JE DE002                ; NIL = Empty, JUMP
        MOV ESI, ECX            ; ESI must point to Exch for deQueue
        CALL deQueueTSS         ; Get the TSS off of the Exchange

        ;Free up the TSS (add it to the free list)
        MOV EBX,pFreeTSS        ; pTSSin^.Next <= pFreeTSS;
        MOV [EAX+NextTSS],EBX           ;
        MOV DWORD PTR [EAX+TSS_pJCB], 0 ; Make TSS invalid
        MOV pFreeTSS,EAX                ; pFreeTSS <= pTSSin;
        INC _nTSSLeft           ;
        JMP DE001               ; Go And Check for more.

DE002:
        MOV DWORD PTR [ECX+Owner], 0     ; Free up the exchange.
        MOV DWORD PTR [ECX+fEMsg], 0     ; Reset msg Flag.
        INC _nEXCHLeft          ; Stats
        STI                     ;
        XOR EAX,EAX             ;ercOK (0)
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 4                  ;
```

GetTSSExch()

The `GetTSSExch()` function is an auxiliary function that returns the exchange of the current TSS to the caller. This is primarily provided for system services that provide direct access blocking calls for customers. These services use the default exchange in the TSS to make a request for the caller of a procedural interface. See Listing 18.23.

Listing 18.23. Get TSS exchange code.

```
;
; Procedural Interface :
;
;       GetTSSExch(pExchRet):dError
;
;          pExchRet is a pointer to where you want the Exchange Handle
;          returned.  The Exchange is a DWORD (4 BYTES).
;
PUBLIC __GetTSSExch:             ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EAX,pRunTSS         ; Get the Ptr To the Running TSS
        MOV ESI,[EBP+0CH]       ; Get the pExchRet in EDX
        MOV EBX, [EAX+TSS_Exch] ; Get Exch in EBX
        MOV [ESI],EBX           ; Put Index of Exch at pExchRet
        XOR EAX, EAX            ; ErcOK
        POP EBP                 ;

        RETF 4                  ;
```

SetPriority()

The auxiliary function `SetPriority()` sets the priority of the caller's task. This doesn't really affect the kernel, because the new priority isn't used until the task is rescheduled for execution. If this is used to set a temporary increase in priority, the complimentary call `GetPriority()` should be used so that the task can be returned to its original priority. See Listing 18.24.

Listing 18.24. Set priority code.

```
;
; SetPriority - This sets the priority of the task that called it
; to the priority specified in the single parameter.
;
; Procedural Interface :
;
;       SetPriority(bPriority):dError
;
;          bPriority is a byte with the new priority.
;
PUBLIC __SetPriority            ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;
       MOV EAX,pRunTSS          ; Get the Ptr To the Running TSS
       MOV EBX,[EBP+0CH]        ; Get the new pri into EBX
       AND EBX, 01Fh            ; Nothing higher than 31!
       MOV BYTE PTR [EAX+Priority], BL ;Put it in the TSS
       XOR EAX, EAX             ; ErcOK - No error.
       POP EBP                  ;
       RETF 4                   ;
```

GetPriority()

The `GetPriority()` function is an auxiliary function that returns the current priority of the running task. Callers should use `GetPriority()` before using the `SetPriority()` call so that they can return to their original priority when their "emergency" is over. See Listing 18.25.

Listing 18.25. Get priority code.

```
;
; GetPriority - This gets the priority of the task that called it
; and passes it to bPriorityRet.
;
; Procedural Interface :
;
;       GetPriority(bPriorityRet):dError
;
;          bPrioritYret is a pointer to a byte where you want the
;             priority returned.
;
```

```
PUBLIC __GetPriority            ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;
       MOV EAX,pRunTSS          ; Get the Ptr To the Running TSS
       MOV EBX,[EBP+0CH]        ; Get the return pointer into EBX
       MOV DL, BYTE PTR [EAX+Priority]
       MOV BYTE PTR [EBX], DL   ;
       XOR EAX, EAX             ; ErcOK - No error.
       POP EBP                  ;
       RETF 4                   ;
```

19

Memory Management Code

The memory management code is contained in the file MemCode.ASM. It begins with data definitions that are used by all the private (local) and public functions.

Memory Management Data

Three important structures you modify and deal with are the page allocation map (PAM), page directories (PDs), and page tables (PTs). The variable naming conventions should help you follow the code. The task state segments are also used extensively. Listing 19.1 is the data segment portion of the memory management code.

Listing 19.1. Memory management constants and data.

```
.DATA

.INCLUDE MOSEDF.INC
.INCLUDE TSS.INC
.INCLUDE JOB.INC

PUBLIC _nPagesFree      DD 0            ;Number of free physical pages left
PUBLIC _oMemMax         DD 000FFFFFh    ;Default to 1 MB
                                        ;Page Allocation Map
PUBLIC rgPAM    DB 2048 DUP (0)         ;1 bit/4Kbytes - 2048 bytes = 64Mb
PUBLIC sPAM     DD 32                   ;Deafult is 32 (1Mb)
sPAMmax         EQU 2048                ;Max is 2048 (64Mb)

MemExch         DD 00000000h            ;Semaphore exchange for Mem Mgmt
pNextPT         DD 00000000h            ;We alloc the "next" page table
                                        ;in advance (MANDATORY)
```

It was easier for me to define masks in assembly language. The following definitions are used to set memory pages for specific access rights:

```
;               20 Bit Address      AVL00DA00UWP
MEMALIAS EQU  00000000000000000000100000000111b ;User Writable
MEMUSERD EQU  00000000000000000000000000000111b ;User Writable (Data)
MEMUSERC EQU  00000000000000000000000000000101b ;User Readable (Code)
MEMSYS   EQU  00000000000000000000000000000101b ;System Read/Write
ALIASBIT EQU  00000000000000000000100000000000b
PRSNTBIT EQU  00000000000000000000000000000001b
;
```

Internal Code

The code is divided into two basic sections. The first section comprises all of the internal routines used by the public functions users access. I call these *internal support code*. I will describe the purpose of each of these functions just before I show you the code. As with all assembler files for DASM, the code section must start with the `.CODE` command.

InitMemMgmt

The `InitMemMgmt` function finds out how much memory, in megabytes, you have, by writing to the highest dword in each megabyte until a read-back test fails. It sets `nPagesFree` after finding out just how much you have. You assume 1MB to start, which means you start at the top of 2MB (1FFFFCh). It places the highest addressable offset in *global* `oMemMax`. You also calculate the number of pages of physical memory this is, and store that number in the global variable `nPagesFree`. See Listing 19.2.

Listing 19.2. Memory management initialization code.

```
.CODE
;IN:   Nothing
;OUT:  Nothing (except that you can use memory management routines now!)
;USED: ALL REGISTERS ARE USED.
;
PUBLIC InitMemMgmt:
        MOV _nPagesFree, 256    ;1 Mb of pages = 256
        MOV EAX,1FFFFCh         ;top of 2 megs (for DWORD)
        XOR EBX,EBX             ;
        MOV ECX,06D72746CH      ;'mrtl' test string value for memory
MEMLoop:
        MOV DWORD PTR [EAX],0h  ;Set it to zero intially
        MOV DWORD PTR [EAX],ECX ;Move in test string
        MOV EBX,DWORD PTR [EAX] ;Read test string into EBX
        CMP EBX,ECX             ;See if we got it back OK
        JNE MemLoopEnd          ;NO!
        ADD EAX,3               ;Yes, oMemMax must be last byte
        MOV _oMemMax,EAX        ;Set oMemMax
        SUB EAX,3               ;Make it the last DWord again
        ADD EAX,100000h         ;Next Meg
        ADD _nPagesFree, 256    ;Another megs worth of pages
        ADD sPAM, 32            ;Increase PAM by another meg
        CMP EAX,3FFFFFCh        ;Are we above 64 megs
        JAE MemLoopEnd          ;Yes!
        XOR EBX,EBX             ;Zero out for next meg test
        JMP MemLoop
MemLoopEnd:
```

The page allocation map is now sized and zeroed. Now you must fill in bits used by OS, which was just loaded, and by the video RAM and boot ROM, neither of which you consider free. The code in Listing 19.3 also fills out each of the page table entries (PTEs) for the initial OS code and data. Note that linear addresses match physical addresses for the initial OS data and code. It's the law!

Listing 19.3. Continuation of memory management initialization.

```
; This first part MARKS the OS code and data pages as used
; and makes PTEs.
;
```

continues

Listing 19.3. continued

```
        MOV EDX, OFFSET pTbl1       ;EDX points to OS Page Table 1
        XOR EAX, EAX                ;Point to 1st physical/linear page (0)
IMM001:
        MOV [EDX], EAX              ;Make Page Table Entry
        AND DWORD PTR [EDX], 0FFFFF000h     ;Leave upper 20 Bits
        OR  DWORD PTR [EDX], 0001h          ;Supervisor, Present
        MOV EBX, EAX
        CALL MarkPage               ;Marks page in PAM
        ADD EDX, 4                  ;Next table entry
        ADD EAX, 4096
        CMP EAX, 30000h             ;Reserve 192K for OS (for now)
        JAE SHORT IMM002
        JMP SHORT IMM001            ;Go for more
```

Now you fill in PAM and PTEs for video and ROM slots. This covers A0000 through 0FFFFFh, which is the upper 384K of the first megabyte. Right now you just mark everything from A0000 to FFFFF as used. The routine in Listing 19.4 could be expanded to search through the ROM pages of ISA memory, C0000h–FFFFFh, finding the inaccessible ones and marking them as allocated in the PAM. Several chip sets on the market, such as the 82C30 C&T, enable you to set ROM areas as usable RAM, but I can't be sure everyone can do it, nor can I provide instructions to everyone.

Listing 19.4. Continuation of memory management initialization.

```
IMM002:
        MOV EAX, 0A0000h            ;Points to 128K Video & 256K ROM area
        MOV EBX, EAX                ;
        SHR EBX, 10                 ;Make it index (SHR 12, SHL 2)
        MOV EDX, OFFSET pTbl1       ;EDX pts to Page Table
        ADD EDX, EBX
IMM003:
        MOV [EDX], EAX                  ;Make Page Table Entry
        AND DWORD PTR [EDX], 0FFFFF000h ;Leave upper 20 Bits
        OR  DWORD PTR [EDX], 0101b      ;Mark it "User" "ReadOnly" & "Present"
        MOV EBX, EAX                ;Setup for MarkPage call
        CALL MarkPage               ;Mark it used in the PAM
        ADD EDX, 4                  ;Next PTE entry
        ADD EAX, 4096               ;Next page please
        CMP EAX, 100000h            ;1Mb yet?
        JAE IMM004                  ;Yes
        JMP SHORT IMM003            ;No, go back for more
;
```

The initial page directory and the page table are static. Now you can go into *paged* memory mode. This is done by loading CR3 with the physical address of the page directory, then reading CR0, logically ANDing it with 8000000h, and then writing it again. After the MOV CR0 you must issue a JMP instruction to clear the prefetch queue of any bogus physical addresses. See Listing 19.5.

Listing 19.5. Turning on paged memory management.

```
IMM004:
        MOV EAX, OFFSET PDir1  ;Physical address of OS page directory
        MOV CR3, EAX           ;Store in Control Reg 3
        MOV EAX, CR0           ;Get Control Reg 0
        OR  EAX, 80000000h     ;Set paging bit ON
        MOV CR0, EAX           ;Store Control Reg 0
        JMP IM0005             ;Clear prefetch queue
IM0005:
;
```

Now you allocate an exchange that the OS uses as a semaphore to prevent reentrant use of any of the critical memory management functions. See Listing 19.6.

Listing 19.6. Allocation of memory management exchange.

```
;
        LEA EAX, MemExch        ;Alloc Semaphore Exch for Memory calls
        PUSH EAX
        CALL FWORD PTR _AllocExch

        PUSH MemExch                ;Send a dummy message to pick up
        PUSH 0FFFFFFF1h
        PUSH 0FFFFFFF1h
        CALL FWORD PTR _SendMsg
```

You must allocate a page table to be used when one must be added to a user PD or OS PD. This must be done in advance of finding out you need one, because you may not have a linear address to access it if the current PTs are all used up! It's a little complicated, I'm afraid. See Listing 19.7.

Listing 19.7. Finishing memory management initialization.

```
        PUSH 1                      ; 1 page for Next Page Table
        MOV EAX, OFFSET pNextPT     ;
        PUSH EAX
        CALL FWORD PTR _AllocOSPage ; Get 'em!

        RETN                ;Done initializing memory managment
;
```

FindHiPage

The `FindHiPage` function finds the first unused physical page in memory from the *top* down and returns the physical address of it to the caller. It also marks the page as used, assuming that you will allocate it. Of course, this means that if you call `FindHiPage` and don't use the page, you must call `UnMarkPage` to release the page. `FindHiPage` reduces `nPagesFree` by one. See Listing 19.8.

Listing 19.8. Find highest physical page code.

```
;
;IN  :  Nothing
;OUT :  EBX is the physical address of the new page, or 0 if error
;USED:  EBX, Flags

        PUSH EAX
        PUSH ECX
        PUSH EDX
        MOV ECX, OFFSET rgPAM       ;Page Allocation Map
        MOV EAX, sPAM               ;Where we are in PAM
        DEC EAX                     ;EAX+ECX will be offset into PAM
FHP1:
        CMP BYTE PTR [ECX+EAX],0FFh     ;All 8 pages used?
        JNE FHP2                    ;No
        CMP EAX, 0                  ;Are we at Bottom of PAM?
        JE  FHPn                    ;no memory left...
        DEC EAX                     ;Another Byte lower
        JMP SHORT FHP1              ;Back for next byte
FHP2:
        MOV EBX, 7                  ;
        XOR EDX, EDX
        MOV DL, BYTE PTR [ECX+EAX]  ;Get the byte with a whole in it...
FHP3:
        BT  EDX, EBX                ;Test bits
        JNC FHPf                    ;FOUND ONE! (goto found)
        CMP EBX, 0                  ;At the bottom of the Byte?
        JE  FHPn                    ;Error (BAD CPU???)
        DEC EBX                     ;Next bit
        JMP FHP3
FHPf:
        BTS EDX, EBX                ;Set the bit indexed by EBX
        MOV BYTE PTR [ECX+EAX], DL  ;Set page in use
        SHL EAX, 3                  ;Multiply time 8 (page number base in byte)
        ADD EBX, EAX                ;Add page number in byte
        SHL EBX, 12                 ;Now EBX = Physical Page Addr (EBX*4096)
        DEC _nPagesFree             ;One less available
        POP EDX
        POP ECX
        POP EAX
        RETN
FHPn:
        XOR EBX, EBX                ;Set to zero for error
        POP EDX
        POP ECX
        POP EAX
        RETN
```

FindLoPage

The `FindLoPage` function finds the first unused physical page in memory from the *bottom* up and returns its physical address to the caller. It also marks the page as used, assuming that you will allocate the page. Once again, if you call `FindLoPage` and don't use the page, you must call `UnMarkPage` to release the page. `FindLoPage` reduces `nPagesFree` by one. See Listing 19.9.

Listing 19.9. Find lowest physical page code.

```
;
;IN  :  Nothing
;OUT :  EBX is the physical address of the new page, or 0 if error
;USED:  EBX, Flags

        PUSH EAX
        PUSH ECX
        PUSH EDX
        MOV ECX, OFFSET rgPAM       ;Page Allocation Map
        XOR EAX, EAX                ;Start at first byte in PAM
FLP1:
        CMP BYTE PTR [ECX+EAX],0FFh     ;All 8 pages used?
        JNE FLP2                    ;No
        INC EAX                     ;Another Byte lower
        CMP EAX, sPAM               ;Are we past at TOP of PAM?
        JAE FLPn                    ;no memory left...
        JMP SHORT FLP1              ;Back for next byte
FLP2:
        XOR EBX, EBX                ;
        XOR EDX, EDX
        MOV DL, BYTE PTR [ECX+EAX]  ;Get the byte with a whole in it...
FLP3:
        BT  EDX, EBX                ;Test bits
        JNC FLPf                    ;FOUND ONE! (goto found)
        INC EBX                     ;Next bit
        CMP EBX, 8                  ;End of the Byte?
        JAE FLPn                    ;Error (BAD CPU???)
        JMP FLP3
FLPf:
        BTS EDX, EBX                ;Set the bit indexed by EBX
        MOV BYTE PTR [ECX+EAX], DL  ;Set page in use

        SHL EAX, 3                  ;Multiply time 8 (page number base in byte)
        ADD EBX, EAX                ;Add page number in byte
        SHL EBX, 12                 ;Now EBX = Physical Page Addr (EBX*4096)
        DEC _nPagesFree             ;One less available
        POP EDX
        POP ECX
        POP EAX
        RETN
FLPn:
        XOR EBX, EBX                ;Set to zero for error
        POP EDX
        POP ECX
        POP EAX
        RETN
```

MarkPage

Given a physical memory address, `MarkPage` finds the bit in the PAM associated with it and sets it to show the physical page in use. This function is used with the routines that initialize all memory management function. This reduces `nPagesFree` by one. See Listing 19.10.

Listing 19.10. Code to mark a physical page in use.

```
;
;IN  :  EBX is the physical address of the page to mark
;OUT :  Nothing
;USED:  EBX, Flags

        PUSH EAX
        PUSH ECX
        PUSH EDX
        MOV EAX, OFFSET rgPAM       ;Page Allocation Map
        AND EBX, 0FFFFF000h         ;Round down to page modulo 4096
        MOV ECX, EBX
        SHR ECX, 15                 ;ECX is now byte offset into PAM
        SHR EBX, 12                 ;Get Bit offset into PAM
        AND EBX, 07h                ;EBX is now bit offset into byte of PAM
        MOV DL, [EAX+ECX]           ;Get the byte into DL
        BTS EDX, EBX                ;BitSet nstruction with Bit Offset
MOV [EAX+ECX], DL           ;Save the new PAM byte
        DEC _nPagesFree             ;One less available
        POP EDX
        POP ECX
        POP EAX
        RETN
```

UnMarkPage

Given a physical memory address, `UnMarkPage` finds the bit in the PAM associated with it and *resets* it to show the physical page available again. This increases `nPagesFree` by one. See Listing 19.11.

Listing 19.11. Code to free a physical page for reuse.

```
;
;IN  :  EBX is the physical address of the page to UNmark
;OUT :  Nothing
;USED:  EBX, Flags

        PUSH EAX
        PUSH ECX
        PUSH EDX
        MOV EAX, OFFSET rgPAM       ;Page Allocation Map
        AND EBX, 0FFFFF000h         ;Round down to page modulo
        MOV ECX, EBX
        SHR ECX, 15                 ;ECX is now byte offset into PAM
        SHR EBX, 12                 ;
        AND EBX, 07h                ;EBX is now bit offset into byte of PAM
        ADD EAX, ECX
        MOV DL, [EAX]
        BTR EDX, EBX                ;BitReset instruction
        MOV [EAX], DL
        INC _nPagesFree             ;One more available
        POP EDX
        POP ECX
        POP EAX
        RETN
```

LinToPhy

The LinToPhy function looks up the physical address of a 32-bit linear address passed in. The JCB is used to identify whose page tables you are translating. The linear address is used to look up the page table entry that is used to get the physical address. This call is used for things like aliasing for messages, DMA operations, and so on. This also leaves the linear address of the PTE itself in ESI for callers that need it. This function supports the similarly named public routine. See Listing 19.12.

Listing 19.12. Code to convert linear to physical addresses.

```
;
; INPUT:    EAX — Job Number that owns memory we are aliasing
;           EBX — Linear address
;
; OUTPUT:   EAX — Physical Address
;           ESI — Linear Address of PTE for this linear address
;
; USED:     EAX, EBX, ESI, EFlags
;
PUBLIC LinToPhy:
        PUSH EBX                ;Save Linear
        CALL GetpJCB            ;Leaves pJCB in EAX
        MOV EAX, [EAX+JcbPD]    ;EAX now has ptr to PD!
        ADD EAX, 2048           ;Move to shadow addresses in PD
        SHR EBX, 22             ;Shift out lower 22 bits leaving 10 bit offset
        SHL EBX, 2              ;*4 to make it a byte offset into PD shadow
        ADD EBX, EAX            ;EBX/EAX now points to shadow
        MOV EAX, [EBX]          ;EAX now has Linear of Page Table
        POP EBX                 ;Get original linear back in EBX
        PUSH EBX                ;Save it again
        AND EBX, 003FFFFFh      ;Get rid of upper 10 bits
        SHR EBX, 12             ;get rid of lower 12 to make it an index
        SHL EBX, 2              ;*4 makes it byte offset in PT
        ADD EBX, EAX            ;EBX now points to Page Table entry!
        MOV ESI, EBX            ;Save this address for caller
        MOV EAX, [EBX]          ;Physical base of page is in EAX
        AND EAX, 0FFFFF000h     ;mask off lower 12
        POP EBX                 ;Get original linear
        AND EBX, 00000FFFh      ;Cut off upper 22 bits of linear
        OR EAX, EBX             ;EAX now has REAL PHYSICAL ADDRESS!
        RETN
```

FindRun

The FindRun function locates a contiguous run of free linear memory in one of the user or operating-system page tables. This is at either address base 0 for the operating system or the 1GB address mark for the user. The EAX register will be set to 0 if you are looking for OS memory; the caller sets it to 256 if you are looking for user memory. The linear address of the run is returned in EAX, unless no run that large exists, in which case you return 0. The linear

run may span page tables if additional tables already exist. This is an interesting routine because it uses two nested loops to walk through the page directory and page tables while using the SIB (scale index base) addressing of the Intel processor for indexing. See Listing 19.13.

Listing 19.13. Code to find a free run of linear memory.

```
;
; IN :  EAX  PD Shadow Base Offset for memory (0 for OS, 256 for user)
;       EBX  Number of Pages for run
;
; OUT:  EAX  Linear address or 0 if no run is large enough
;       EBX  still has count of pages
; USED: EAX, EBX, EFlags  (all other registers saved & restored)
;
;
FindRun:
        PUSH EBX                ;Holds count of pages (saved for caller)
        PUSH ECX                ;Keeps count of how many free found so far
        PUSH EDX                ;Index into PD for PT we are working on
        PUSH ESI                ;Address of PD saved here
        PUSH EDI                ;

        MOV ECX, EBX            ;Copy number of pages to ECX. Save in EBX
        MOV EDX, EAX            ;Index into shadow addresses from EAX

        CALL GetpCrntJCB        ;Leaves pCrntJCB in EAX
        MOV ESI, [EAX+JcbPD]    ;ESI now has ptr to PD
        ADD ESI, 2048           ;Move to shadow addresses

FR0:
        MOV EDI, [ESI+EDX*4]    ;Linear address of next page table into EDI
        OR EDI, EDI             ;Is the address NON-ZERO (valid)?
        JNZ FR1                 ;Yes, go for it
        XOR EAX, EAX            ;Return 0 cause we didn't find it!
        JMP SHORT FREXIT        ;
FR1:
        XOR EAX, EAX            ;EAX indexes into PT (to compare PTEs)
FR2:
        CMP EAX, 1024           ;Are we past last PTE of this PT?
        JB  FR3                 ;No, keep testing
        INC EDX                 ;Next PT!
        JMP SHORT FR0           ;
FR3:
        CMP DWORD PTR [EDI+EAX*4], 0    ;Zero means it's empty (available)
        JNE FR4                         ;In use
        DEC ECX                 ;One more empty one!
        JZ  FROK                ;We found enough entries goto OK
        INC EAX                 ;Not done yet, Next PTE Please.
        JMP SHORT FR2           ;
FR4:
        ;If we got here we must reset ECX for full count and
        ;go back and start looking again
        INC EAX                 ;Not empty, next PTE please
        MOV ECX, EBX            ;We kept original count in EBX
        JMP FR2
FROK:
        ;If we got here it means that ECX has made it to zero and
        ;we have a linear run large enough to satisy the request.
```

```
        ;The starting linear address is equal to number of the last
        ;PTE we found minus ((npages -1) * 4096)
        ;EDX was index into PD, while EAX was index into PT.
        ;EBX still has count of pages.

        SHL EDX, 22         ;EDX is 10 MSBs of Linear Address
        SHL EAX, 12         ;EAX is next 10 bits of LA
        OR EAX, EDX         ;This is the linear address we ended at
        DEC EBX             ;One less page (0 offset)
        SHL EBX, 12         ;Times size of page (* 4096)
        SUB EAX, EBX        ;From current linear address in tables

FREXIT:
        POP EDI             ;
        POP ESI             ;
        POP EDX             ;
        POP ECX             ;
        POP EBX             ;
        RETN
```

AddRun

The `AddRun` function adds one or more page table entries (PTEs) to a page table, or tables, if the run spans two or more tables. The address determines the protection level of the PTEs you add. If the address is less than 1GB, it means operating-system memory space that you will set to *system*. An address greater than 1GB is set to *user* level protection. The linear address of the run should be in EAX, and the count of pages should be in EBX. This is the way `FindRun` left them. Many functions that pass data in registers are designed to complement each other's register usage for speed. See Listing 19.14.

Listing 19.14. Adding a run of linear memory.

```
;
; IN :  EAX  Linear address of first page
;       EBX  Number of Pages to add
; OUT:  Nothing
; USED: EAX, EFlags
;
AddRun:
        PUSH EBX                ;(save for caller)
        PUSH ECX                ;
        PUSH EDX                ;
        PUSH ESI                ;
        PUSH EDI                ;

        MOV ECX, EBX            ;Copy number of pages to ECX (EBX free to use).
        MOV EDX, EAX            ;LinAdd to EDX
        SHR EDX, 22             ;Get index into PD for first PT
        SHL EDX, 2              ;Make it index to DWORDS

        PUSH EAX                ;Save EAX thru GetpCrntJCB call
```

continues

Listing 19.14. continued

```
        CALL GetpCrntJCB        ;Leaves pCrntJCB in EAX
        MOV ESI, [EAX+JcbPD]    ;ESI now has ptr to PD!
        POP EAX                 ;Restore linear address

        ADD ESI, 2048           ;Offset to shadow address of PD
        ADD ESI, EDX            ;ESI now points to initial PT (EDX now free)

        MOV EDX, EAX            ;LinAdd into EDX again
        AND EDX, 003FF000h      ;get rid of upper 10 bits & lower 12
        SHR EDX, 10             ;Index into PD for PT (10 vice 12 -> DWORDS)
AR0:
        MOV EDI, [ESI]          ;Linear address of next page table into EDI

        ;At this point, EDI is pointing the next PT.
        ;SO EDI+EDX will point to the next PTE to do.
        ;Now we must call FindPage to get a physical address into EBX,
        ;then check the original linear address to see if SYSTEM or USER
        ;and OR in the appropriate control bits, THEN store it in PT.

AR1:
        CALL FindHiPage         ;EBX has Phys Pg (only EBX affected)
        OR EBX, MEMSYS          ;Set PTE to present, User ReadOnly
        CMP EAX, 40000000h      ;See if it's a user page
        JB  AR2
        OR EBX, MEMUSERD        ;Sets User/Writable bits of PTE

AR2:
        MOV DWORD PTR [EDI+EDX], EBX    ;EDX is index to exact entry
        DEC ECX                         ;Are we done??
        JZ ARDone
        ADD EDX, 4              ;Next PTE please.
        CMP EDX, 4096           ;Are we past last PTE of this PT?
        JB AR1                  ;No, go do next PTE
        ADD ESI, 4              ;Yes, next PDE (to get next PT)
        XOR EDX,EDX             ;Start at the entry 0 of next PT
        JMP SHORT AR0           ;
ARDone:
        POP EDI             ;
        POP ESI             ;
        POP EDX             ;
        POP ECX             ;
        POP EBX             ;
        RETN
```

AddAliasRun

The `AddAliasRun` function adds one or more PTEs to a page table—or tables, if the run spans two or more tables—adding PTEs from another job's PTs, marking them as alias entries. Aliased runs are always at *user* protection levels even if they are in the operating-system address span.

The new linear address of the run should be in EAX, and the count of pages should be in EBX. Once again, this is the way `FindRun` left them. The ESI register has the linear address you are aliasing, and the EDX register has the job number. See Listing 19.15.

Listing 19.15. Adding an aliased run of linear memory.

```
;
; IN :  EAX  Linear address of first page of new alias entries
;            (from find run)
;       EBX  Number of Pages to alias
;       ESI  Linear Address of pages to Alias (from other job)
;       EDX  Job Number of Job we are aliasing
;
; OUT:  Nothing
; USED: EAX, EFlags
;
AliasLin    EQU DWORD PTR [EBP-4]
AliasJob    EQU DWORD PTR [EBP-8]

AddAliasRun:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        SUB ESP, 8

        MOV AliasLin, ESI
        MOV AliasJob, EDX

        PUSH EBX                ;(save for caller)
        PUSH ECX                ;
        PUSH EDX                ;
        PUSH ESI                ;
        PUSH EDI                ;

        ;This first section sets to make [ESI] point to first PT that
        ;we have to move the other guy's physical pages into

        MOV ECX, EBX            ;Copy number of pages to ECX (EBX free to use).
        MOV EDX, EAX            ;LinAdd to EDX
        SHR EDX, 22             ;Get index into PD for first PT
        SHL EDX, 2              ;Make it index to DWORDS

        PUSH EAX                ;Save EAX thru GetpCrntJCB call
        CALL GetpCrntJCB        ;Leaves pCrntJCB in EAX
        MOV ESI, [EAX+JcbPD]    ;ESI now has linear address of PD
        POP EAX                 ;Restore linear address

        ADD ESI, 2048           ;Offset to shadow addresses in PD
        ADD ESI, EDX            ;ESI now points to first PT of interest

        MOV EDX, EAX            ;LinAdd into EDX again
        AND EDX, 003FF000h      ;get rid of upper 10 bits & lower 12
        SHR EDX, 10             ;Index into PD for PT (10 vice 12 -> DWORDS)
ALR0:
        MOV EDI, [ESI]          ;Linear address of crnt page table into EDI

        ;At this point, EDI is pointing to the PT we are in.
        ;SO then EDI+EDX will point to the next PTE to do.
        ;Now we must call LinToPhy with Linear Add & JobNum
        ; to get a physical address into EAX.
        ;This is the Physical address to store in the new PTE.  We must
        ;mark it MEMALIAS before adding it to PT.
ALR1:
        PUSH ESI                ;Save for next loop (used by LinToPhy)
```

continues

Listing 19.15. continued

```
        MOV EAX, AliasJob       ;Job we are aliasing
        MOV EBX, AliasLin       ;Address we are aliasing
        ADD AliasLin, 4096      ;Set up for next loop (post`increment)
        CALL LinToPhy           ;

        ;EAX now has physical address for this page
        ;

        AND EAX, 0FFFFF000h     ;cut off system bits of PTE
        OR EAX, MEMALIAS        ;Set system bits as ALIAS

        POP ESI                 ;Restore ESI (LinToPhy used it)

        ;Now store it in new PTE

        MOV DWORD PTR [EDI+EDX], EAX    ;EDX is index to exact entry

        DEC ECX                 ;Are we done??
        JZ ALRDone
        ADD EDX, 4              ;Next PTE please.
        CMP EDX, 4096           ;Are we past last PTE of this PT?
        JB ALR1                 ;No, go do next PTE
        ADD ESI, 4              ;Yes, next PDE (to get next PT)
        XOR EDX,EDX             ;Start at the entry 0 of next PT
        JMP SHORT ALR0          ;
ALRDone:
        POP EDI             ;
        POP ESI             ;
        POP EDX             ;
        POP ECX             ;
        POP EBX             ;

        MOV ESP,EBP             ;
        POP EBP                 ;
        RETN
```

AddUserPT

The `AddUserPT` function creates a new user page table, initializes it, and sticks it in the user's page directory. This will be in user address space (greater than 1GB).

This is easier than `AddOSPT`, as shown in the following, because there is no need to update anyone else's PDs. This sets the protection on the PT to *user*-read-and-write. Individual PTEs will be set read-only for code. See Listing 19.16.

Listing 19.16. Adding a page table for user memory.

```
;
; IN :  Nothing
; OUT:  0 if OK or Error (ErcNoMem - no free phy pages!)
; USED: EAX, EFlags
;
AddUserPT:
```

```
        PUSH EBX                ;(save for caller)
        PUSH ECX                ;
        PUSH EDX                ;
        PUSH ESI                ;
        PUSH EDI                ;

        MOV EAX, _nPagesFree    ;See if have enuf physical memory
        OR EAX, EAX
        JNZ AUPT01
        MOV EAX, ErcNoMem       ;Sorry, out of physical mem
        JMP AUPTDone
AUPT01:
        CALL GetCrntJobNum      ;Leaves job num in EAX (for LinToPhy)
        MOV EBX, pNextPT        ;Pre allocated Page (Linear Address)
        CALL LinToPhy           ;EAX will have Physical address

        ; Put it in the User PD (and linear in shadow).
        ; Find first empty slot

        CALL GetpCrntJCB        ;pJCB in EAX
        MOV EDI, JcbPD          ;Offset to PcbPD in JCB
        ADD EDI, EAX            ;EDI points to UserPD Address
        MOV ESI, [EDI]          ;ESI now points to PD
        ADD ESI, 2048           ;ESI now points to upper 1 GB in PD
        MOV ECX, 511            ;Number of entries (at least 1 is already gone)
AUPT02:
        ADD ESI, 4              ; Next possible empty entry
        MOV EBX, [ESI]
        OR EBX, EBX             ; Is it empty?
        LOOPNZ AUPT02           ; No! (Try again)

        ; ESI now points to empty Slot
        ; Physical Address of new table is still in EAX
        ; Get Linear address back into EBX
        ; and put them into PD

        OR  EAX, MEMUSERD       ;Set user bits (Read/Write)
        MOV [ESI], EAX          ;Physical address in lower half
        ADD ESI, 2048           ;Move to shadow
        MOV EBX, pNextPT        ;Linear add back into EBX
        MOV [ESI], EBX          ;Put in Linear address of PT (upper half)

        ;Now we now need another PreAllocated Page Table for
        ;next time. Get a run of 1 for next new page table

        MOV EBX, 1              ;size of request
        XOR EAX, EAX            ;PD shadow offset needed by FindRun (0)
        CALL FindRun
        OR EAX, EAX             ;was there an error (0 means no mem)
        JNZ AUPT05
        MOV EAX, ErcNoMem       ;
        JMP SHORT AUPTDone
AUPT05:
        MOV pNextPT, EAX        ;save pNextPT (the linear address)
        CALL AddRun             ;AddRun will return NON-ZERO on error
AUPTDone:
        POP EDI             ;
        POP ESI             ;
```

continues

Listing 19.16. continued

```
        POP EDX             ;
        POP ECX             ;
        POP EBX             ;
        RETN
```

AddOSPT

The `AddOSPT` function is more complicated than `AddUserPT` because a reference to each new operating-system page table must be placed in all user page directories. You must do this to ensure the operating-system code can reach its memory no matter what job or task it is running in. Adding an operating-system page table doesn't happen often; in many cases, it won't happen except once or twice while the operating system is running. See Listing 19.17.

Listing 19.17. Adding an operating system page table.

```
;
; IN :  Nothing
; OUT:  0 if OK or Error (ErcNoMem - no free phy pages!)
; USED: EAX, EFlags
;
AddOSPT:
        PUSH EBX                ;(save for caller)
        PUSH ECX                ;
        PUSH EDX                ;
        PUSH ESI                ;
        PUSH EDI                ;

        MOV EAX, _nPagesFree    ;See if have enuf physical memory
        OR EAX, EAX
        JNZ AOPT01
        MOV EAX, ErcNoMem       ;Sorry, out of physical mem
        JMP AOPTDone
AOPT01:
        MOV EAX, 1              ;OS Job Number (Monitor)
        MOV EBX, pNextPT        ;Pre allocated Page (Linear Address)
        CALL LinToPhy           ;EAX will have Physical address

        ; Put it in the OS PD (and linear in shadow).
        ; Find first empty slot

        MOV ESI, OFFSET PDir1   ; ESI points to OS Pdir
        MOV ECX, 511            ; Count of PDEs to check
AOPT02:
        ADD ESI, 4              ; Next possible empty entry
        MOV EBX, [ESI]
        OR EBX, EBX             ; Is it empty?
        LOOPNZ AOPT02           ; No! (Try again)

        ; ESI now points to empty PDir Slot
        ; EAX still has Physical Address of new table
        ; Get Physical Address back into EBX
        ; and put them into PDir
```

```
        OR EAX, PRSNTBIT        ;Set present bit
        MOV [ESI], EAX          ;Physical address in lower half
        ADD ESI, 2048           ;Move to shadow
        MOV EBX, pNextPT        ;Linear add back into EBX
        MOV [ESI], EBX          ;Put in Linear address of PT (upper half)

        ; Update ALL PDs from PDir1 !!
        ; This doesn't happen often if it happens at all.
        ; The OS will usually not take 4 MBs even with ALL
        ; of its dynamic structures (except on
        ; a 32 Mb system or larger and when fully loaded)

        MOV EDX, nJCBs          ; # of dynamic JCBs
AOPT03:
        MOV EAX, EDX            ;Next JCB
        CALL GetpJCB            ;EAX now has pointer to a job's PD
        MOV ECX, [EAX+JcbPD]    ;See if PD id zero (Inactive JCB)
        OR ECX, ECX             ;Is it a valid Job? (0 if not)
        JZ AOPT04               ;No, Not a valid JCB (unused)

        ADD EAX, JcbPD          ;EAX NOW points to PD of JCB
        MOV EBX, OFFSET PDir1   ;Source of Copy

        PUSH EDX                ;Save nJCB we are on

        PUSH EAX                ;Save values on stack
        PUSH EBX

        PUSH EBX                ;Source
        PUSH EAX                ;Destination
        PUSH 1024               ;Lower half of PD (Physical Adds)
        CALL FWORD PTR _CopyData

        POP  EBX                ;Get values from stack
        POP  EAX

        ADD EBX, 2048           ;Move to shadow
        PUSH EBX
        ADD EAX, 2048           ;Move to shadow
        PUSH EAX
        PUSH 1024               ;Upper half of PD (Linear Adds)
        CALL FWORD PTR _CopyData

        POP EDX                 ; Get back JCB number

AOPT04:
        DEC EDX
        CMP EDX, 2
        JA AOPT03               ;Jobs 1 & 2 use PDir1 (Mon & Debugger)

        ;At this point the new table is valid to ALL jobs!
        ;We now need another PreAllocated Page Table for
        ;next time. Get a run of 1 for next new page table

        MOV EBX, 1              ;size of request
        XOR EAX, EAX            ;PD shadow offset needed by FindRun (0)
```

continues

Listing 19.17. continued

```
        CALL FindRun
        OR EAX, EAX             ;was there an error (0 means no mem)
        JNZ AOPT05
        MOV EAX, ErcNoMem       ;
        JMP SHORT AOPTDone
AOPT05:
        MOV pNextPT, EAX        ;save pNextPT (the linear address)
        CALL AddRun             ;AddRun
        XOR EAX, EAX            ;Set ErcOK (0)
AOPTDone:
        POP EDI             ;
        POP ESI             ;
        POP EDX             ;
        POP ECX             ;
        POP EBX             ;
        RETN
```

Public Memory Management Calls

You now begin the *public* call definitions for memory management. Not all of the calls are presented here because some are very similar. The calling conventions all follow the Pascal-style definition.

AddGDTCallGate

`AddGDTCallGate()` builds and adds a GDT entry for a call gate, allowing access to OS procedures. This call doesn't check to see if the GDT descriptor for the call is already defined. It assumes you know what you are doing, and it overwrites the descriptor if it is already defined. The selector number is checked to make sure you're in range. This is 40h through the highest allowed call gate number. See Listing 19.18.

Listing 19.18. Adding a call gate to the GDT.

```
;
; IN: AX - Word with Call Gate ID type as follows:
;
;           DPL entry of 3 EC0x   (most likely)
;           DPL entry of 2 CC0x   (Not used in MMURTL)
;           DPL entry of 1 AC0x   (Not used in MMURTL)
;           DPL entry of 0 8C0x   (OS call ONLY)
;           (x = count of DWord params 0-F)
;
;     CX    Selector number for call gate in GDT (constants!)
;     ESI   Offset of entry point in segment of code to execute
;
; OUT:  EAX Returns Errors, else 0 if all's well
;
; USES: EAX, EBX, ECX, ESI, EFLAGS
```

```
PUBLIC __AddCallGate:
        CMP CX, 40h         ;Is number within range of callgates?
        JAE AddCG01         ;not too low.
        MOV EAX, ercBadGateNum
        RETF
AddCG01:
        MOVZX EBX, CX
        SUB EBX, 40         ;sub call gate base selector
        SHR EBX, 3          ;make index vice selector
        CMP EBX, nCallGates ;see if too high!
        JBE AddCG02         ;No.
        MOV EAX, ercBadGateNum  ;Yes.
        RETF
AddCG02:
        MOVZX EBX, CX       ;Extend selector into EBX
        ADD EBX, GDTBase    ;NOW a true offset in GDT
        MOV WORD PTR [EBX+02], 8    ;Put Code Seg selector into Call gate
        MOV [EBX], SI       ;0:15 of call offset
        SHR ESI, 16         ;move upper 16 of offset into SI
        MOV [EBX+06], SI    ;16:31 of call offset
        MOV [EBX+04], AX    ;call DPL & ndParams
        XOR EAX, EAX        ;0 = No Error
        RETF
```

AddIDTGate

`AddIDTGate()` builds and adds an interrupt descriptor table (IDT) trap, interrupt, or task gate. The selector of the call is always 8 for interrupt or trap; for a task gate, the selector of the call is the task state segment (TSS) of the task. See Listing 19.19.

Listing 19.19. Adding entries to the IDT.

```
;
; IN:   AX  - Word with Gate ID type as follows:
;               Trap Gate with DPL of 3       8F00
;               Interrupt Gate with DPL of 3  8E00
;               Task Gate with DPL of 3       8500
;
;       BX  - Selector of gate (08 or TSS selector for task gates)
;
;       CX  - Word with Interrupt Number (00-FF)
;
;       ESI - Offset of entry point in OS code to execute
;             (THIS MUST BE 0 FOR TASK GATES)
;
; USES: EAX, EBX, ECX, EDX, ESI, EFLAGS

PUBLIC __AddIDTGate:
        MOVZX EDX, CX               ;Extend INT Num into EDX
        SHL EDX, 3                  ;Gates are 8 bytes each (times 8)
        ADD EDX, OFFSET IDT         ;EDX now points to gate
        MOV WORD PTR [EDX+4], AX    ;Put Gate ID into gate
```

continues

Listing 19.19. continued

```
        MOV EAX, ESI
        MOV WORD PTR [EDX], AX      ;Put Offset 15:00 into gate
        SHR EAX, 16
        MOV WORD PTR [EDX+6], AX    ;Put Offset 31:16 into gate
        MOV WORD PTR [EDX+2], BX    ;Put in the selector
        RETF
;
```

AllocOSPage

`AllocOSPage()` allocates one or more pages of physical memory and returns a linear pointer to one or more pages of contiguous memory in the operating system space. A result code is returned in the EAX register. The steps involved depend on the internal routines described previously. The steps are these:

1. Ensure you have physical memory by checking `nPagesFree`.
2. Find a contiguous run of linear pages to allocate.
3. Allocate each physical page, placing it in the run of PTEs.

You search through the page tables for the current job and find enough contiguous PTEs to satisfy the request. If the current PT doesn't have enough contiguous entries, you add another page table to the operating-system page directory. This allows runs to span table entries. `AllocPage()` and `AllocDMAPage()` are not shown here because they are almost identical to `AllocOSPage()`, but not close enough to easily combine their code. See Listing 19.20.

Listing 19.20. Allocating a page of linear memory.

```
; Procedureal Interface :
;
;   AllocOSPage(dn4KPages,ppMemRet): dError
;
;
n4KPages    EQU [EBP+10h]   ;These equates are also used by AllocPage
ppMemRet    EQU [EBP+0Ch]   ;

PUBLIC __AllocOSPage:           ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        PUSH MemExch            ;Wait at the MemExch for Msg
        MOV EAX, pRunTSS        ;Put Msg in callers TSS Message Area
        ADD EAX, TSS_Msg
        PUSH EAX
        CALL FWORD PTR _WaitMsg
        CMP EAX,0h              ;Kernel Error??
        JNE SHORT ALOSPExit     ;Yes! Serious problem.

        MOV EAX,n4KPages        ;size of request
        OR  EAX,EAX             ;More than 0?
        JNZ ALOSP00             ;Yes
```

```
        MOV EAX,ercBadMemReq    ;Can't be zero!
        JMP ALOSPExit           ;
ALOSP00:
        CMP EAX, _nPagesFree    ;See if have enuf physical memory
        JBE ALOSP01             ;Yes
        MOV EAX, ErcNoMem       ;Sorry boss, we're maxed out
        JMP SHORT ALOSPExit
ALOSP01:
        MOV EBX,n4KPages        ;size of request
        XOR EAX, EAX            ;PD shadow offset needed by FindRun (0)
        CALL FindRun
        OR EAX, EAX             ;(0 = No Runs big enuf)
        JNZ SHORT ALOSP02       ;No Error!

        ;If we didn't find a run big enuf we add a page table

        CALL AddOSPT            ;Add a new page table (we need it!)
        OR EAX, EAX             ;See if it's 0 (0 = NO Error)
        JZ SHORT ALOSP01        ;Go back & try again
        JMP SHORT ALOSPExit     ;ERROR!!
ALOSP02:
                                ;EAX now has linear address
                                ;EBX still has count of pages
        CALL AddRun             ;Does not return error
                                ;EAX still has new linear address
        MOV EBX, ppMemRet       ;Get address of caller's pointer
        MOV [EBX], EAX          ;Give em new LinAdd
        XOR EAX, EAX            ;No error
ALOSPExit:                      ;
        PUSH EAX                ;Save last error
        PUSH MemExch            ;Send a Semaphore msg (so next guy can get in)
        PUSH 0FFFFFFF1h         ;
        PUSH 0FFFFFFF1h         ;
        CALL FWORD PTR _SendMsg ;
        POP EAX                 ;Get original error back (ignore kernel erc)
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
```

AliasMem

`AliasMem()` creates alias pages in the current job's PD and PTs, if the current PD is different than the PD for the job specified. This allows system services to access a caller memory for messaging without having to move data around. The pages are created at *user* protection level even if they are in operating memory space. This is for the benefit of system services installed in operating-system memory. Even if the address is only two bytes, if it crosses page boundaries, you need two pages. This wastes no physical memory, however—only an entry in a table.

The steps involved in the algorithm are these:

1. See if the current PD equals specified job PD. If so, Exit. No alias is needed.

2. Calculate how many entries (pages) will be needed.
3. See whether they are available.
4. Make PTE entries and return alias address to caller.

See Listing 19.21.

Listing 19.21. Code to alias memory.

```
;
; Procedureal Interface :
;
;   AliasMem(pMem, dcbMem, dJobNum, ppAliasRet): dError
;
;   pMem   is the address to alias.
;   dcbMem is the number of bytes needed for alias access.
;   JobNum is the job number that pMem belongs to.
;   ppAliasRet is the address to return the alias address to.
;
;
;pMem       EQU [EBP+24]        ;
;dcbMem     EQU [EBP+20]        ;
;JobNum     EQU [EBP+16]        ;
;ppAliasRet EQU [EBP+12]        ;

PUBLIC __AliasMem              ;
       PUSH EBP                ;
       MOV EBP,ESP             ;
       CALL GetCrntJobNum      ;Puts Current Job Num in EAX
       MOV EBX, [EBP+16]       ;Get Job number for pMem
       CMP EAX, EBX            ;Are they the same Page Directory??
       JNE ALSPBegin           ;No, alias it
       XOR EAX, EAX            ;Yes, No Error
       JMP ALSPDone            ;Exit, we're done

ALSPBegin:
       ;Now wait our turn with memory management

;      PUSH MemExch            ;Wait at the MemExch for Msg
;      MOV EAX, pRunTSS        ;Put Msg in callers TSS Message Area
;      ADD EAX, TSS_Msg
;      PUSH EAX
;      CALL FWORD PTR _WaitMsg
;      CMP EAX,0h              ;Kernel Error??
;      JNE ALSPDone            ;Yes! Serious problem.

       ; We're IN!

ALSP00:
       MOV EBX, [EBP+24]           ;pMem into EAX
       AND EBX, 0FFFh              ;MODULO 4096  (remainder)
       MOV EAX, [EBP+20]           ;dcbMem
       ADD EAX, EBX                ;Add the remainder of address
       ADD EAX, DWORD PTR [EBP+20] ;Add to dcbMem
       SHR EAX, 12                 ;EAX is nPages-1
       INC EAX                     ;EAX is now nPages we need!
       MOV ECX, EAX                ;Save nPages in ECX
```

```
        ;Now we find out whos memory we are in to make alias
        ;EAX is 256 for user space, 0 for OS
        ;EBX is number of pages for run

        CALL GetCrntJobNum          ;See if it is OS based service
        CMP EAX, 1                  ;OS Job?
        JE SHORT ALSP011            ;Yes
        MOV EAX, 256                ;No, User memory
        JMP SHORT ALSP01
ALSP011:
        XOR EAX, EAX                ;Set up for OS memory space
ALSP01:
        MOV EBX, ECX                ;Number of pages we need into EBX
        CALL FindRun                ;EAX has 0 or 256

        ;EAX is now linear address or 0 if no run is large enough
        ;EBX  still has count of pages

        OR EAX, EAX                 ;Was there enough PTEs?
        JNZ ALSP04                  ;Yes

        CALL GetCrntJobNum          ;See if it is OS based service
        CMP EAX, 1                  ;OS Job?
        JE SHORT ALSP03             ;Yes
        CALL AddUserPT              ;No!  Add a new USER page table
        JMP SHORT ALSP03
ALSP02:
        CALL AddOSPT                ;No!  Add a new OS page table
ALSP03:
        OR EAX, EAX                 ;0 = NO Error
        JZ SHORT ALSP00             ;Go back & try again
        JMP SHORT ALSPExit          ;ERROR!!

ALSP04:
        ;EAX has linear address (from find run) Sve in EDI
        ;EBX still has number of pages to alias
        ;Set ESI to linear address of pages to alias (from other job)
        ;Set EDX job number of job we are aliasing

        MOV EDI, EAX                ;Save alias page address base
        MOV ESI, [EBP+24]           ;Address to alias
        MOV EDX, [EBP+16]           ;Job number
        CALL AddAliasRun

        ;Now, take new alias mem and add trailing bits to address
        ;and return to caller so he knows address (EDI is lin add)

        MOV EAX, [EBP+24]           ;original pMem
        AND EAX, 0FFFh              ;Get remaining bits
        ADD EDI, EAX
        MOV ESI, [EBP+12]           ;pAliasRet
        MOV [ESI], EDI              ;Returned address to caller!
        XOR EAX, EAX                ;Set to 0 (no error)

        ;We are done
ALSPExit:                   ;
;       PUSH EAX            ;Save last error
```

continues

Listing 19.21. continued

```
;       PUSH MemExch            ;Send a Semaphore msg (so next guy can get in)
;       PUSH 0FFFFFFF1h             ;
;       PUSH 0FFFFFFF1h             ;
;       CALL FWORD PTR _SendMsg ;
;       POP EAX                 ;Get original error back (ignore kernel erc)
ALSPDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 16                 ;
```

DeAliasMem

`DeAliasMem()` zeroes out the page entries that were made during the `AliasMem()` call. You do not need to go through the operating system memory semaphore exchange because you are only zeroing out PTEs one at a time. This would not interfere with any of the memory allocation routines. See Listing 19.22.

Listing 19.22. Code to remove alias PTEs.

```
;
; Procedureal Interface :
;
;       DeAliasMem(pAliasMem, dcbAliasBytes, JobNum):ercType
;
;   pAliasMem is the address to "DeAlias"
;   dcbAliasBytes is the size of the original memory aliased
;
;pAliasMem      EQU [EBP+20]
;dcbAliasBytes  EQU [EBP+16]
;AliasJobNum    EQU [EBP+12]

PUBLIC __DeAliasMem             ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        ;Calculate the number of pages to DeAlias

        MOV EBX, [EBP+20]           ;pMem into EAX
        AND EBX, 0FFFh              ;MODULO 4096  (remainder)
        MOV EAX, [EBP+16]           ;dcbMem
        ADD EAX, EBX                ;Add the remainder of address
        ADD EAX, DWORD PTR [EBP+20] ;Add to dcbMem
        SHR EAX, 12                 ;EAX is nPages-1
        INC EAX                     ;EAX is now nPages we need!
        MOV ECX, EAX                ;Number of pages into EDI & ECX
        MOV EDI,ECX                 ;Save also in EDI (for compare)
        MOV EDX, [EBP+20]           ;Linear Mem to DeAlias
DALM01:
        MOV EBX, EDX            ;Address of next page to deallocate
        MOV EAX, [EBP+12]       ;Job num into EAX for LinToPhy
        CALL LinToPhy           ;Call this to get address of PTE into ESI
```

```
        ;Now we have Physical Address in EAX (we don't really need it)
        ;and pointer to PTE in ESI (We NEEDED THIS).
        ;See if PTE is an alias, if so just ZERO PTE.
        ;DO NOT deallocate the physical page

        MOV EBX, [ESI]          ;Get PTE into EBX
        TEST EBX, PRSNTBIT      ;Is page present (valid)???
        JNZ DALM02              ;Yes, it's page is present

        CMP ECX, EDI            ;NO! (did we do any at all)
        JNE DALM011             ;We did some.
        MOV EAX, ErcBadLinAdd   ;None at all!
        JMP SHORT DALMExit

DALM011:
        MOV EAX, ErcBadAlias    ;We dealiased what we could,
        JMP SHORT DALMExit      ;but less than you asked for!

DALM02:
        TEST EBX, ALIASBIT      ;Is page an ALIAS?
        JZ DALM03               ;NO - DO not zero it!

        ;If we got here the page is presnt and IS an alias
        ;so we zero out the page.

        XOR EAX, EAX            ;
        MOV [ESI], EAX          ;ZERO PTE entry
DALM03:

        ADD EDX, 4096           ;Next linear page
        LOOP DALM01
                                ;If we fall out EAX = ErcOK already
DALMExit:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;
```

DeAllocPage

`DeAllocPage()` frees up linear memory and physical memory that was acquired with any of the allocation calls. This will free physical pages only if the page is not marked as an alias. It will always free linear memory, providing it is valid. Even if you specify more pages than are valid, this will deallocate or dealias as much as it can before reaching an invalid page. See Listing 19.23.

Listing 19.23. Deallocating linear memory.

```
;
; Procedureal Interface :
;
;       DeAllocPage(pOrigMem, n4KPages):ercType
;
```

continues

Listing 19.22. continued

```
;   pOrigMem is a POINTER which should be pointing to memory page(s) to be
;   deallocate.  The lower 12 bits of the pointer is actually ignored
;   because we deallocate 4K pages.
;
;   n4KPages is the number of 4K pages to deallocate
;
pOrigMem    EQU [EBP+10h]
n4KPagesD   EQU [EBP+0Ch]       ;

PUBLIC __DeAllocPage:           ;
       PUSH EBP                 ;
       MOV EBP,ESP              ;

       PUSH MemExch             ;Wait at the MemExch for Msg
       MOV EAX, pRunTSS         ;Put Msg in callers TSS Message Area
       ADD EAX, TSS_Msg
       PUSH EAX
       CALL FWORD PTR _WaitMsg
       CMP EAX,0h               ;Error??
       JNE DAMExit              ;Yes!

       MOV EDX, pOrigMem        ;Linear Mem to deallocate
       AND EDX, 0FFFFF000h      ;Drop odd bits from address (MOD 4096)
       MOV ECX, n4KPagesD       ;Number of pages to deallocate

DAP01:
       MOV EBX, EDX             ;Address of next page to deallocate
       CALL GetCrntJobNum       ;Leave Job# in EAX for LinToPhy
       CALL LinToPhy            ;

       ;Now we have Physical Address in EAX
       ;and pointer to PTE in ESI.
       ;See if PTE is an alias, if so just ZERO PTE,
       ;else deallocate physical page THEN zero PTE

       MOV EBX, [ESI]           ;Get PTE into EBX
       TEST EBX, PRSNTBIT       ;Is page present (valid)???
       JNZ DAP02                ;Yes, it's page is present

       CMP ECX, n4KPagesD       ;NO! (did we do any at all)
       JNE DAP011               ;We did some..
       MOV EAX, ErcBadLinAdd    ;None at all!
       JMP SHORT DAMExit

DAP011:
       MOV EAX, ErcShortMem     ;We deallocated what we could,
       JMP SHORT DAMExit        ;but less than you asked for!

DAP02:
       TEST EBX, ALIASBIT       ;Is page an ALIAS?
       JNZ DAP03                ;Yes, it's an Alias

       ;If we got here the page is presnt and NOT an alias
       ;so we must unmark (release) the physical page.
```

```
        AND EBX, 0FFFFF000h     ;get rid of OS bits
        CALL UnMarkPage         ;

DAP03:
        XOR EAX, EAX            ;
        MOV [ESI], EAX          ;ZERO PTE entry

        ADD EDX, 4096           ;Next linear page
        LOOP DAP01
                                ;If we fall out EAX = ErcOK already
DAMExit:
        PUSH EAX                ;save Memory error

        PUSH MemExch            ;Send a dummy message to pick up
        PUSH 0FFFFFFF1h         ; so next guy can get in
        PUSH 0FFFFFFF1h
        CALL FWORD PTR _SendMsg ;
        CMP EAX, 0              ;Kernel error has priority
        JNE DAMExit1            ; over memory error

        POP EAX                 ;get Memory error back

DAMExit1:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
```

QueryMemPages

`QueryMemPages()` gives callers the number of remaining physical pages that can be allocated. You can do this because you can give them *any* physical page in the system. Their 1GB linear space is sure to hold what you have left. See Listing 19.24.

Listing 19.24. Code to find number of free pages.

```
;
; Procedureal Interface :
;
;       QueryMemPages(pdnPagesRet):ercType
;
;   pdnPagesRet is a pointer where you want the count of pages
;   left available returned
;
pMemleft    EQU [EBP+0Ch]

PUBLIC __QueryPages:            ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        MOV ESI, pMemLeft
```

continues

Listing 19.24. continued.

```
        MOV EAX,  _nPagesFree
        MOV [ESI], EAX
        XOR EAX, EAX            ;No Error

        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 4                  ;

;
```

GetPhyAdd

`GetPhyAdd()` returns the physical address for a linear address. This call would be used by device drivers that have allocated buffers in their data segment and need to know the physical address for DMA purposes. Keep in mind that the last 12 bits of the physical address always match the last 12 of the linear address because this is below the granularity of a page. See Listing 19.25.

Listing 19.25. Public function for linear to physical conversion.

```
;
;
; Procedureal Interface :
;
;       GetPhyAdd(JobNum, LinAdd, pPhyRet):ercType
;
;   LinAdd is the Linear address you want the physical address for
;   pPhyRet points to the unsigned long where the physical address will
;   be returned
;
;
;JobNum  EQU [EBP+20]
;LinAdd  EQU [EBP+16]
;pPhyRet EQU [EBP+12]
;

PUBLIC  __GetPhyAdd:            ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EBX, [EBP+16]       ; Linear Address
        MOV EAX, [EBP+20]       ; Job Number
        CALL LinToPhy
        MOV ESI, [EBP+12]       ; pPhyRet
        MOV [ESI], EAX          ;
        XOR EAX, EAX            ; No Error
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12                 ;
```

20

Timer Management Code

The timer code is the heart of all timing functions for the operating system. This file contains all of the code that uses or deals with timing functions in MMURTL.

The timer interrupt service routine (ISR) is also documented here, along with a piece of the kernel, actually the scheduler. It's at the end of the timer ISR.

Choosing a Standard Interval

Most operating systems have a timer interrupt function that is called at a fixed interval to update a continuously running counter. MMURTL is no exception. I experimented with several different intervals. I began with 50ms, but determined I couldn't get the resolution required for timing certain events, and it also wasn't fast enough to provide smooth, preemptive multitasking. I ended up with 10ms. Back when I started this project, 20-MHz machines were "screamers," and I was even worried about consuming bandwidth on machines that were slower than that. As you well know, with 100-MHz machines around, and anything less than 20-MHz all but disappearing, my worries were unfounded.

I did some time-consuming calculations to figure what percentage of CPU bandwidth I was using with my timer ISR. I added up all of the clock cycles in the ISR for a maximum time-consuming interrupt, as well as the fastest one possible. Somewhere between these two was the average. The average really depends on program loading, because the timer interrupt sends messages for the `Sleep()` and `Alarm()` functions. I was consuming less than 0.2 percent of available bandwidth. Even with these calculations, however, testing is the only real way to be sure it will function properly. Your bandwidth is the percentage of clocks executed for a given period, divided by the total clocks for the same period. In a 10-ms period, we have 200,000 clocks on a 20-MHz machine. My calculated worst case was approximately 150us, or 3,000 clocks, which works out to about 1.6 percent for the worst case. The average was far less. It was 0.2 percent.

Even more important than total bandwidth was the effect on interrupt latency. The most time-critical interrupts are nonbuffered serial communications, especially synchronous. A single communications channel running at 19,200 bps will interrupt every 520us. Two of them will come in at 260us.

The following are the instructions in the timer interrupt service routine that could be "looped" on for the maximum number of times in the worst case. The numbers before each instruction are the count of clocks to execute the instruction on a 20-MHz-386, a slow machine. This doesn't take into account nonaligned memory operands, certain problems with memory access such as noncached un-decoded instructions, memory wait-states, and so on, so I simply added 10 percent for the overhead, which is very conservative. See Listing 20.1.

Listing 20.1. Loop in timer ISR.

```
    IntTmr01:
6       CMP DWORD PTR [EAX+fInUse],FALSE
3       JE IntTmr03
```

```
6       CMP DWORD PTR [EAX+CountDown],0h
3       JNE IntTmr02
2       PUSH EAX
2       PUSH ECX
5       PUSH DWORD PTR [EAX+TmrRespExch]
2       MOV EAX, -1
2       PUSH EAX
2       PUSH EAX
40      CALL FWORD PTR _ISendMsg
2       POP ECX
2       POP EAX
6       MOV DWORD PTR [EAX+fInUse],FALSE
5       DEC nTmrBlksUsed
7       JMP IntTmr03
    IntTmr02:
6       DEC DWORD PTR [EAX+CountDown]
    IntTmr03:
2       ADD EAX,sTmrBlk
11      LOOP IntTmr01
```

This works out to 125 clocks for each loop, including the 10 percent overhead, plus 130 clocks for the execution of `ISendMsg()` in each loop. Multiply the total (255) times the maximum number of timer blocks (64), and it totals 16,320 clocks. That's a bunch! On a 20-MHz machine, that would be 860us just for this worst-case interrupt. But hold on a minute! That only happens if *all* of the tasks on the system want an alarm or must wake up from sleep *exactly* at the same time. The odds of this happening are truly astronomical. The real worst case would probably be more like 150us, and I tried a hundred calculations with a spreadsheet and finally with test programs. This I could live with. The only additional time consumer is to tack on a task switch at the end if you must preempt someone.

A task switch made by jumping to a TSS takes approximately 400 clocks on a 386, and 250 on a 486 or a Pentium. Four hundred clocks on a 20-MHz CPU is 50 nanoseconds times 400, or 20us. This doesn't cause a problem at all.

Timer Data

The following section is the data and constants defined for the timer interrupt and associated functions. `SwitchTick` and `dfHalted` are flags for the task-scheduling portion of the timer interrupt. See Listing 20.2.

Listing 20.2. Data and constants for timer code.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE TSS.INC
.ALIGN DWORD

; TIMER INTERRUPT COUNTER and TimerBlocks
```

continues

Listing 20.2. continued

```
; The Timer Block is a structure that contains
; the exchange number of a task that is sleeping,
; or one that has requested an alarm.
; Each TimerBlock is 12 Bytes long
; These are the offsets into the structure
;
sTmrBlk     EQU 12
fInUse      EQU 0   ;DD 00000000h
TmrRespExch EQU 4   ;DD 00000000h
CountDown   EQU 8   ;DD 00000000h
;
EXTRN SwitchTick DD
EXTRN dfHalted   DD

PUBLIC TimerTick    DD 0            ;Incremented every 10ms (0 on bootup).
PUBLIC nTmrBlksUsed DD 0            ;Number of timer blocks in use
PUBLIC rgTmrBlks    DB (sTmrBlk * nTmrBlks) DUP (0)
```

Timer Code

The timer interrupt and timing related function are all defined in the file `TmrCode.ASM`. The three external *near* calls, the first things defined in this code segment, allow the timer interrupt to access kernel helper functions from `Kernel.ASM`. The timer interrupt is part of the scheduling mechanism. See Listing 20.3.

Listing 20.3. External functions for timer code.

```
.CODE
EXTRN ChkRdyQ NEAR
EXTRN enQueueRdy NEAR
EXTRN deQueueRdy NEAR
;
```

The Timer Interrupt

The timer interrupt checks the timer blocks for values to decrement. The timer interrupt fires off every 10 milliseconds. The `Sleep()` and `Alarm()` functions set these blocks as callers require.

The timer interrupt code also performs the important function of keeping tabs on CPU hogs. It's really the only part of task scheduling that isn't cooperative. It is a small, yet very important part.

At all times on the system, only one task is actually executing. You have now interrupted that task. Other tasks may be waiting to run at the ready queue. They may have been placed there

by other ISRs, and they may be of an equal, or higher, priority than the task that is now running, which is the one you interrupted.

You check to see if the same task has been running for 30ms or more. If so, you call `ChkRdyQ` and check the priority of that task. If it is the same or higher, you switch to it. The task you interrupted is placed back on the ready queue in exactly the state you interrupted it. You couldn't reschedule tasks this way if you were using *interrupt tasks* because this would *nest* hardware task switches. It would require manipulating items in the TSS to make it look as if they weren't nested. Keep this in mind if you use interrupt tasks instead of interrupt procedures as I do. See Listing 20.4.

Listing 20.4. The timer interrupt service routine.

```
;
PUBLIC IntTimer:
        PUSHAD                          ;INTS are disabled automatically
        INC TimerTick                   ;Timer Tick, INT 20
        CMP nTmrBlksUsed, 0             ;Anyone sleeping or have an alarm set?
        JE  SHORT TaskCheck             ;No...

IntTmr00:                               ;Yes!
        LEA EAX,rgTmrBlks               ;EAX has addr of Timer Blocks
        MOV ECX,nTmrBlks                ;ECX has count of Timer Blocks
        CLD                             ;Move forward thru the blocks

IntTmr01:
        CMP DWORD PTR [EAX+fInUse],FALSE        ;Timer Block found?
        JE IntTmr03                     ;No  - goto next block
        CMP DWORD PTR [EAX+CountDown],0h        ;Yes - is count at zero?
        JNE IntTmr02                    ;No  - goto decrement it
        PUSH EAX                        ;save ptr to rgTmpBlk
        PUSH ECX                        ;save current count
        PUSH DWORD PTR [EAX+TmrRespExch]    ;Yes - ISend Message
        MOV EAX, -1                     ;FFFFFFFFh
        PUSH EAX                        ;bogus msg
        PUSH EAX                        ;bogus msg
        CALL FWORD PTR _ISendMsg        ;tell him his times up!
        POP ECX                         ;get count back
        POP EAX                                 ;get ptr back
        MOV DWORD PTR [EAX+fInUse],FALSE        ;Free up the timer block
        DEC nTmrBlksUsed                        ;Correct count of used blocks
        JMP IntTmr03                            ;skip decrement - empty blk

IntTmr02:
        DEC DWORD PTR [EAX+CountDown]   ;10ms more gone...

IntTmr03:
        ADD EAX,sTmrBlk                 ;next block please!
        LOOP IntTmr01                   ;unless were done

        ;We will now check to see if this guy has been running
        ;for more then 30ms (3 ticks). If so, we will
        ;switch him out if someone with an equal or higher pri
        ;is on the RdyQ
```

continues

Listing 20.4. continued

```
TaskCheck:
        MOV EAX, dfHalted       ;Is this from a HALTed state?
        OR EAX, EAX
        JNZ TaskCheckDone       ;YES.. no need to check tasks

        MOV EAX, TimerTick
        MOV EBX, SwitchTick
        SUB EAX, EBX
        CMP EAX, 3              ;Change this to change the "time slice"
        JL SHORT TaskCheckDone  ;Hasn't been 30ms yet for this guy!

        CALL ChkRdyQ            ;Get next highest Pri in EAX (Leave Queued)
        OR EAX, EAX             ;Is there one at all??
        JZ TaskCheckDone        ;No...

        MOV ESI, pRunTSS
        MOV DL, [ESI+Priority]  ;DL is pri of current
        CMP DL, [EAX+Priority]  ;Compare current pri to highest queued
                                ;The CMP subtracts Pri of Queued from
                                ;current pri. If the Queued Pri is LOWER
                                ;or Equal, we want to Switch. That means
                                ;if we got a CARRY on the subtract, his
                                ;number was higher and we DON'T
        JC TaskCheckDone        ;If current is higher(lower num), keep going!

        CALL deQueueRdy         ; Get high priority TSS off the RdyQ
        MOV EDI, EAX            ;  and save in EDI
        MOV EAX, ESI            ; Put current one in EAX
        CALL enQueueRdy         ;  and on the RdyQ

        MOV pRunTSS,EDI         ; Make the TSS in EDI the Running TSS
        MOV BX,[EDI+Tid]        ; Get the task ID
        MOV TSS_Sel,BX          ; Put it in the JumpAddr for Task Swtich
        INC _nSwitches          ; Keep track of how many switches for stats
        INC _nSlices            ; Keep track of how many preemptive switches
        MOV EAX, TimerTick      ; Save time of this switch for scheduler
        MOV SwitchTick, EAX     ;
        PUSH 0                     ;Must do this before we switch!
        CALL FWORD PTR _EndOfIRQ   ;
        JMP FWORD PTR [TSS]     ; JMP TSS (This is the task swtich)
        POPAD                   ;
        IRETD                   ;

TaskCheckDone:
        PUSH 0                     ;Must do this before we switch!
        CALL FWORD PTR _EndOfIRQ   ;
        POPAD                   ;
        IRETD                   ;
```

Sleep()

The sleep routine delays the calling task by setting up a timer block with a countdown value and by setting up an exchange to send a message to when the countdown reaches zero. The

timer interrupt sends the message and clears the block when it reaches zero. This requires an exchange. The exchange used is the `TSS_Exch` in the TSS for the current task. See Listing 20.5.

Listing 20.5. Code for the `Sleep` function.

```
DelayCnt    EQU [EBP+0Ch]
;
PUBLIC __Sleep:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        MOV EAX, DelayCnt
        CMP EAX, 0                  ;See if there's no delay
        JE Delay03
        LEA EAX,rgTmrBlks           ;EAX points to timer blocks
        MOV ECX,nTmrBlks            ;Count of timer blocks
        CLD                         ;clear direction flag
Delay01:
        CLI                         ;can't let others interfere
        CMP DWORD PTR [EAX+fInUse],FALSE      ;Empty block?
        JNE Delay02                 ;No  - goto next block
        MOV EBX,DelayCnt            ;Get delay count
        MOV [EAX+CountDown],EBX     ;
        MOV DWORD PTR [EAX+fInUse],TRUE       ;Use the Timer Block
        INC nTmrBlksUsed            ;Up the blocksInUse count
        MOV ECX,pRunTSS             ;Get TSS_Exch for our use
        MOV EBX,[ECX+TSS_Exch]      ;
        MOV [EAX+TmrRespExch],EBX   ;put it in timer block!
        STI
        PUSH EBX                    ;Pass exchange (for WaitMsg)
        ADD ECX,TSS_Msg             ;Offset of msg area
        PUSH ECX
        CALL FWORD PTR _WaitMsg       ;and Wait for it to come back
        MOV EAX,ErcOk               ;all is well
        JMP Delay03
Delay02:
        STI
        ADD EAX,sTmrBlk
        LOOP Delay01                ;unless were done
        MOV EAX,ErcNoMoreTBs        ;Sorry, out of timer blocks
Delay03:
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 4                      ;FAR return from publics
```

Alarm()

`Alarm()` sets up a timer block with a fixed message that will be sent when the countdown reaches zero. The message is *not* repeatable. It must be set up each time. The message will always be two dwords with 0FFFFFFFFh (-1) in each. See Listing 20.6.

Listing 20.6. Code for the `Alarm` function.

```
;    Procedural Interface:
;       Alarm(nAlarmExch, AlarmCnt):dErc
;
AlarmExch   EQU [EBP+10h]
AlarmCnt    EQU [EBP+0Ch]
;
;
PUBLIC __Alarm:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        MOV EAX, AlarmCnt
        CMP EAX, 0                  ;See if there's no delay
        JE Alarm03
        LEA EAX,rgTmrBlks           ;EAX points to timer blocks
        MOV ECX,nTmrBlks            ;Count of timer blocks
        CLD                         ;clear direction flag

Alarm01:
        CLI                         ;can't let others interfere
        CMP DWORD PTR [EAX+fInUse],FALSE      ;Empty block?
        JNE Alarm02                 ;No  - goto next block
        MOV EBX,AlarmCnt            ;Get delay count
        MOV [EAX+CountDown],EBX     ;
        MOV DWORD PTR [EAX+fInUse],TRUE       ;Use the Timer Block
        INC nTmrBlksUsed            ;Up the blocksInUse count
        MOV EBX, AlarmExch
        MOV [EAX+TmrRespExch],EBX   ;put it in timer block!
        STI                         ;It's OK to interrupt now
        MOV EAX,ErcOk               ;all is well
        JMP Alarm03

Alarm02:
        STI                         ;It's OK to interrupt now
        ADD EAX,sTmrBlk
        LOOP Alarm01                ;unless were done
        MOV EAX,ErcNoMoreTBs        ;Sorry, out of timer blocks

Alarm03:
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 8                      ;
;
```

KillAlarm()

`KillAlarm` searches the timer blocks looking for any block that is destined for the specified `Alarm` exchange. All alarms that are set to fire off to that exchange are killed. If the alarm is already queued through the kernel, which means the message has already been sent, nothing will stop it. See Listing 20.7.

Listing 20.7. Code for the `KillAlarm` function.

```
;    Procedural Interface:
;       KillAlarm(nAlarmExch):dErc
;
KAlarmExch  EQU [EBP+0Ch]
;
PUBLIC __KillAlarm:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        CMP nTmrBlksUsed, 0         ;No blocks in use
        JE KAlarm03                 ; so we get out!
        MOV EBX,KAlarmExch          ;Get exchange for killing alarms to
        LEA EAX,rgTmrBlks           ;EAX points to timer blocks
        MOV ECX,nTmrBlks            ;Count of timer blocks
        CLD                         ;clear direction flag

KAlarm01:
        CLI                         ;can't let others interfere
        CMP DWORD PTR [EAX+fInUse],TRUE     ;Block in use?
        JNE KAlarm02                ;No - goto next block
        CMP [EAX+TmrRespExch],EBX   ;Does this match the Exchange?
        JNE KAlarm02
        MOV DWORD PTR [EAX+fInUse],FALSE    ;Make Empty
        DEC nTmrBlksUsed            ;Make blocksInUse correct

KAlarm02:
        STI                         ;It's OK to interrupt now
        ADD EAX,sTmrBlk
        LOOP KAlarm01               ;unless were done

KAlarm03:
        XOR EAX,EAX                 ;ALl done -- ErcOk
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 4                      ;
;
```

MicroDelay()

`MicroDelay()` provides small-value timing delays for applications and device drivers. The count is in 15us increments. The timing for this delay is based on the toggle of the refresh bit from the system status port. The refresh bit is based on the system's quartz crystal oscillator, which drives the processor clock.

The task is actually not suspended at all. This forms an instruction loop checking the toggled value of an I/O port.

This call will not be very accurate for values less than 3 or 4 (45 to 60 microseconds). But it's still very much needed. The call can also be inaccurate because of interrupts that are not disabled. See Listing 20.8.

Listing 20.8. Code for the `MicroDelay` function.

```
;    Procedural Interface

;        MicroDelay(dDelay):derror

PUBLIC __MicroDelay:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        MOV ECX, [EBP+0Ch]          ;Get delay count
        CMP ECX, 0
        JE MDL01                    ;get out if they came in with 0!

MDL00:
        IN AL, 61h                  ;Get system status port
        AND AL, 10h                 ;check refrest bit
        CMP AH, AL                  ;Check toggle of bit
        JE MDL00                    ;No toggle yet
        MOV AH, AL                  ;Toggle! Move to AH for next compare
        LOOP MDL00

MDL01:
        XOR EAX, EAX
        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 4                      ;
```

GetCMOSTime()

This reads the time from the CMOS clock on the PC-ISA machines. MMURTL doesn't keep the time internally. The time is returned from the CMOS clock as a dword. The low order byte is the seconds in Binary Coded Decimal (BCD); the next byte is the minutes in BCD; the next byte is the Hours in BCD; and the high order byte is 0. See Listing 20.9.

Listing 20.9. Code to read the CMOS time.

```
;    Procedural Interface:
;       GetCMOSTime(pdTimeRet):derror

pCMOSTimeRet EQU [EBP+12]

PUBLIC __GetCMOSTime:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        XOR EBX, EBX                ;Clear time return

        MOV EAX,04h                 ;Hours
        OUT 70h,AL
        IN AL,71h
        MOV BL, AL

        SHL EBX, 8                  ;Minutes
        MOV EAX,02h
```

```
        OUT 70h,AL
        IN AL,71h
        MOV BL,AL

        SHL EBX, 8                  ;Seconds
        MOV EAX,00h
        OUT 70h,AL
        IN AL,71h
        MOV BL,AL

        MOV ESI, pCMOSTimeRet       ;Give 'em the time
        MOV [ESI], EBX
        XOR EAX, EAX                ; No Error

        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 4                      ;
```

GetCMOSDate()

The date is returned from the CMOS clock as a DWord. The low-order byte is the day of the week (BCD, 0–6 0=Sunday); the next byte is the day (BCD 1–31); the next byte is the month (BCD 1–12); and the high-order byte is year (BCD 0–99). See Listing 20.10.

Listing 20.10. Code to read the CMOS date.

```
;    Procedural Interface:
;
;        GetCMOSDate(pdTimeRet):derror
;
pCMOSDateRet EQU [EBP+12]
PUBLIC __GetCMOSDate:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        XOR EBX, EBX                ;Clear date return

        MOV EAX,09h                 ;Year
        OUT 70h,AL
        IN AL,71h
        MOV BL, AL
        SHL EBX, 8                  ;

        MOV EAX,08h                 ;Month
        OUT 70h,AL
        IN AL,71h
        MOV BL, AL
        SHL EBX, 8

        MOV EAX,07h                 ;Day of month
        OUT 70h,AL
        IN AL,71h
        MOV BL,AL
        SHL EBX, 8                  ;
```

continues

Listing 20.10. continued

```
        MOV EAX,06h                 ;Day of week
        OUT 70h,AL
        IN AL,71h
        MOV BL,AL

        MOV ESI, pCMOSDateRet       ;Give 'em the time
        MOV [ESI], EBX
        XOR EAX, EAX                ; No Error

        MOV ESP,EBP                 ;
        POP EBP                     ;
        RETF 4                      ;
```

GetTimerTick()

This returns the ever-increasing timer tick to the caller. MMURTL maintains a double word counter that begins at zero and is incremented until the machine is reset or powered down. Because it is a dword, there are over 4 billion ticks before rollover occurs. With a 10ms interval, this amounts to 262,800,000 ticks per month. This means the system tick counter will rollover approximately every 16 months. See Listing 20.11.

Listing 20.11. Code to return the timer tick.

```
; The procedural interface:
;
;   GetTimerTick(pdTickRet):derror
;
; The Current Timer Tick is returned (it's a DWord).
;
pTickRet EQU [EBP+12]

PUBLIC __GetTimerTick:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        MOV ESI, pTickRet
        MOV EAX, TimerTick
        MOV [ESI], EAX
        XOR EAX, EAX                ;No Error
        POP EBP                     ;
        RETF 4                      ;
```

Beep_Work()

`Beep_Work()` is an internal function (a helper) used by the public calls `Tone()` and `Beep()`. Hardware timer number 2, which is connected to the system internal speaker, is used to generate

tones. The system clock drives the timer so the formula to find the proper frequency is a function of the clock's frequency. The length of time the tone is on is controlled by the `Sleep()` function, which is driven by the system tick counter. See Listing 20.12.

Listing 20.12. Helper function for `Beep()` and `Tone()`.

```
;The clock freq to Timer 2 is 1.193182 Mhz
;To find the divisor of the clock, divide 1.193182Mhz by Desired Freq.
;This does all work for BEEP and TONE

;EBX needs the desired tone frequency in HERTZ
;ECX needs length of tone ON-TIME in 10ms increments

BEEP_Work:
        MOV AL, 10110110b       ;Timer 2, LSB, MSB, Binary
        OUT 43h, AL
        XOR EDX, EDX
        MOV EAX, 1193182        ;1.193182Mhz
        DIV EBX             ;DIVISOR is in EBX (Freq)
        OUT 42h, AL         ;Send quotient (left in AX)
        MOV AL, AH
        NOP
        NOP
        NOP
        NOP
        OUT 42h, AL
        IN AL, 61h
        OR AL, 00000011b
        PUSH EAX
        POP EAX
        OUT 61h, AL
        PUSH ECX            ;
        CALL FWORD PTR _Sleep       ;ECX is TIME ON in 50ms incs.
        IN AL, 61h
        NOP
        NOP
        NOP
        NOP
        AND AL, 11111100b
        OUT 61h, AL
        RETN
```

Beep()

This function has nothing to with the Road Runner; if it had, I would have called it `Beep_Beep()`. (Sorry, I couldn't resist.) This provides a fixed tone from the system's internal speaker as a public call for applications to make noise at the user. It uses the helper function `Beep_Work` described earlier. See Listing 20.13.

Listing 20.13. Code to produce a beep.

```
;    Procedural Interface:
;
;        Beep()

PUBLIC __Beep:
        PUSH EBP                    ;
        MOV EBP,ESP                 ;
        MOV EBX, 800                ;Freq
        MOV ECX, 35                 ;350ms
        CALL Beep_Work
        POP EBP                     ;
        RETF
```

Tone()

`Tone` allows the caller to specify a frequency and duration of a tone to be generated by the system's internal speaker. This call uses the helper function `Beep_Work` described earlier. See Listing 20.14.

Listing 20.14. Code to produce a tone.

```
;    Procedural Interface:
;
;        Tone(dFreq, dTickseRet):derror
;
;   dFreq is a DWord with the FREQUENCY in HERTZ
;   dTicks is a DWord with the duration of the tone in
;     10ms increments

ToneFreq    EQU [EBP+10h]
ToneTime    EQU [EBP+0Ch]
;
PUBLIC __Tone:
        PUSH EBP                        ;
        MOV EBP,ESP                     ;
        MOV EBX, ToneFreq
        MOV ECX, ToneTime
        CALL Beep_Work
        POP EBP                         ;
        RETF 8
```

21

Initialization Code

This chapter contains code from two separate files. The first file is Main.ASM. It is the entry point in the operating system, the first instruction executed after boot up. The second file is InitCode.ASM, a workhorse for Main.ASM.

OS Global Data

Listing 21.1 begins the data segment after the static tables that were in previous include files. The order of the first five files included in the assembler template file for MMURTL is critical. The tables before this portion of the data segment are the following:

The interrupt descriptor table (IDT)

The global descriptor table (GDT)

The initial OS page directory (PD)

The initial OS page table (PT)

The public call gate table

The public call gate table is not a processor-defined table as the others are. It defines the selectors for public call gates so the rest of the OS code can call into the OS.

Following these tables is the data segment portion from Main.ASM, which is presented later in this chapter.

In Chapter 7, "OS Initialization," I covered the basics of a generic sequence that would be required for initialization of an operating system.

Certain data structures and variables are allocated that may be used only once during the initialization process. Many things have to be started before you can allocate memory where the dynamic allocation of resources can begin. The code contains comments noting which of these items are temporary.

Periodically, you will see the `.ALIGN DWORD` command issued to the assembler. Even though the Intel processors can operate on data that is not aligned by its size, access to the data is faster if it is. I generally try to align data that is accessed often, and I don't worry about the rest of it. The assembler does no alignment on its own. Everything is packed, unless you tell it otherwise with the `.ALIGN` command. Listing 21.1.

Listing 21.1. Main operating system data.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE JOB.INC
.INCLUDE TSS.INC

.ALIGN DWORD
;=============================================================================
; Kernel Structures - See MOSEDF.INC for structure details.
```

```
;===========================================================================

PUBLIC MonTSS       DB sTSS dup (0) ; Initial TSS for OS/Monitor
PUBLIC DbgTSS       DB sTSS dup (0) ; Initial TSS for Debugger
PUBLIC pFreeTSS     DD NIL          ; Pointer to Free List of Task State Segs
PUBLIC pDynTSSs     DD 0            ; ptr to alloced mem for dynamic TSSs
PUBLIC _nTSSLeft    DD nTSS-2       ; For stats (less static TSSs)

;-----------------

PUBLIC rgLBs    DB (nLB*sLINKBLOCK) dup (0) ; pool of LBs
PUBLIC pFreeLB  DD NIL                      ; Ptr to Free List of Link Block
PUBLIC _nLBLeft DD nLB                      ; For Monitor stats

;-----------------
; The RUN Queue for "Ready to Run" tasks

PUBLIC RdyQ  DB (sQUEUE*nPRI) dup (0)   ; Priority Based Ready Queue

;-----------------
; Two static Job Control Blocks (JCBs) to get us kick-started.
; The rest of them are in allocated memory

PUBLIC MonJCB      DB sJCB dup (0)      ; Monitor JCB
PUBLIC DbgJCB      DB sJCB dup (0)      ; Debugger JCB

;-----------------

PUBLIC GDTLimit     DW 0000h        ;Global Descriptor Table Limit
PUBLIC GDTBase      DD 00000000h    ;base
PUBLIC IDTLimit     DW 0000h        ;Interrupt Descriptor Table Limit
PUBLIC IDTBase      DD 00000000h    ;base

;-----------------
PUBLIC rgSVC    DB (sSVC*nSVC) dup (0)      ; Setup an array of Service Descriptors

;-----------------
;Exchanges take up a fair amount of memory. In order to use certain kernel
;primitives, we need exchanges before they are allocated dynamically.
;We have an array of 3 exchanges set aside for this purpose.  After
;the dynamic array is allocated an initialized, we copy these into
;the first three dynamic exchanges.  These static exchanges are used
;by the Monitor TSS, Debugger TSS, and memory Management.

PUBLIC nExch  DD 3                  ; This will be changed after allocation
                                    ; of dynamic exchanges.
rgExchTmp  DB (sEXCH * 3) dup (0)   ; Setup three static temporary Exchanges
                                    ; for Monitor and Debugger TSSs
                                    ; These are moved to a dynamic Exch Array
                                    ; as soon as it's allocated.
PUBLIC   prgExch    DD OFFSET rgExchTmp     ; Pointer to current array of
Exchanges.
PUBLIC   pExchTmp   DD 0                ; Pointer to dynamic array Exchanges.
PUBLIC   _nEXCHLeft DD nDynEXCH         ; For Monitor stats

;-----------------
;Scheduling management variables
```

continues

Listing 21.1. continued

```
PUBLIC TSS          DD 00000000h    ; Used for jumping to next task
PUBLIC TSS_Sel      DW 0000h        ; "    "

.ALIGN DWORD

PUBLIC pRunTSS      DD NIL          ; Pointer to the Running TSS
PUBLIC SwitchTick   DD 0            ; Tick of last task switch
PUBLIC dfHalted     DD 0            ; nonzero if processor was halted

PUBLIC _nSwitches   DD 0            ; # of switches for statistics
PUBLIC _nSlices     DD 0            ; # of sliced switches for stats
PUBLIC _nReady      DD 0            ; # of task Ready to Run for stats
PUBLIC _nHalts      DD 0            ; # of times CPU halted for stats

;THESE ARE THE INITIAL STACKS FOR THE OS Monitor AND Debugger

OSStack     DD 0FFh DUP (00000000h) ;1K OS Monitor Stack
OSStackTop  DD 00000000h
OSStackSize EQU OSStackTop-OSStack

Stack1      DD 0FFh DUP (00000000h) ;1K Debugger Stack
Stack1Top   DD 00000000h
Stack1Size  EQU Stack1Top-Stack1

;-----------------

rgNewJob    DB 'New Job'            ;Placed in New JCBs
cbNewJob    EQU 7

rgOSJob     DB 'MMOS Monitor'       ;Placed in first JCB
cbOSJob     EQU 12

rgDbgJob    DB 'Debugger    '       ;Name for Job
cbDbgJob    DD 12                   ;Size of Job Name

_BootDrive  DD 00                   ;Source drive of the boot
```

Operating System Entry Point

The first instruction in Main.ASM is the entry point to the operating system. This is the first instruction executed after the operating system is booted. Look for the `.START` assembler command.

The address of this instruction is known in advance because the loader (boot) code moves the data and code segments of the operating system to a specific hard-coded address. The `.VIRTUAL` command allows you to tell the assembler where this code will execute, and all subsequent address calculations by the assembler are offset by this value. It is similar to the MS-DOS assembler `ORIGIN` command, with one major difference: DASM doesn't try to fill the segment prior to the virtual address with dead space.

The .VIRTUAL command can be used only once in an entire program, and it must be at the very beginning of the segment it's used in. For MMURTL, the start address is 10000h, the first byte of the second 64K in physical memory.

You'll see many warnings about the order of certain initialization routines. I had to make many warnings to myself just to keep from "housekeeping" the code into a nonworking state. I felt it was important to leave all these comments in for you. See Listing 21.2.

Listing 21.2. OS initialization code and entry point.

```
; This begins the OS Code Segment
.CODE
;
.VIRTUAL 10000h             ;64K boundry. This lets the assembler know
                            ;that this is the address where we execute
;
; BEGIN OS INITIALIZATION CODE
;
; This code is used to initialize the permanent OS structures
; and calls procedures that initialize dynamic structures too.
;
; "Will Robinson, WARNING, WARNING!! Dr. Smith is approaching!!!"
;  BEWARE ON INITIALIZATION.  The kernel structures and their
;  initialization routines are so interdependent, you must pay
;  close attention before you change the order of ANYTHING.
;  (Anything before we jump to the monitor code that is)

EXTRN InitCallGates NEAR
EXTRN InitOSPublics NEAR
EXTRN _Monitor NEAR
EXTRN InitIDT NEAR
EXTRN InitFreeLB NEAR
EXTRN InitDMA NEAR
EXTRN Set8259 NEAR
EXTRN InitKBD NEAR
EXTRN AddTSSDesc NEAR
EXTRN InitMemMgmt NEAR
EXTRN InitNewJCB NEAR
EXTRN InitVideo NEAR
EXTRN InitFreeTSS NEAR
EXTRN InitDynamicJCBs NEAR
EXTRN InitDynamicRQBs NEAR
EXTRN DbgTask NEAR
;=============================================================================
; Set up the initial Stack Pointer (SS = DS already).
; This is the first code we execute after the loader
; code throws us into protected mode. It's a FAR jump
; from that code...
;=============================================================================

.START
PUBLIC OSInitBegin:
        LEA EAX,OSStackTop          ; Setup initial OS Stack
        MOV ESP,EAX                 ; FIRST THING IN OS CODE.
        MOV _BootDrive, EDX         ; Left there from BootSector Code
```

continues

Listing 21.2. continued

```
;=============================================================================
; Set up OS Common Public for IDT and GDT Base and Limits - SECOND THING IN OS
;=============================================================================

        SGDT FWORD PTR GDTLimit      ;A formality - we know where they are!
        SIDT FWORD PTR IDTLimit      ;

;=============================================================================
; Setup Operating System Structures and motherboard hardware
; THIS IS RIGHT AFTER WE GET A STACK.
; YOU CAN'T ALLOCATE ANY OS RESOURCES UNTIL THIS CODE EXECUTES!!!
;=============================================================================

        CALL InitCallGates           ; Sets up all call gates as DUMMYs
                                     ;  except AddCallGate which
                                        must be made valid first!

        CALL InitOSPublics           ; Sets up OS PUBLIC call gates

        CALL InitIDT                 ;Sets up default Interrupt table
                                     ;NOTE: This uses CallGates!

        MOV ECX,nLB                  ; count of Link Blocks
        MOV EDX,sLinkBlock           ; EDX is size of a Link Block
        CALL InitFreeLB              ; Init the array of Link Blocks

        CALL InitDMA                 ; Sets up DMA with defaults

        CALL Set8259                 ; Set up 8259s for ints (before KBD)

        CALL InitKBD                 ; Initialize the Kbd hardware

        PUSH 0                       ; Highest IRQ number (all IRQs)
        CALL FWORD PTR _EndOfIRQ     ; Tell em to work

        ;Set time counter divisor to 11938 - 10ms ticks
        ;Freq in is 1.193182 Mhz/11932 = 100 per second
        ;or 1 every 10 ms.  (2E9Ch = 11932 decimal)

        MOV AL,9Ch                   ; Makes the timer Tick (lo)
        OUT 40h,AL                   ; 10 ms apart by setting
        MOV AL,02Eh                  ; clock divisior to (hi byte)
        OUT 40h,AL                   ; 11,932

        STI                          ; We are ready to GO (for now)

;=============================================================================
; The following code finishes the initialization procedures BEFORE the
; OS goes into paged memory mode.
; We set up an initial Task by filling a static TSS, creating and loading
; a descriptor entry for it in the GDT and we do the same for the debugger.
;=============================================================================

        ; Make the default TSS for the CPU to switch from a valid one.
        ; This TSS is a valid TSS after the first task switch.
```

```
        ; IMPORTANT - Allocate Exch and InitMemMgmt calls depend on
        ; pRunTSS being valid.  They can not be called before
        ; this next block of code!!!  Note that this TSS does NOT
        ; get placed in the linked list with the rest of the TSSs.
        ; It will never be free. We also have to manually make its
        ; entry in the GDT.

        ;The following code section builds a descriptor entry for
        ;the initial TSS (for Montitor program) and places it into the GDT

        MOV EAX, sTSS               ; Limit of TSS (TSS + SOFTSTATE)
        MOV EBX, 0089h              ; G(0),AV(0),LIM(0),P(1),DPL(0),B(0)
        MOV EDX, OFFSET MonTSS      ; Address of TSS
        MOV EDI, OFFSET rgTSSDesc   ; Address of GDT entry to fill
        CALL AddTSSDesc

        ;Now that we have valid Descriptor, we set up the TSS itself
        ;and Load Task Register with the descriptor (selector)
        ;Note that none of the TSS register values need to be filled in
        ;because they will be filled by the processor on the first
        ;task switch.

        MOV EBX, OFFSET MonTSS      ; Get ptr to initial TSS in EBX
        MOV pRunTSS,EBX             ; this IS our task now!!!
        MOV EAX, OFFSET rgTSSDesc   ; ptr to initial TSS descriptor
        SUB EAX, OFFSET GDT         ; Sub offset of GDT Base to get Sel of TSS
        MOV WORD PTR [EBX+TSS_IOBitBase], 0FFh; I/O Permission
        MOV [EBX+Tid],AX            ; Store TSS Selector in TSS (Task ID)
        LTR WORD PTR [EBX+Tid]              ; Setup the Task Register
        MOV BYTE PTR [EBX+Priority], 25     ; Priority 25 (monitor is another APP)
        MOV DWORD PTR [EBX+TSS_CR3], OFFSET PDir1   ;Physical address of PDir1
        MOV WORD PTR [EBX+TSSNum], 1        ;Number of first TSS (Duh)

        ;Set up Job Control Block for Monitor (always Job 1)
        ;JOB 0 is not allowed. First JCB IS job 1!

        MOV EAX, OFFSET MonJCB          ;
        MOV DWORD PTR [EAX+JobNum], 1   ;Number the JCB
        MOV EBX, OFFSET MonTSS          ;Must put ptr to JCB in TSS
        MOV [EBX+TSS_pJCB], EAX         ;pJCB into MonTSS
        MOV EBX, OFFSET PDir1           ;Page Directory
        MOV ESI, OFFSET rgOSJob         ;Job Name
        MOV ECX, cbOSJob                ;Size of Name
        XOR EDX, EDX                    ;NO pVirtVid yet (set up later in init)
        CALL InitNewJCB


; IMPORTANT - You can't call AllocExch before pRunTSS is VALID!

; THIS IS THE FIRST POINT AllocExh is valid

        MOV EAX, pRunTSS
        ADD EAX, TSS_Exch           ;Alloc exch for initial (first) TSS
        PUSH EAX
        CALL FWORD PTR _AllocExch
```

continues

Listing 21.2. continued

```
;=============================================================================
;
; Set up DEBUGGER Task and Job
; The debugger is set up as another job with its one task.
; The debugger must not be called (Int03) until this code executes,
; AND the video is initialized.
; We can't use NewTask because the debugger operates independent of
; the kernel until it is called (INT 03 or Exception)
;
       ;The following code section builds a descriptor entry for
       ;the initial TSS (for Debugger) and places it into the GDT

       MOV EAX, sTSS               ; Limit of TSS (TSS + SOFTSTATE)
       MOV EBX, 0089h              ; G(0),AV(0),LIM(0),P(1),DPL(0),B(0)
       MOV EDX, OFFSET DbgTSS      ; Address of Debugger TSS
       MOV EDI, OFFSET rgTSSDesc+8 ; Address of GDT entry to fill in
       CALL AddTSSDesc

       ;Now that we have valid Descriptor, we set up the TSS itself

       MOV EAX, OFFSET rgTSSDesc+8 ; ptr to second TSS descriptor
       SUB EAX, OFFSET GDT         ; Sub offset of GDT Base to get Sel of TSS
       MOV EBX, OFFSET DbgTSS      ; Get ptr to initial TSS in EBX
       MOV WORD PTR [EBX+TSS_IOBitBase], 0FFh; I/O Permission
       MOV [EBX+Tid],AX            ; Store TSS Selector in TSS (Task ID)
       MOV BYTE PTR [EBX+Priority], 1      ; Debugger is HIGH Priority
       MOV DWORD PTR [EBX+TSS_CR3], OFFSET PDir1   ;Physical address of PDir1
       MOV EDX, OFFSET DbgTask
       MOV [EBX+TSS_EIP],EDX
       MOV WORD PTR [EBX+TSS_CS],OSCodeSel     ; Put OSCodeSel in the TSS
       MOV WORD PTR [EBX+TSS_DS],DataSel       ; Put DataSel in the TSS
       MOV WORD PTR [EBX+TSS_ES],DataSel       ;
       MOV WORD PTR [EBX+TSS_FS],DataSel       ;
       MOV WORD PTR [EBX+TSS_GS],DataSel       ;
       MOV WORD PTR [EBX+TSS_SS],DataSel       ;
       MOV WORD PTR [EBX+TSS_SS0],DataSel      ;
       MOV EAX, OFFSET Stack1Top
       MOV DWORD PTR [EBX+TSS_ESP],EAX         ; A 1K Stack in the Dbg TSS
       MOV DWORD PTR [EBX+TSS_ESP0],EAX        ;
       MOV DWORD PTR [EBX+TSS_EFlags],00000202h    ; Load the Flags Register
       MOV WORD PTR [EBX+TSSNum], 2                ; Number of Debugger TSS

       ;Set up Job Control Block for Debugger
       ;JOB 0 is not allowed. First JCB IS job 1, debugger is always 2

       MOV EAX, OFFSET DbgJCB
       MOV DWORD PTR [EAX+JobNum], 2          ;Number the JCB
       MOV [EBX+TSS_pJCB], EAX     ;EBX still points to DbgTSS
       MOV EBX, OFFSET PDir1       ;Page Directory (OS PD to start)
       MOV ESI, OFFSET rgDbgJob    ;Name
       MOV ECX, cbDbgJob           ;size of name
       MOV EDX, 1                  ;Debugger gets video 1
       CALL InitNewJCB

       ;Now allocate the default exchange for Debugger
```

```
MOV EAX, OFFSET DbgTSS
ADD EAX, TSS_Exch          ;Alloc exch for Debugger TSS
PUSH EAX
CALL FWORD PTR _AllocExch
```

At this point, all of the static data that needs to be initialized is done. You had to set up all the necessary items for kernel messaging so you could initialize memory management. This was necessary because memory management uses messaging.

There are many "chickens and eggs" here. Move one incorrectly and *whamo*—the code doesn't work. Not even the debugger. After about 30 hours of debugging with no debugger, you'll appreciate the importance of keeping the order correct.

The memory-management initialization routines were presented in Chapter 19, "Memory Management Code." See Listing 21.3.

Listing 21.3. Continuation of OS initialization.

```
;=============================================================
; ALSO NOTE: The InitMemMgmt call enables PAGING!  Physical addresses
; will not necessarily match Linear addresses beyond this point.
; Pay attention!
;

        CALL InitMemMgmt            ;InitMemMgmt allocates an exch so
                                    ;this the first point it can be called.
                                    ;It also calls SEND!
```

After you have the memory allocation calls working, you can allocate space for all of the dynamic arrays and structures, such as task state segments, exchanges, and video buffers. See Listing 21.4.

Listing 21.4. Continuation of OS initialization.

```
;=============================================================
; FIRST POINT memory management calls are valid
;=============================================================

        ;Now we will allocate two virtual video screens (1 Page each)
        ;for the Monitor and Debugger and place them in the JCBs
        ;Also we set pVidMem for Monitor to VGATextBase address
        ;cause it has the active video by default

        PUSH 1                      ; 1 page
        MOV EAX, OFFSET MonJCB      ; Ptr to Monitor JCB
        ADD EAX, pVirtVid           ; Offset in JCB to pVirtVid
        PUSH EAX
        CALL FWORD PTR _AllocOSPage ; Get 'em!
        MOV EAX, OFFSET MonJCB      ; Make VirtVid Active for Monitor
        MOV DWORD PTR [EAX+pVidMem], VGATextBase
```

continues

Listing 21.4. continued

```
        PUSH 1                          ; 1 page
        MOV EAX, OFFSET DbgJCB          ;
        ADD EAX, pVirtVid
        PUSH EAX
        CALL FWORD PTR _AllocOSPage ; Get 'em!
        MOV EAX, OFFSET DbgJCB          ;
        MOV EBX, [EAX+pVirtVid]
        MOV [EAX+pVidMem], EBX          ; Video NOT active for Debugger

        CALL InitVideo                  ;Set Text Screen 0


;       MOV EAX,30423042h       ;BB on CYAN - So we know we are here!
;       MOV DS:VGATextBase+00h,EAX

;================================================================
; FIRST POINT video calls are valid
; FIRST POINT Debugger is working  (only because it needs the video)

; At this point we can allocate pages for dynamic structures and
; initialize them.
;================================================================

; Allocate 8 pages (32768 bytes) for 64 Task State Segments (structures).
; Then call InitFreeTSS (which fills them all in with default values)

        PUSH 8                          ; 8 pages for 64 TSSs (32768 bytes)
        MOV EAX, OFFSET pDynTSSs        ;
        PUSH EAX
        CALL FWORD PTR _AllocOSPage ; Get 'em!

        XOR EAX, EAX                    ; Clear allocated memory for TSSs
        MOV ECX, 8192                   ; (512 * 64 = 32768 = 8192 * 4)
        MOV EDI, pDynTSSs               ; where to store 0s
        REP STOSD                       ; Do it

        MOV EAX, pDynTSSs
        MOV ECX, nTSS-2                 ; count of dynamic TSSs (- OS & Dbgr TSS)
        CALL InitFreeTSS                ; Init the array of Process Control Blocks

        CALL InitDynamicJCBs
        CALL InitDynamicRQBs

;================================================================
; Allocate 1 page (4096 bytes) for 256 Exchanges (16*256=4096).
; Exchanges are 16 bytes each.  Then zero the memory which has
; the effect of initializing them because all fields in an Exch
; are zero if not allocated.

        PUSH 1                          ; 1 pages for 256 Exchs (4096 bytes)
        MOV EAX, OFFSET pExchTmp        ; Returns ptr to allocated mem in pJCBs
        PUSH EAX                        ;
        CALL FWORD PTR _AllocOSPage ; Get it!

        XOR EAX, EAX                    ; Clear allocated memory
```

```
MOV ECX, 1024                ; (4*1024=4096)
MOV EDI, pExchTmp            ; where to store 0s
REP STOSD                    ; Store EAX in 1024 locations

;Now we move the contents of the 3 static exchanges
;into the dynamic array.  This is 60 bytes for 3
;exchanges.

MOV ESI, prgExch             ; Source (static ones)
MOV EDI, pExchTmp            ; Destination (dynamic ones)
MOV ECX, 12                  ; 12 DWords (3 Exchanges)
REP MOVSD                    ; Move 'em!
MOV EAX, pExchTmp            ; The new ones
MOV prgExch, EAX             ; prgExch now points to new ones
MOV nExch, nDynEXCH          ; 256 to use (-3 already in use)
```

All of the basic resource managers are working at this point. You actually have a working operating system. From here, you go to code that will finish off the higher-level initialization functions, such as device drivers and loader tasks. See Listing 21.5.

Listing 21.5. End of initialization code.

```
        CALL _Monitor                   ;Head for the Monitor!!

        ;The Monitor call never comes back (it better not...)

        HLT         ;Well, you never know (I AM human)
;
```

Initialization Helpers

The file InitCode.ASM provides much of the support "grunt work" required to get the operating system up and running. The procedure `InitOSPublics()` sets up all the default values for call gates and interrupt vectors, and it initializes some of the free dynamically allocated structures, such as TSSs and link blocks.

Near the end of this file are the little assembly routines to fill in each of the 100-plus call gate entries in the GDT. I have not included them all here in print because they are so repetitive. This is noted in the comments. See Listing 21.6.

Listing 21.6. Initialization subroutines.

```
.CODE

EXTRN IntQ NEAR
EXTRN IntDivBy0 NEAR
EXTRN IntDbgSS  NEAR
```

continues

Listing 21.6. continued

```
EXTRN IntDebug  NEAR
EXTRN IntOverFlow NEAR
EXTRN INTOpCode   NEAR
EXTRN IntDblExc   NEAR
EXTRN INTInvTss   NEAR
EXTRN INTNoSeg    NEAR
EXTRN INTStkOvr   NEAR
EXTRN IntGP       NEAR
EXTRN INTPgFlt    NEAR
EXTRN IntPICU2    NEAR
EXTRN IntTimer    NEAR
EXTRN IntKeyBrd   NEAR

;=============================================================================
;The following code initializes structures just after bootup
;=============================================================================
;
; INPUT : ECX,EDX
; OUTPUT : NONE
; REGISTERS : EAX,EBX,ECX,FLAGS
; MODIFIES : pFreeLB,rgLBs
;
; This routine will initialize a free pool of link blocks.
; The data used in this algorithm are an array of ECX link blocks (rgLBs),
; each EDX bytes long and pointer to a list of free link blocks (pFreeLB).
;
; The pFreeLB pointer is set to address the first element in rgLBs. Each
; element of rgLBs is set to point to the next element of rgLBs. The
; last element of rgLBs is set to point to nothing (NIL).
;
PUBLIC InitFreeLB:
        LEA EAX,rgLBs           ; pFreeLB <= ^rgLBs;
        MOV pFreeLB,EAX         ;

LB_Loop:
        MOV EBX,EAX                     ; for I = 0 TO ECX
        ADD EAX,EDX                     ;  rgLBs[I].Next <=
        MOV [EBX+NextLB],EAX            ;    ^rgLBs[I+1];
        LOOP LB_Loop                    ;
        MOV DWORD PTR [EBX+NextLB], 0   ; rgFree[1023].Next <= NIL;
        RETN                            ;
;=============================================================================
; AddTSSDesc
; Builds a descriptor for a task and places it in the GDT.  If you
; check the Intel documentation the bits of data that hold this
; information are scattered through the descriptor entry, so
; we have to do some shifting, moving, anding and oring to get
; the descriptor the way the processor expects it.  See the Intel
; docs for a complete description of the placement of the bits.
;
; Note: The granularity bit represents the TSS itself, not the code
; that will run under it!
;
;
; IN:
;   EAX - Size of TSS
```

```
;   EBX - Decriptor type (default for OS TSS is 0089h)
;       (0089h - G(0),AV(0),LIM(0000),P(1),DPL(00),(010),B(0),(1))
;   EDX - Address of TSS
;   EDI - Address of Desc in GDT
; OUT:
;   GDT is updated with descriptor
; USED:
;   EFlags  (all other registers are saved)

PUBLIC AddTSSDesc:
        ;The following code section builds a descriptor entry for
        ;the TSS and places it into the GDT
        PUSH EAX
        PUSH EBX
        PUSH EDX
        PUSH EDI
        DEC EAX                 ; (Limit is size of TSS-1)
        SHL EBX,16              ; Chinese puzzle rotate
        ROL EDX,16              ; Exchange hi & lo words of Base Addr
        MOV BL,DH               ; Base 31 .. 24
        MOV BH,DL               ; Base 23 .. 16
        ROR EBX,8               ; Rotate to Final Alignment
        MOV DX,AX               ; Limit 15 .. 0 with Base 15 .. 0
        AND EAX,000F0000h       ; Mask Limit 19 .. 16
        OR EBX,EAX              ; OR into high order word
        MOV [EDI],EDX           ; Store lo double word
        MOV [EDI+4],EBX         ; Store hi double word
        POP EDI
        POP EDX
        POP EBX
        POP EAX
        RETN

;===========================================================================
; InitFreeTSS
; INPUT :  EAX, ECX
; OUTPUT : NONE
; USED :   ALL General registers, FLAGS
; MODIFIES : pFreeTSS (and the dynamic array of TSSs)
;
; This routine initializes the free pool of Task State Segments.
; On entry:
;   EAX points to the TSSs to initialize (allocated memory).
;   ECX has the count of TSSs to initialize.
;   The size of the TSS is taken from the constant sTSS.
;
; The pFreeTSS pointer is set to address the first TSS. The NextTSS
; field in each TSS is set to point to the next free TSS. The
; last TSS is set to point to nothing (NIL). The IOBitBase field is
; also set to FFFFh for NULL I/O permissions in each TSS.
; NOTE: The allocated memory area for the TSS MUST BE ZEROED before
; calling this routine.  By default, we add the TSS descriptors at OS
; protection level. If we spawn or add a User level TSS we must
; OR the DPL bits with 3!

PUBLIC InitFreeTSS:
        MOV pFreeTSS,EAX            ; First one free to use
        MOV EDI, OFFSET rgTSSDesc   ; ptr to TSS descriptors
```

continues

Listing 21.6. continued

```
        ADD EDI, 16                     ; First two TSSs are Static (Mon & Dbgr)
        MOV EDX, sTSS                   ; Size of TSS (in bytes) into EDX
        MOV EBX, 3                      ; Number of first dynamic TSS

TSS_Loop:
        MOV ESI,EAX                         ; for I = 0 TO ECX
        ADD EAX,EDX                         ;   EAX <= rgTSSs[I].Next
        MOV [ESI+NextTSS],EAX               ;      ^rgTSSs[I+1];
        MOV WORD PTR [ESI+TSS_IOBitBase], 0FFFFh    ; IOBitBase
        MOV [ESI+TSSNum], BX                ; TSS Number
        MOV WORD PTR [ESI+TSS_DS], DataSel      ;Set up for Data Selectors
        MOV WORD PTR [ESI+TSS_ES], DataSel
        MOV WORD PTR [ESI+TSS_FS], DataSel
        MOV WORD PTR [ESI+TSS_GS], DataSel
        MOV WORD PTR [ESI+TSS_SS], DataSel
        MOV WORD PTR [ESI+TSS_SS0], DataSel
        PUSH EAX                        ;Save pTSS
        MOV EAX,EDI                     ; Get offset of Curr TssDesc in EAX
        SUB EAX, OFFSET GDT             ; Sub offset of GDT Base to get Sel of TSS
        MOV WORD PTR [ESI+Tid],AX       ; Store TSS Selector in TSS (later use)
        PUSH EBX
        PUSH EDX

        MOV EAX,EDX             ; Size of TSS (TSS + SOFTSTATE)
        MOV EDX,ESI             ; Address of TSS
        MOV EBX,0089h           ; G(0),AV(0),LIM(0),P(1),DPL(0),B(0)

        CALL AddTSSDesc

        ADD EDI,8               ; Point to Next GDT Slot (for next one)
        POP EDX
        POP EBX
        POP EAX
        INC EBX                         ; TSS Number
        LOOP TSS_Loop                   ;
        MOV DWORD PTR [ESI+NextTSS], 0  ; rgFree[LastOne].Next <= NIL;
        RETN                            ;

;===========================================================================
; DUMMY CALL for uninitialized GDT call gate slots
;===========================================================================

DummyCall:
        MOV EAX, ercBadCallGate
        RETF
```

Besides setting up all the call gates with a dummy procedure, `InitCallGates` takes care of another problem. This problem is that the call to `AddCallGate()` is a public function called through a call gate. This means it can't add itself. This code manually adds the call for `AddCallGate()` so you can add the rest of them.

As I mentioned before, each of the more than 100 calls is placed in the GDT. There is room reserved for more than 600.

`InitIDT`, a little further down in the code, sets up each of the known entries in the IDT, as well as filling in all the unknown entries with a dummy ISR that simply returns from the interrupts. I have left all these in so that you can see which are *interrupt gates* and which are *interrupt traps*. The processor handles each of these a little differently. See Listing 21.7.

Listing 21.7. Initialize call gates and the IDT.

```
;=============================================================================
; InitCallGates inits the array of call gates with an entry to a generic
; handler that returns ErcNotInstalled when called. This prevents new code
; running on old MMURTLs or systems where special call gates don't exist
; without crashing too horribly.
;
; IN: Nothing
; Out : Nothing
; Used : ALL registers and flags
;
PUBLIC InitCallGates:

        ;First we set up all call gates to point to a
        ;dummy procedure

        MOV ECX, nCallGates     ;Number of callgates to init

InitCG01:
        PUSH ECX                ;Save nCallGates
        DEC ECX                 ;make it an index, not the count
        SHL ECX, 3              ;
        ADD ECX, 40h            ;Now ecx is selector number
        MOV EAX, 0EC00h         ;DPL 3, 0 Params
        MOV DX, OSCodeSel
        MOV ESI, OFFSET DummyCall

        ;Same code as in PUBLIC AddCallGate
        MOVZX EBX, CX
        SUB EBX, 40         ;sub call gate base selector
        SHR EBX, 3          ;make index vice selector
        MOVZX EBX, CX       ;Extend selector into EBX
        ADD EBX, GDTBase    ;NOW a true offset in GDT
        MOV WORD PTR [EBX+02], 8    ;Put Code Seg selector into Call gate
        MOV [EBX], SI       ;0:15 of call offset
        SHR ESI, 16         ;move upper 16 of offset into SI
        MOV [EBX+06], SI    ;16:31 of call offset
        MOV [EBX+04], AX    ;call DPL & ndParams

        POP ECX                 ;ignore error...
        LOOP InitCG01           ;This decrements ECX till 0

        ;Another chicken and egg here.. In order to be able
        ;to call the FAR PUBLIC "AddCallGate" though a callgate,
        ;we have to add it as a callgate... ok....

        MOV EAX, 08C00h     ;AddCallGate — 0 DWord Params  DPL 0
        MOV ECX, 0C8h
        MOV DX, OSCodeSel
```

continues

Listing 21.7. continued

```
        MOV ESI, OFFSET __AddCallGate
        ;Same code as in PUBLIC AddCallGate
        MOVZX EBX, CX
        SUB EBX, 40          ;sub call gate base selector
        SHR EBX, 3           ;make index vice selector
        MOVZX EBX, CX        ;Extend selector into EBX
        ADD EBX, GDTBase     ;NOW a true offset in GDT
        MOV WORD PTR [EBX+02], 8    ;Put Code Seg selector into Call gate
        MOV [EBX], SI        ;0:15 of call offset
        SHR ESI, 16          ;move upper 16 of offset into SI
        MOV [EBX+06], SI     ;16:31 of call offset
        MOV [EBX+04], AX     ;call DPL & ndParams
        RETN

;============================================================================
; InitIDT
; First, inits the IDT with 256 entries to a generic
; handler that does nothing (except IRETD) when called.
; Second, adds each of the basic IDT entries for included
; software and hardware interrupt handlers.
; ISRs loaded with device drivers must use SetIRQVector.
;
; IN    : Nothing
; Out   : Nothing
; Used  : ALL registers and flags
;
PUBLIC InitIDT:
        MOV ECX, 255             ;Last IDT Entry
InitID01:
        PUSH ECX
        MOV EAX, 08F00h          ;DPL 3, TRAP GATE
        MOV EBX, OSCodeSel
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate
        POP ECX
        LOOP InitID01

        ;Now we add each of the known interrupts

        MOV ECX, 0               ;Divide By Zero
        MOV EAX, 08F00h          ;DPL 3, TRAP GATE
        MOV EBX, OSCodeSel       ;
        MOV ESI, OFFSET IntDivBy0
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 1               ;Single Step
        MOV EAX, 08F00h          ;DPL 3, Trap gate
        MOV EBX, OSCodeSel       ;
        MOV ESI, Offset IntDbgSS
        CALL FWORD PTR _AddIDTGate

        ;Trying 8E00 (Int gate vice trap gate which leave Ints disabled)

        MOV ECX, 3               ;Breakpoint
        MOV EAX, 08E00h          ;DPL 3, Trap Gate (for Debugger) WAS 8F00
        MOV EBX, OSCodeSel       ;This will be filled in with TSS of Dbgr later
```

```
MOV ESI, OFFSET IntDebug
CALL FWORD PTR _AddIDTGate

MOV ECX, 4              ;Overflow
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntOverFlow
CALL FWORD PTR _AddIDTGate

MOV ECX, 6              ;Invalid OPcode
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntOpCode
CALL FWORD PTR _AddIDTGate

MOV ECX, 8              ;Double Exception
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntDblExc
CALL FWORD PTR _AddIDTGate

MOV ECX, 0Ah            ;Invalid TSS
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntInvTSS
CALL FWORD PTR _AddIDTGate

MOV ECX, 0Bh            ;Seg Not Present
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntNoSeg
CALL FWORD PTR _AddIDTGate

MOV ECX, 0Ch            ;Int Stack Overflow
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntStkOvr
CALL FWORD PTR _AddIDTGate

MOV ECX, 0Dh            ;GP fault
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntGP
CALL FWORD PTR _AddIDTGate

MOV ECX, 0Eh            ;Int Page Fault
MOV EAX, 08F00h         ;DPL 3, TRAP GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntPgFlt
CALL FWORD PTR _AddIDTGate

MOV ECX, 20h            ;Int TIMER              IRQ0
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntTimer
CALL FWORD PTR _AddIDTGate
```

continues

Listing 21.7. continued

```
MOV ECX, 21h            ;Int KEYBOARD           IRQ1
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntKeyBrd
CALL FWORD PTR _AddIDTGate

MOV ECX, 22h            ;Int PICU 2 (from PICU) IRQ2
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntPICU2
CALL FWORD PTR _AddIDTGate

MOV ECX, 23h            ;Int COM2               IRQ3
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 24h            ;Int COM1               IRQ4
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 25h            ;Int LPT2               IRQ5
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 26h            ;Int Floppy             IRQ6
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ    ;FDD will set this himself
CALL FWORD PTR _AddIDTGate

MOV ECX, 27h            ;IntLPT1                IRQ7
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 28h            ;Int .........          IRQ8
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 29h            ;Int .........          IRQ9
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
MOV EBX, OSCodeSel      ;
MOV ESI, OFFSET IntQ
CALL FWORD PTR _AddIDTGate

MOV ECX, 2Ah            ;Int .........          IRQ10
MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
```

```
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 2Bh            ;Int .........          IRQ11
        MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 2Ch            ;Int .........          IRQ12
        MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 2Dh            ;Int .........          IRQ13
        MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 2Eh            ;Int .........          IRQ14
        MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        MOV ECX, 2Fh            ;Int .........          IRQ15
        MOV EAX, 08E00h         ;DPL 3, INTERRUPT GATE
        MOV EBX, OSCodeSel      ;
        MOV ESI, OFFSET IntQ
        CALL FWORD PTR _AddIDTGate

        RETN


;=============================================================================
; InitOSPublics adds all OS primitives to the array of call gates. This can't be
called
; before initcallgates, but MUST be called before the first far call to any
; OS primitive thorugh a call gate!!!
; IF YOU ADD AN OS PUBLIC MAKE SURE IT GETS PUT HERE!!!!!
;
; IN   : Nothing
; Out  : Nothing
; Used : ALL registers and flags
;
PUBLIC InitOSPublics:

        MOV EAX, 0EC02h     ;WaitMsg — 2 DWord Params, DPL 3
        MOV ECX, 40h
        MOV DX, OSCodeSel
        MOV ESI, OFFSET __WaitMsg
        CALL FWORD PTR _AddCallGate
```

continues

Listing 21.7. continued

```
MOV EAX, 0EC03h     ;SendMsg — 3 DWord Params, DPL 3
MOV ECX, 48h
MOV DX, OSCodeSel
MOV ESI, OFFSET __SendMsg
CALL FWORD PTR _AddCallGate

MOV EAX, 08C03h     ;ISendMsg — 3 DWord params, DPL 0
MOV ECX, 50h
MOV DX, OSCodeSel
MOV ESI, OFFSET __ISendMsg
CALL FWORD PTR _AddCallGate

MOV EAX, 0EC01h     ;Set Priority - 1Dword param, DPL 3
MOV ECX, 58h
MOV DX, OSCodeSel
MOV ESI, OFFSET __SetPriority
CALL FWORD PTR _AddCallGate

;The rest of the call gate set ups go here.

RETN
```

22

Job Management Code

After you get the kernel out of the way and when you want to begin loading things to run on your system, this is where you'll end up. The program loader is in this chapter.

Before I wrote this code, pictures of the complications of writing a loader floated in my head for quite a while. Writing the loader was every bit as difficult as I imagined. The process of allocating all the resources, making sure that each was correctly set and filled out, and finally starting the new task can be complicated. I have tried to break down the process logically and keep it all in a high-level language.

Reclamation of Resources

Equally as difficult was the death of a job and the reclamation of its valuable resources. In the functions `ExitJob()` and `Chain()`, I drift in and out of assembly language to do things such as switch to a temporary stack when the running task no longer has a valid stack. When things don't work right, it is usually quite a mess.

The resources you reclaim include all the memory, exchanges, link blocks, request blocks, the JCB, TSSs, and finally the operating system memory. Pieces of related code in the kernel, the monitor, and other places provide assistance with resource reclamation.

Job Management Helpers

The file JobCode.ASM provides many small helper functions for job management. These calls, many of them public functions, are used by almost every module in the operating system. See Listing 22.1.

Listing 22.1. Job management subroutines.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE JOB.INC
.INCLUDE TSS.INC

PUBLIC pFreeJCB     DD 0                ; Ptr to free Job Control Blocks
PUBLIC pJCBs        DD 0                ; JCBs are in allocated memory
PUBLIC _nJCBLeft    DD nJCBs            ; For Monitor stats

EXTRN MonJCB  DB    ; Monitor JCB reference
EXTRN _BootDrive DD ; From Main.asm

;======= End data,Begin Code ==================

.CODE
;================================================================
;InitNewJCB is used initially by the OS to fill in the first two
;jobs (Monitor & Debugger)
;
```

```
PUBLIC InitNewJCB:
; INPUT :   EAX — Ptr to JCB that is to be filled in
;           EBX — Linear Ptr to Page Directory for Job
;           ESI — pbJobName
;           ECX — cbJobName
;           EDX — Pointer to Job Virtual Video Buffer (all jobs have one!)
;
; OUTPUT :  JOB Number in EAX
; USED :    EAX, EBX, ECX, EDX, EDI, ESI, EFlags
; MODIFIES : JCB pointed to in EBX
;
; This fills in a JCB with new information.  This is used to initilaize
; a new JCB during OS init and when a new Job is loaded and run.
;
        MOV [EAX+JcbPD],EBX             ;Put Ptr to PD into JCB
        MOV EDI, EAX                    ;EDI points to JCB
        ADD EDI, sbJobName              ;Now to JobName
        MOV BYTE PTR [EDI], CL          ;size is filled in
        INC EDI                         ;first byte of name
        REP MOVSB                       ;Move it in
        MOV [EAX+pVirtVid], EDX         ;Video number is in JCB
        MOV DWORD PTR [EAX+nCols], 80   ;
        MOV DWORD PTR [EAX+nLines], 25  ;
        MOV DWORD PTR [EAX+NormAttr], 7 ;
        MOV EAX, [EAX+JobNum]
        RETN


;===========================================================================
; InitFreeJCB
; INPUT :   EAX - Address of JCBs to be initialized
;           ECX - Count of JCBs
;           EDX - Size of JCBs
; OUTPUT :  NONE
; USED:     EAX,EBX,ECX,EDX,ESI EFLAGS
; MODIFIES: pFreeJCB, pJCBs
;
; This routine will initialize the free pool of Job Control Blocks (JCBs).
; EAX points to the first JCB,
; ECX is count of JCBs,
; EDX is size of each JCB.
;
; The pFreeJCB pointer is set to address the first element in rgJCBs.
; Each element of rgJCBs is set to point to the next element of rgJCBs.
; The last element of rgJCBs is set to point to nothing (NIL).
; The JCBs are also sequentially numbered. We can't use its position
; in the array because some JCBs are static (Mon and Debugger), while
; others (the ones we are initializing now) are dynamicly allocated.
;
PUBLIC InitFreeJCB:
        MOV pFreeJCB,EAX        ;Set up OS pointer to list
        MOV pJCBs, EAX          ;Set up global ptr to first JCB
        MOV EBX, 3              ;1st number for Dynamic JCBs
JCB_Loop:
        MOV ESI,EAX             ;EBX has pointer to current one
        ADD EAX,EDX             ;EAX points to next one
        MOV [ESI+NextJCB],EAX   ;Make current point to next
        MOV [ESI+JobNum], EBX   ;Number it
```

continues

Listing 22.1. continued

```
        INC EBX
        LOOP JCB_Loop           ;Go back till done
        MOV DWORD PTR [ESI+NextJCB], 0   ;Make last one NIL
        RETN                    ;

;=============================================================
; Allocate 4 pages (16384 bytes) for 32 Job Control Blocks (structures).
; Then call InitFreeJCB

PUBLIC InitDynamicJCBs:
        PUSH 4                          ; 4 pages for 32 JCBs (16384 bytes)
        MOV EAX, OFFSET pJCBs           ; Returns ptr to allocated mem in pJCBs
        PUSH EAX                        ;
        CALL FWORD PTR _AllocOSPage ; Get 'em!

        XOR EAX, EAX                    ; Clear allocated memory for JCBs
        MOV ECX, 4096                   ; (4*4096=16384 - DWORDS!)
        MOV EDI, pJCBs                  ; where to store 0s
        REP STOSD                       ; Do it

        MOV EAX, pJCBs                  ; Ptr to JCBs
        MOV ECX, nJCBs                  ; Count of Job Control Blocks
        MOV EDX, sJCB                   ; EDX is size of a JCB
        CALL InitFreeJCB                ; Init the array of JCBs
        RETN

;=============================================================================

PUBLIC NewJCB:
; INPUT : NONE
; OUTPUT : EAX
; REGISTERS : EAX,EBX,FLAGS
; MODIFIES : pFreeJCB
;
; This routine will return to the caller a pointer to the next free jcb.
; The data used in this algorithm is the free jcb pointer (pFreeJCB).
; This routine will return in EAX register the address of the next free jcb.
; If none exists, then EAX will contain NIL (0). This routine will also
; update the value of pFreeJCB to point to the next "unused" JCB in
; the free pool.
;
        MOV EAX,pFreeJCB        ;Get OS pointer to JCBs
        CMP EAX,0               ;IF pFreeJCB=NIL THEN Return;
        JE NewJCBDone           ;
        MOV EBX,[EAX+NextJCB]   ;Get pointer to next free one
        MOV pFreeJCB,EBX        ;Put it in OS pointer
        DEC DWORD PTR _nJCBLeft         ;
NewJCBDone:
        RETN                    ;

;=============================================================================

PUBLIC DisposeJCB:
; INPUT : EAX
; OUTPUT : NONE
; REGISTERS : EBX,FLAGS
```

```
; MODIFIES : pFreeJCB
;
; This routine will place the jcb pointed to by EAX back into the free
; pool of JCBs pointed to by (pFreeJCB) if EAX is not NIL.
; This invalidates the JCB by placing 0 in JcbPD.
;
        CMP EAX, 0                      ; If pJCBin = NIL THEN Return;
        JE DispJCBDone                  ;
        MOV DWORD PTR [EAX+JcbPD], 0    ;Invalidate JCB
        MOV EBX,pFreeJCB                ;EBX has OS ptr to free list
        MOV [EAX+NextJCB],EBX           ;Move it into newly freed JCB
        MOV pFreeJCB,EAX                ;Move ptr to newly frred JCB to OS
        INC DWORD PTR _nJCBLeft                 ;
DispJCBDone:
        RETN                    ;

;=============================================================
;
; GetpCrntJCB
; Returns a pointer to the current Job Control Block in EAX.
; This is based on which Task is executing.  All TSSs are
; assigned to a Job.  A Job may have more than one Task.
;
; INPUT:    Nothing
; OUTPUT:   EAX — Linear Address of current JCB
; USED:     EAX, EFlags
;
2PUBLIC GetpCrntJCB:
        MOV EAX, pRunTSS        ;Current Task State Segment
        MOV EAX, [EAX+TSS_pJCB] ;Pointer to JCB
        RETN

;=============================================================
;
; GetCrntJobNum
; Many OS functions deal with the Job number. The Job number
; is a field in the JCB structure.
; Returns the Job number for the currently executing task.
; This is based on which Task is executing.  All TSSs are
; assigned to a Job!  A Job may have more than one Task.
;
; INPUT:    Nothing
; OUTPUT:   EAX — Current Job Number
; USED:     EAX, EFlags
;
PUBLIC GetCrntJobNum:
        CALL GetpCrntJCB
        MOV EAX, [EAX+JobNum]           ;Current JCB
        RETN

;=============================================================
;
; GetpJCB
; Returns a pointer to a Job Control Block identified by number
; in EAX.  All TSSs are assigned to a Job.
;
; INPUT:    EAX — Job Number of desired pJCB
```

continues

Listing 22.1. continued

```
; OUTPUT:   EAX — Linear Address of the JCB or 0 for invalid number
; USED:     EAX, EFlags
;
PUBLIC GetpJCB:
        PUSH EDX
        CMP EAX, 1
        JNE GetpJCB1
        MOV EAX, OFFSET MonJCB
        POP EDX
        RETN
GetpJCB1:
        CMP EAX, 2
        JNE GetpJCB2
        MOV EAX, OFFSET DbgJCB
        POP EDX
        RETN
GetpJCB2:
        CMP EAX, nJCBs+2        ;Add in two static JCBs
        JLE GetpJCB3            ;Within range of JCBs
        XOR EAX, EAX
        POP EDX
        RETN
GetpJCB3:
        SUB EAX, 3              ;Take off static JCBs+1 (make it an offset)
        MOV EDX, sJCB
        MUL EDX                 ;Times size of JCB
        ADD EAX, pJCBs          ;Now points to desired JCB
        POP EDX
        RETN                    ;

;=============================================================
;
; GetJobNum
; Many OS functions deal with the Job number. The Job number
; is a field in the JCB structure.
; Returns the Job number for the pJCB in EAX in EAX.
;
; INPUT:    EAX pJCB we want job number from.
; OUTPUT:   EAX — Current Job Number
; USED:     EAX, EFlags
;
PUBLIC GetJobNum:
        MOV EAX, [EAX+JobNum]           ;Current JCB
        RETN

;=============================================================
;
; AllocJCB  (NEAR)
; This allocates a new JCB (from the pool).  This is a NEAR
; call to support the public job management calls in high level
; languages.
;
; Procedural Interface :
;
;       AllocJCB(pdJobNumRet, ppJCBRet):ercType
```

```
;
;   pdJobNumRet is the number of the new JCB.
;   pJCBRet is a pointer where you want the pointer to the new JCB is returned.
;
;   ErcNoMoreJCBs will be returned if no more JCBs are avaialble.
;
; pdJobNum          EQU [EBP+12]
; pJCBRet           EQU [EBP+8]

PUBLIC _AllocJCB:               ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        CALL NewJCB             ; Get a new JCB
        OR EAX, EAX             ;
        JNZ SHORT AJCB01        ; We got one!
        MOV EAX, ErcNoMoreJCBs  ; Sorry, out of them!
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETN 8                  ;

AJCB01:
        MOV ESI, [EBP+8]        ;pJCBRet
        MOV [ESI], EAX
        MOV ESI, [EBP+12]       ;Job Num
        CALL GetJobNum          ;
        MOV [ESI], EAX          ;
        XOR EAX, EAX            ;No error
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETN 8                  ;

;============================================================
;
; DeAllocJCB  (NEAR)
; This Deallocates a JCB (returns it to the pool).  This is a NEAR
; call to support the public job management calls in high level
; languages in the OS code.
;
; Procedural Interface :
;
;       DeAllocJCB(pJCB):ercType
;
;   pJCB is a pointer the JCB to be deallocated.
;
;   ErcNoMoreJCBs will be returned if no more JCBs are avaialble.
;
; pJCB          EQU [EBP+8]

PUBLIC _DeAllocJCB:             ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        MOV EAX, [EBP+8]        ; pJCB
        CALL DisposeJCB         ; Get a new JCB
        XOR EAX, EAX            ;
        MOV ESP,EBP             ;
        POP EBP                 ;
```

continues

Listing 22.1. continued

```
        RETN 4                  ;

;=========================================================
;===== BEGIN PUBLIC FAR JOB CALLS ========================
;=========================================================
;
; GetpJCB
; This PUBLIC returns a pointer to the JCB for the JobNum
; you specifiy.
;
; Procedural Interface :

;       GetpJCB(dJobNum, pJCBRet):ercType
;
;   dJobNum is the number of the JCB you want.
;   pJCBRet is a pointer where you want the JCB returned.
;
;   ErcBadJobNum will be returned if dJobNum is out of range
;
;   ErcBadJobNum will be returned if dJobNum is invalid
;   or 0 will be returned with the data.
;
; dJobNum           EQU [EBP+16]
; pJCBRet           EQU [EBP+12]

PUBLIC __GetpJCB:               ;
        PUSH EBP                ;
        MOV EBP,ESP             ;

        MOV EAX, [EBP+16]       ;Job Num
        OR EAX, EAX
        JZ GetpJcbBad           ;0 is invalid
        CMP EAX, nJCBs + 2;     ;Dynamic + 2 static
        JBE GetpJcbOK
GetpJcbBad:
        MOV EAX, ErcBadJobNum   ;
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
GetpJcbOk:
        CALL GetpJCB            ;puts address of JCB in EAX
        MOV ESI, [EBP+12]       ;pJCBRet
        MOV [ESI], EAX
        CMP DWORD PTR [EAX+JcbPD], 0        ;Is this a valid JCB
        JNE GetpJCBOk1
        MOV EAX, ErcInvalidJCB  ;JCB we are pointing to is unused
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
GetpJcbOk1:
        XOR EAX, EAX
 8       MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ;
```

```
;=============================================================
;
; GetJobNum
; This PUBLIC returns the number for the current Job. This is
; the job that the task that called this belongs to.
;
; Procedural Interface :
;
;       GetJobNum(pJobNumRet):ercType
;
;   pJCBRet is a pointer where you want the JCB returned.
;
; pJobNumRet    EQU [EBP+12]

PUBLIC __GetJobNum:             ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetCrntJobNum      ;Leave jobnum in EAX
        MOV ESI, [EBP+12]       ;pJobNumRet
        MOV [ESI], EAX          ;
        XOR EAX, EAX            ;No Error
        POP EBP                 ;
        RETF 4                  ;

;=============================================================
;
; GetSystemDisk
; This PUBLIC returns a single byte which represents the
; disk that the system booted from. This is from the public
; variable _BootDrive defined in Main.asm.
; It returns 0-n (which corresponds to drive A-x)
; This code is here for lack of a better place.
; It's really not a filesystem function either. And will
; still be needed if a loadable filesystem is installed.
;
; Procedural Interface :
;
;       GetSystemDisk(pSysDiskRet):ercType
;
;           pSysDiskRet is a pointer to a byte where
;           the number representing the system disk is returned.
;
;  pSysDiskRet  EQU [EBP+12]

PUBLIC __GetSystemDisk          ;
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EAX, _BootDrive
        AND EAX, 7Fh            ;get rid of high bits
        MOV ESI, [EBP+12]       ;pJobNumRet
        MOV [ESI], AL           ;
        XOR EAX, EAX            ;No Error
        POP EBP                 ;
        RETF 4                  ;
;================ MODULE END ================================
```

Job Management Listing

This file, Jobc.c, contains all the major high-level functions for job management. Functions for starting and ending jobs and public functions for working with the job control block are contained in this module. See Listing 22.2.

Listing 22.2. Job management code.

```
/* This file contains functions and data used to support
   loading or terminating jobs (or services).
   It contains the public functions:

   Chain()           Loads new job run file in current JCB & PD
   LoadNewJob()      Loads a new job into a new JCB & PD
   ExitJob()         Exits current job, loads ExitJob if specified

   GetExitJob()      Gets run file name that will be loaded upon ExitJob
   SetExitJob()      Sets run file name to load upon ExitJob

   SetCmdLine()      Sets the command line for next job
   GetCmdLine()      Gets the command line for the current job

   GetPath()         Gets the path prefix for the current job
   Setpath()         Sets the path prefix for the current job

   SetUserName()     Sets the Username for the current job
   GetUserName()     Gets the Username for the current job

*/


#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char
#define TRUE 1
#define FALSE 1

#include "MKernel.h"
#include "MMemory.h"
#include "MData.h"
#include "MTimer.h"
#include "MVid.h"
#include "MKbd.h"
#include "MJob.h"
#include "MFiles.h"

#include "runfile.h"

#define MEMUSERD 7          /* User Writable (Data) */

#define ErcOpCancel     4   /* Operator cancel */
#define ErcOutOfRange  10   /* Bad Exchange specified in OS call */
#define ErcBadJobNum   70   /* A Bad job number was specified */
```

```
#define ErcNoExitJob   76   /* No ExitJob specified on ExitJob(n)*/
#define ErcBadRunFile  74   /* Couldn't load specified ExitRunFile */

/* Near Support Calls from JobCode.ASM */

extern long AllocJCB(long *pdJobNumRet, char *ppJCBRet);
extern long RemoveRdyJob(char *pJCB);
extern long GetExchOwner(long Exch, char *pJCBRet);
extern long SetExchOwner(long Exch, char *pNewJCB);
extern long SendAbort(long JobNum, long Exch);

/* Temporary NEAR externals from the monitor program for debugging */

extern long xprintf(char *fmt, ...);
extern U32 Dump(unsigned char *pb, long cb);

/* We switch to this stack when we clean out a user
   PD before we rebuild it.
*/

static long TmpStack[128];      /* 512 byte temporary stack */

/* Used for allocating/filling in new JCB */

static struct JCBRec *pNewJCB;      /* Used to access a JCB */
static struct JCBRec *pTmpJCB;
static struct JCBRec *pCrntJCB;

static long JobNum;

/* For ExitJob and Chain cause They can't have local vars! */

static long JobNumE, job_fhE;
static long ExchE, ercE, iE;
static long BogusMsg[2];
static long *pPDE;
static char *pExchJCBE
static long KeyCodeE;

static char aFileE[80];
static long cbFileE;

extern unsigned long KillExch;  /* From the Monitor */

/* Run file data */

static char *pCode, *pData, *pStack;    /* Ptrs in User mem to load to */
static long sCode,   sData,  sStack;    /* Size of segments */
static unsigned long oCode, oData;      /* Offset in file to Code & Data */

static long  offCode = 0,               /* Virtual Offset for Code & Data Segs */
      offData = 0;

static unsigned long    nCDFIX = 0,
                oCDFIX = 0,
                nCCFIX = 0,
                oCCFIX = 0,
                nDDFIX = 0,
```

continues

Listing 22.2. continued

```
                oDDFIX = 0,
                nDCFIX = 0,
                oDCFIX = 0;

static char *pStart, filetype;

static struct tagtype tag;


/************ INTERNAL SUPPORT CALLS *****************/

/*****************************************************
 This deallocates all user memory in a PD.  This is
 used when we ExitJob to free up all the memory.
 We get the pointer to the PD and "walk" through all
 the User PTs deallocating linear memory. When all
 PTs have been cleaned, we eliminate the user PDEs.
 This leaves a blank PD for reuse if needed.
 NOTE: This must be called from a task that is
 running in the JCB that owns the memory!
*****************************************************/

static void CleanUserPD(long *pPD)
{
long i, j, k, erc;
unsigned long *pPT;
char *pMem;

    for (i=768; i<1024; i++) {          /* Look at each shadow PDE */
        if (pPD[i]) {                   /* If it's a linear address (non 0)*/
            pPT = pPD[i] & 0xFFFFF000;  /* Point to Page Table */
            for (j=0; j<1024; j++) {

            /* Get Beginning address for each run to deallocate */

                k = 0;  /* nPages to deallocate */
                pMem = ((i-512) * 0x400000) + (j * 4096);
                while ((pPT[j++]) && (j<1024))
                    k++;            /* one more page */
                if (k)
                {           /* we have pages (one or more) */
                    erc =  DeAllocPage(pMem, k);
                }
            }
        }
    }
}

/*****************************************************
  This opens, reads and validates the run file.
  If all is well, it leaves it open and returns the
  file handle, else it closes the run file and returns
  the error to the caller.
  The caller is responsible for closing the file!!!
*****************************************************/
```

```
static long GetRunFile(char *pFileName, long cbFileName, long *pfhRet)
{
long erc, i, fh, dret;
char fDone, junk;

    offCode = 0;
    offData = 0;
    nCDFIX = 0;
    oCDFIX = 0;
    nCCFIX = 0;
    oCCFIX = 0;
    nDDFIX = 0;
    oDDFIX = 0;
    nDCFIX = 0;
    oDCFIX = 0;

    *pfhRet = 0;    /* default to 0 */

            /* Mode Read, Stream type */
    erc = OpenFile(pFileName, cbFileName, 0, 1, &fh);

    if (!erc) { /* File opened OK */

        fDone = 0;
        while ((!erc) && (!fDone)) {
            tag.id = 0;
            erc = ReadBytes (fh, &tag, 5, &dret);

            switch (tag.id) {
                case IDTAG:
                    erc = ReadBytes (fh, &filetype, 1, &dret);
                    if ((filetype < 1) || (filetype > 3))
                        erc = ErcBadRunFile;
                    break;
                case SEGTAG:
                    erc = ReadBytes (fh, &sStack, 4, &dret);
                    if (!erc) erc = ReadBytes (fh, &sCode, 4, &dret);
                    if (!erc) erc = ReadBytes (fh, &sData, 4, &dret);
                    break;
                case DOFFTAG:
                    erc = ReadBytes (fh, &offData, 4, &dret);
                    break;
                case COFFTAG:
                    erc = ReadBytes (fh, &offCode, 4, &dret);
                    break;
                case STRTTAG:
                    erc = ReadBytes (fh, &pStart, 4, &dret);
                    break;
                case CODETAG:
                    erc = GetFileLFA(fh, &oCode);
                    if (!erc)
                        erc = SetFileLFA(fh, oCode+tag.len);    /* skip it */
                    break;
                case DATATAG:
                    erc = GetFileLFA(fh, &oData);
                    if (!erc)
                        erc = SetFileLFA(fh, oData+tag.len);    /* skip it */
```

continues

Listing 22.2. continued

```
                    break;
                case CDFIXTAG:
                    erc = GetFileLFA(fh, &oCDFIX);
                    nCDFIX = tag.len/4;
                    if (!erc)
                        erc = SetFileLFA(fh, oCDFIX+tag.len);   /* skip it */
                    break;
                case CCFIXTAG:
                    erc = GetFileLFA(fh, &oCCFIX);
                    nCCFIX = tag.len/4;
                    if (!erc)
                        erc = SetFileLFA(fh, oCCFIX+tag.len);   /* skip it */
                    break;
                case DDFIXTAG:
                    erc = GetFileLFA(fh, &oDDFIX);
                    nDDFIX = tag.len/4;
                    if (!erc)
                        erc = SetFileLFA(fh, oDDFIX+tag.len);   /* skip it */
                    break;
                case DCFIXTAG:
                    erc = GetFileLFA(fh, &oDCFIX);
                    nDCFIX = tag.len/4;
                    if (!erc)
                        erc = SetFileLFA(fh, oDCFIX+tag.len);   /* skip it */
                    break;
                case ENDTAG:
                    fDone = TRUE;
                    break;
                default:
                    erc = GetFileLFA(fh, &i);
                    if (tag.len > 1024)
                        erc = ErcBadRunFile;
                    if (!erc)
                        erc = SetFileLFA(fh, i+tag.len);    /* skip it */
                    if (erc)
                        erc = ErcBadRunFile;
                break;
            }

        }
        if (erc)
            CloseFile(fh);
    }
    if (!erc)
        *pfhRet = fh;
if (erc)
    xprintf("Erc from GetRunFile in JobC: %d\r\n", erc);

    return (erc);
}

/*******************************************************/
/********** PUBLIC CALLS FOR JOB MANAGEMENT ************/
/*******************************************************/

/***********************************************/
```

```
long far _SetExitJob(char *pRunFile, long dcbRunFile)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if (dcbRunFile > 79)
        return (ErcBadJobParam);
    else if (!dcbRunFile)
        pTmpJCB->JcbExitRF[0] = 0;
    else {
        CopyData(pRunFile, &pTmpJCB->JcbExitRF[1], dcbRunFile);
        pTmpJCB->JcbExitRF[0] = dcbRunFile;
    }
    return(0);
}

/************************************************/
long far _GetExitJob(char *pRunRet, long *pdcbRunRet)
{
long JobNum, i;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    i = pTmpJCB->JcbExitRF[0];
    if (i)
        CopyData(&pTmpJCB->JcbExitRF[1], pRunRet, i);
    *pdcbRunRet = i;
    return(0);
}

/************************************************/
long far _SetPath(char *pPath, long dcbPath)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if (dcbPath > 69)
        return (ErcBadJobParam);
    else if (!dcbPath)
        pTmpJCB->sbPath[0] = 0;
    else {
        CopyData(pPath, &pTmpJCB->sbPath[1], dcbPath);
        pTmpJCB->sbPath[0] = dcbPath;
    }
    return(0);
}

/************************************************/
long far _GetPath(long JobNum, char *pPathRet, long *pdcbPathRet)
{
long i, erc;

    erc = GetpJCB(JobNum, &pTmpJCB);  /* Get pJCB to JobNum */
    if (!erc) {
        i = pTmpJCB->sbPath[0];
```

continues

Listing 22.2. continued

```
        if (i)
            CopyData(&pTmpJCB->sbPath[1], pPathRet, i);
        *pdcbPathRet = i;
    }
    return(erc);
}
/***********************************************/
long far _SetCmdLine(char *pCmd, long dcbCmd)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if (dcbCmd > 79)
        return (ErcBadJobParam);
    else if (!dcbCmd)
        pTmpJCB->JcbCmdLine[0] = 0;
    else {
        CopyData(pCmd, &pTmpJCB->JcbCmdLine[1], dcbCmd);
        pTmpJCB->JcbCmdLine[0] = dcbCmd;
    }
    return(0);
}

/***********************************************/
long far _GetCmdLine(char *pCmdRet, long *pdcbCmdRet)
{
long JobNum, i;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    i = pTmpJCB->JcbCmdLine[0];
    if (i)
        CopyData(&pTmpJCB->JcbCmdLine[1], pCmdRet, i);
    *pdcbCmdRet = i;
    return(0);
}


/***********************************************/
long far _SetUserName(char *pUser, long dcbUser)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if (dcbUser > 29)
        return (ErcBadJobParam);
    else if (!dcbUser)
        pTmpJCB->sbUserName[0] = 0;
    else {
        CopyData(pUser, &pTmpJCB->sbUserName[1], dcbUser);
        pTmpJCB->sbUserName[0] = dcbUser;
    }
    return(0);
}
```

```
/************************************************/
long far _GetUserName(char *pUserRet, long *pdcbUserRet)
{
long JobNum, i;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    i = pTmpJCB->sbUserName[0];
    if (i)
        CopyData(&pTmpJCB->sbUserName[1], pUserRet, i);
    *pdcbUserRet = i;
    return(0);
}

/************************************************/
long far _SetSysIn(char *pName, long dcbName)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if ((dcbName > 49) || (!dcbName))
        return (ErcBadJobParam);
    else {
        CopyData(pName, &pTmpJCB->JcbSysIn[1], dcbName);
        pTmpJCB->JcbSysIn[0] = dcbName;
    }
    return(0);
}

/************************************************/
long far _GetSysIn(char *pFileRet, long *pdcbFileRet)
{
long JobNum, i;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    i = pTmpJCB->JcbSysIn[0];
    if (i)
        CopyData(&pTmpJCB->JcbSysIn[1], pFileRet, i);
    *pdcbFileRet = i;
    return(0);
}

/************************************************/
long far _SetSysOut(char *pName, long dcbName)
{
long JobNum;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if ((dcbName > 49) || (!dcbName))
        return (ErcBadJobParam);
    else {
        CopyData(pName, &pTmpJCB->JcbSysOut[1], dcbName);
        pTmpJCB->JcbSysOut[0] = dcbName;
```

continues

Listing 22.2. continued

```
    }
    return(0);
}

/***********************************************/
long far _GetSysOut(char *pFileRet, long *pdcbFileRet)
{
long JobNum, i;

    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    i = pTmpJCB->JcbSysOut[0];
    if (i)
        CopyData(&pTmpJCB->JcbSysOut[1], pFileRet, i);
    *pdcbFileRet = i;
    return(0);
}

/***********************************************/
long far _SetJobName(char *pName, long dcbName)
{
long JobNum;
    GetJobNum(&JobNum);
    GetpJCB(JobNum, &pTmpJCB);      /* Get pJCB to current Job */
    if (dcbName > 13)
        dcbName = 13;
    if (dcbName)
        CopyData(pName, &pTmpJCB->sbJobName[1], dcbName);
    pTmpJCB->sbJobName[0] = dcbName;
    return(0);
}

/*******************************************************
  This creates and loads a new Job from a RUN file.
  This returns ErcOK and new job number if loaded OK
  else an error is returned.
*******************************************************/

long far _LoadNewJob(char *pFileName, long cbFileName, long *pJobNumRet)
{
long erc, i, fh, dret, nPages;
unsigned long *pPD, *pPT, *pVid, *pOSPD;
long *pFix;
U32 PhyAdd;


    erc =  GetRunFile(pFileName, cbFileName, &fh);

    if (!erc) {

        /* We set these to zero so we can tell if they have
        been allocated in case we fail so we know what to deallocate
        */

        JobNum = 0;
        pNewJCB = 0;
```

```
pPD = 0;
pPT = 0;
pVid = 0;

erc = AllocJCB(&JobNum, &pNewJCB);

/* Alloc OS memory pages required for new job */

if (!erc)
    erc = AllocOSPage(3, &pPD);     /* Job's PD, PT * pVirtVid */

pPT = pPD + 4096;           /* 1 page later */
pVid = pPD + 8192;          /* 2 pages later */

if (!erc) {
    FillData(pPT, 4096, 0);     /* Zero user PT */
    FillData(pVid, 4096, 0);    /* Zero user video */

    GetpJCB(1, &pTmpJCB);       /* Get OS pJCB */
    pOSPD = pTmpJCB->pJcbPD;    /* Pointer to OS PD */

    GetPhyAdd(1, pPT, &PhyAdd); /* Get Phy Add for new PT */

    PhyAdd |= MEMUSERD;         /* Set user code bits in PDE */

    pOSPD[256] = PhyAdd;        /* Make User PDE in OS PD */
    pOSPD[768] = pPT;           /* Shadow Linear Address of PT */

    /*
    xprintf("pOSPD  : %08x\r\n", pOSPD);
    xprintf("pUserPD: %08x\r\n", pPD);
    xprintf("pUserPT: %08x\r\n", pPT);
    xprintf("PhyAdd : %08x\r\n", PhyAdd);
    ReadKbd(&KeyCodeE, 1);
    */

    /* Now we can allocate User Job Memory */
    /* Allocate user memory for Stack Code and Data */
    /* This is STILL done in the OS PD */

    nPages = sStack/4096;
    if (sStack%4096) nPages++;
    erc = AllocPage(nPages, &pStack);
    sStack = nPages * 4096;             /* set to whole pages */

    nPages = sCode/4096;
    if (sCode%4096) nPages++;
    if (!erc)
        erc = AllocPage(nPages, &pCode);

    nPages = sData/4096;
    if (sData%4096) nPages++;
    if (!erc)
        erc = AllocPage(nPages, &pData);

    /* Right now, the OS PD looks exacly like we want
    the User PD to look.  We will now copy the entire
    OS PD into the User's New PD.
```

continues

Listing 22.2. continued

```
*/

CopyData(pOSPD, pPD, 4096);     /* Copy OS PD to User PD */

/* All Job memory is now allocated, so let's LOAD IT! */

if (!erc)
    erc = SetFileLFA(fh, oCode);
if (!erc)
    erc = ReadBytes (fh, pCode, sCode, &dret);

if (!erc)
    erc = SetFileLFA(fh, oData);
if (!erc)
    erc = ReadBytes (fh, pData, sData, &dret);

/* Now that we have read in the code and data we
apply fixups to these segments from the runfile
*/

if (!erc) {

    if (nCDFIX) {
        erc = SetFileLFA(fh, oCDFIX);   /* back to fixups */
        while ((nCDFIX--) && (!erc)) {
            erc = ReadBytes (fh, &i, 4, &dret);
             pFix = pCode + i;          /* Where in CSeg */
            *pFix = *pFix - offData + pData;
        }
    }

    if (nCCFIX) {
        erc = SetFileLFA(fh, oCCFIX);   /* back to fixups */
        while ((nCCFIX--) && (!erc)) {
            erc = ReadBytes (fh, &i, 4, &dret);
            pFix = pCode + i;          /* Where in CSeg */
            *pFix = *pFix - offCode + pCode;
        }
    }

    if (nDCFIX) {
        erc = SetFileLFA(fh, oDCFIX);   /* back to fixups */
        while ((nDCFIX--) && (!erc)) {
            erc = ReadBytes (fh, &i, 4, &dret);
            pFix = pData + i;          /* Where in DSeg */
            *pFix = *pFix - offCode + pCode;
        }
    }

    if (nDDFIX) {
        erc = SetFileLFA(fh, oDDFIX);   /* back to fixups */
        while ((nDDFIX--) && (!erc)) {
            erc = ReadBytes (fh, &i, 4, &dret);
            pFix = pData + i;          /* Where in DSeg */
            *pFix = *pFix - offData + pData;
        }
```

```
        }

    }

    /* Clean the OS PD of User memory */

    FillData(&pOSPD[256], 1024, 0); /* Clean OS PD of User PDEs */
    FillData(&pOSPD[768], 1024, 0); /* Clean OS PD of User Shadow */

    /* Now we fill in the rest of the User's JCB */

    pNewJCB->pJcbPD = pPD;          /* Lin Add of PD */
    pNewJCB->pJcbStack = pStack;    /* Add of Stack */
    pNewJCB->sJcbStack = sStack;    /* Size of Code */
    pNewJCB->pJcbCode  = pCode;     /* Add of Code */
    pNewJCB->sJcbCode  = sCode;     /* Size of Code */
    pNewJCB->pJcbData  = pData;     /* Add of Code */
    pNewJCB->sJcbData  = sData;     /* Size of Code */

    pNewJCB->sbUserName[0] = 0;     /* Zero UserName */
    pNewJCB->sbPath[0] = 0;         /* No Default path */
    pNewJCB->JcbExitRF[0] = 0;      /* No Exit Run File */
    pNewJCB->JcbCmdLine[0] = 0;     /* No Cmd Line */

    CopyData("KBD", &pNewJCB->JcbSysIn[1], 3);
    pNewJCB->JcbSysIn[0] = 3;       /* Size */

    CopyData("VID", &pNewJCB->JcbSysOut[1], 3);
    pNewJCB->JcbSysOut[0] = 3;      /* Size */

    pNewJCB->pVidMem = pVid;        /* Default to Virt Vid */
    pNewJCB->pVirtVid = pVid;       /* Virtual Video memory */
    pNewJCB->CrntX = 0;             /* Vid X Position */
    pNewJCB->CrntY = 0;             /* Vid Y Position */
    pNewJCB->nCols = 80;            /* Columns */
    pNewJCB->nLines = 25;           /* Lines */

    pNewJCB->VidMode = 0;           /* 80x25 VGA Color Text */
    pNewJCB->fCursOn = 1;           /* Cursor On */
    pNewJCB->fCursType = 0;         /* UnderLine */


    /* Finally, we crank up the new task and schedule it for
    execution!
    */
    if (!erc)
        erc = AllocExch(&i);

    if (!erc)
        erc = NewTask(JobNum, 0x18, 25, 0, i,
                      pStack+sStack-4,
                      pStart+pCode-offCode);
    if (!erc)
        SetExchOwner(i, pNewJCB);   /* Exch now belongs to new JCB */
}

CloseFile(fh);
}   /* read run file OK */
```

continues

Listing 22.2. continued

```
    if (!erc)
        *pJobNumRet = JobNum;

    return(erc);
}

/*********************************************************
  This loads a job into an existing PD and JCB.  This is
  called by Chain() and may also be called by ExitJob()
  if an ExitJob was specified in the current JCB.
  The PD, pVid & first PT still exist in OS memory.
  (CleanPD left the first PT for us). ExitJob and Chain
  are responsible for opening and validating the runfile
  and setting up the run file variables.
*********************************************************/

static long LoadJob(char *pJCB, long fh)
{
long erc, i, dret, nPages;
long *pFix;

    pNewJCB = pJCB;

    /* Allocate user memory for Stack, Code and Data */
    /* This is done in the context of the USER PD */

    nPages = sStack/4096;
    if (sStack%4096) nPages++;
    erc = AllocPage(nPages, &pStack);
    sStack = nPages * 4096;             /* set to whole pages */

    nPages = sCode/4096;
    if (sCode%4096) nPages++;
    if (!erc)
        erc = AllocPage(nPages, &pCode);

    nPages = sData/4096;
    if (sData%4096) nPages++;

    erc = AllocPage(nPages, &pData);

    /* All Job memory is now allocated, so let's LOAD IT! */

    if (!erc)
        erc = SetFileLFA(fh, oCode);
    if (!erc)
        erc = ReadBytes (fh, pCode, sCode, &dret);

    if (!erc)
        erc = SetFileLFA(fh, oData);
    if (!erc)
        erc = ReadBytes (fh, pData, sData, &dret);

    /* Now that we have read in the code and data we
    apply fixups to these segments from the runfile
```

```
    */

    if (!erc) {
        if (nCDFIX) {
            erc = SetFileLFA(fh, oCDFIX);   /* back to fixups */
            while ((nCDFIX--) && (!erc)) {
                erc = ReadBytes (fh, &i, 4, &dret);
                pFix = pCode + i;           /* Where in CSeg */
                *pFix = *pFix - offData + pData;
            }
        }

        if (nCCFIX) {
            erc = SetFileLFA(fh, oCCFIX);   /* back to fixups */
            while ((nCCFIX--) && (!erc)) {
                erc = ReadBytes (fh, &i, 4, &dret);
                pFix = pCode + i;           /* Where in CSeg */
                *pFix = *pFix - offCode + pCode;
            }
        }

        if (nDCFIX) {
            erc = SetFileLFA(fh, oDCFIX);   /* back to fixups */
            while ((nDCFIX--) && (!erc)) {
                erc = ReadBytes (fh, &i, 4, &dret);
                pFix = pData + i;           /* Where in DSeg */
                *pFix = *pFix - offCode + pCode;
            }
        }

        if (nDDFIX) {
            erc = SetFileLFA(fh, oDDFIX);   /* back to fixups */
            while ((nDDFIX--) && (!erc)) {
                erc = ReadBytes (fh, &i, 4, &dret);
                pFix = pData + i;           /* Where in DSeg */
                *pFix = *pFix - offData + pData;
            }
        }

        /* Now we fill in the rest of the User's JCB */
        pNewJCB->pJcbStack = pStack;    /* Add of Stack */
        pNewJCB->sJcbStack = sStack;    /* Size of Code */
        pNewJCB->pJcbCode  = pCode;     /* Add of Code */
        pNewJCB->sJcbCode  = sCode;     /* Size of Code */
        pNewJCB->pJcbData  = pData;     /* Add of Code */
        pNewJCB->sJcbData  = sData;     /* Size of Code */

    }
    CloseFile(fh);
    return(erc);
}


/*****************************************************
 This is started as a new task in the context of a
 job that is being killed off. This is done to allow
 memory access and also reuse the code for ExitJob
 in the monitor. This task, its exchange, and TSS
```

continues

Listing 22.2. continued

```
 will be reclaimed by the monitor along with the
 JCB and all OS memory pages for the PD,PT and video.
*****************************************************/

void _KillTask(void)
{

    GetJobNum(&JobNumE);
    GetpJCB(JobNumE, &pTmpJCB);     /* Get pJCB to this Job */

    /* Clean the PD of all user memory leaving OS memory to be
       deallocated by the caller (monitor or whoever).
    */

    pPDE = pTmpJCB->pJcbPD;
    CleanUserPD(pPDE);

    GetTSSExch(&ExchE);

    ercE = 0;
    while(!ercE)            /* clear the exchange */
        ercE = CheckMsg(ExchE, BogusMsg);

    ISendMsg(KillExch, ExchE, ErcOpCancel);
    SetPriority(31);
    WaitMsg(ExchE, BogusMsg);
    White(1);               /*In case we get scheduled again*/
    /* He's History! */
}


/*****************************************************
 This is called from one job to kill another job or
 service. This cleans up ALL resources that the
 job had allocated.
 This is used to kill run-away jobs, or terminate a
 job just for the fun of it.
 It results in a violent death for the job specified.
 This must never be called from a task within the job
 to be killed.  A job may terminate itself with ExitJob().
*****************************************************/

long far _KillJob(long JobNum)
{
long erc;
    /* Make sure it's not the Monitor, Debugger or the current job. */

    GetJobNum(&JobNumE);
    if ((JobNum == JobNumE) ||
        (JobNum == 1) ||
        (JobNum == 2))

        return(ErcBadJobNum);

    erc = GetpJCB(JobNum, &pTmpJCB);            /* Get pJCB to the Job */
    if (erc)
```

```
        return(erc);

    pTmpJCB->ExitError = ErcOpCancel; /* Operator said DIE! */

    /* Remove ALL tasks for this job that are at the ReadyQue.
       The task we are in does not belong to the job we are
       killing so we can remove them all.
    */

    RemoveRdyJob(pTmpJCB);  /* It always returns ErcOk */

    /* Deallocate ALL exchanges for this job */

     ercE = 0;
     iE = 0;
     while (ercE != ErcOutOfRange)
     {
        ercE = GetExchOwner(iE, &pExchJCBE);
        if ((!ercE) && (pExchJCBE == pTmpJCB))
            DeAllocExch(iE);
        iE++;
     }
     ercE = 0;       /* Clear the error */

    /* Now that the user can't make anymore requests,
       We send "Abort" messages to all services.
       This closes all files that were opened by the Job
       and frees up any other resources held for this
       job by any service.

       We will allocate one exchange for the job
       that is being killed so we can use it for SendAbort
       and also as the exchange number we send to the
       KillExch in the monitor which will kill of the JCB
       completely (he also switches video and keyboard
       if needed).
     */

    erc = AllocExch(&ExchE);        /* Get an Exch */
    SetExchOwner(ExchE, pTmpJCB);   /* make him the owner */
    SendAbort(JobNum, ExchE);   /* Notify all services */

        /*JobNum, CodeSeg, Priority, fDebug, Exch, ESP, EIP */
    erc = NewTask(JobNum, 0x08, 3, 0, ExchE, &TmpStack[127], &_KillTask);
    return(erc);

    /* He's History! */
}

/*****************************************************
 This is called from Exit() in C or directly from a user
 job or service. This cleans up ALL resources that the
 job had allocated.
 This also checks for an exit run file to load if
 one is specified. If no exit run file is specified
 we just kill the JCB entirely and if video and
 keyboard are assigned we assign them to the Monitor.
*****************************************************/
```

continues

Listing 22.2. continued

```
void far _ExitJob(long dError)
{
/* NO LOCAL VARIABLES BECAUSE WE SWITCH STACKS!! */

    GetJobNum(&JobNumE);
    GetpJCB(JobNumE, &pCrntJCB);        /* Get pJCB to current Job */
    pCrntJCB->ExitError = dError;

    /* Remove ALL tasks for this job that are at the ReadyQue.
       The task we are in won't be removed because it's RUNNING!
    */

    RemoveRdyJob(pCrntJCB); /* It always returns ErcOk */

    /* Deallocate all exchanges for this job except the one belonging
       to current TSS!  The Dealloc Exchange call will invalidate
       all TSSs found at exchanges belonging to this user, and
       will also free up RQBs and Link Blocks.  The job will not be
       able to initiate requests or send messages after this unless
       it is done with the TSSExchange because it will get a kernel
       error (invalid exchange).
    */

     /* Find out what our TSS exchange is so
     we don't deallocate it too! */

     GetTSSExch(&ExchE);

     ercE = 0;
     iE = 0;
     while (ercE != ErcOutOfRange) {
        ercE = GetExchOwner(iE, &pExchJCBE);
        if ((!ercE) && (iE != ExchE) && (pExchJCBE == pCrntJCB))
            DeAllocExch(iE);
        iE++;
     }

    /* Now that the user can't make anymore requests,
       Send Abort messages to all services.
       This closes all files that were opened by the Job
       and frees up any other resources held for this
       job by any service.
    */

    SendAbort(JobNumE, ExchE);
    ercE = 0;               /* Clear the error */
    while(!ercE)            /* clear the exchange */
        ercE = CheckMsg(ExchE, BogusMsg);
    ercE = 0;               /* Clear the error */

    /* We must now switch to a temporary stack so we can
    clean out the user PD (we are on his stack right now!).
    */

#asm
    MOV EAX, OFFSET _TmpStack
```

```
    ADD EAX, 508
    MOV ESP, EAX
    MOV EBP, EAX
#endasm

    /* Clean the PD of all user memory leaving OS memory for next
    job if there is one.
    */

    pPDE = pCrntJCB->pJcbPD;
    CleanUserPD(pPDE);

    /* Look for Exit Run file to load if any exists.  If no exit run
    file, we deallocate the PD and JCB then return to JOB 1. */

    GetExitJob(aFileE, &cbFileE);       /* Exit Run File!! */

    if (!cbFileE)
        ercE = ErcNoExitJob;

    if (!ercE)
        ercE =  GetRunFile(aFileE, cbFileE, &job_fhE);

    if (!ercE)
        ercE = LoadJob(pCrntJCB, job_fhE);

    if (!ercE) {

            pStart = pStart+pCode-offCode;

            /* Now we RETURN to new job's address after we put him
            on his new stack. */

#asm
            MOV EAX, _pStack
            MOV EBX, _sStack
            ADD EAX, EBX
            SUB EAX, 4
            MOV ESP, EAX
            MOV EBP, EAX
            PUSH 18h
            MOV EAX, _pStart
            PUSH EAX
            RETF                ;We are history!
#endasm

    }

    if (ercE) {     /* something failed or we don't have an ExitRF */

        /* In case there is no job to run or a fatal error has happened
        we send a message (ISendMsg) to the monitor status
        task with our TSSExch and the Error. Then he will WIPE US OUT!
        We use ISend (vice send) so he can't run before we get to
        the exchange otherwise we will be placed back on the readyQueue!
        */

        ISendMsg(KillExch, ExchE, ercE);
```

continues

Listing 22.2. continued

```
        SetPriority(31);
        WaitMsg(ExchE, BogusMsg);
        while(1);                    /*In case we get scheduled again*/
        /* We are NO MORE */
    }
}

/*****************************************************
 This is called to execute a program without changing
 the ExitJob. This is so you can run program B from
 program A and return to program A when program B is
 done.  This runs Job B in the "context" of Job A
 which means Job B inherits the JCB and PD of job A
 so it can use things like the command line and
 path that were set up by A.
 Information can be passed to Job B by calling
 SetCmdLine (if Job B reads it), and also by
 setting the ExitError value in the parameter.
 Chain will only return to you if there was an
 error loading the Job.  In other words, if Chain
 fails in a critial section we try to load the
 ExitJob and pass it the error.
*****************************************************/

long far _Chain(char *pFileName, long cbFileName, long dExitError)
{
/* NO LOCAL VARIABLES BECAUSE WE SWITCH STACKS!! */

    CopyData(pFileName, aFileE, cbFileName);
    cbFileE = cbFileName;

    ercE =  GetRunFile(pFileName, cbFileName, &job_fhE);
    if (ercE)
    {
        CloseFile(job_fhE); /* if it had a handle at all */
        return(ercE);
    }

    CloseFile(job_fhE);     /* we will open it again after SendAbort */

2   GetJobNum(&JobNumE);
    GetpJCB(JobNumE, &pCrntJCB);        /* Get pJCB to current Job */
    pCrntJCB->ExitError = dExitError;

    /* Remove ALL tasks for this job that are at the ReadyQue.
       The task we are in won't be removed because it's RUNNING!
    */

    RemoveRdyJob(pCrntJCB); /* It always returns ErcOk */

    /* Deallocate all exchanges for this job except the one belonging
       to current TSS!  The Dealloc Exchange call will invalidate
       all TSSs found at exchanges belonging to this user, and
       will also free up RQBs and Link Blocks.  The job will not be
       able to initiate requests or send messages after this unless
       it is done with the TSSExchange because it will get a kernel
```

```
      error (invalid exchange).
   */

    /* Find out what our TSS exchange is so
    we don't deallocate it too! */

    GetTSSExch(&ExchE);

    ercE = 0;
    iE = 0;
    while (ercE != ErcOutOfRange) {
       ercE = GetExchOwner(iE, &pExchJCBE);
       if ((!ercE) && (iE != ExchE) && (pExchJCBE == pCrntJCB))
           DeAllocExch(iE);
       iE++;
    }

   /* Now that the user can't make anymore requests,
      Send Abort messages to all services.
      This closes all files that were opened by the Job
      and frees up any other resources held for this
      job by any services.
   */

   SendAbort(JobNumE, ExchE);

   ercE = 0;               /* Clear the error */
   while(!ercE)            /* clear the exchange of abort responses*/
       ercE = CheckMsg(ExchE, BogusMsg);
   ercE = 0;               /* Clear the error */

   /* We must now switch to a temporary stack so we can
   clean out the user PD (we are on his stack right now!).
   */

#asm
   MOV EAX, OFFSET _TmpStack
   ADD EAX, 508
   MOV ESP, EAX
   MOV EBP, EAX
#endasm

   /* Clean the PD of all user memory leaving OS memory for next
   job if there is one.
   */

   pPDE = pCrntJCB->pJcbPD;
   CleanUserPD(pPDE);

   /* Try to load the Chain file. Don't bother checking error
      cause it was valid if we got here!
   */

   GetRunFile(aFileE, cbFileE, &job_fhE); /* it was valid before!*/
   ercE = LoadJob(pCrntJCB, job_fhE);

   if (ercE) {
```

continues

Listing 22.2. continued

```
        /* We have errored in a critical part of Chain (LoadJob).
        The original user's job is destroyed, and we can't run the
        chain file (bummer).
        The only thing left to do is Look for Exit Run file to load
        if any exists.  If no exit run file, we kill this guy
        and return to the monitor if he had the screen. */

        GetExitJob(aFileE, &cbFileE);       /* Exit Run File!! */

        if (!cbFileE)
            ercE = ErcNoExitJob;

        if (!ercE)
            ercE =  GetRunFile(aFileE, cbFileE, &job_fhE);

        if (!ercE)
            ercE = LoadJob(pCrntJCB, job_fhE);
    }

    if (!ercE) {        /* No error */

        pStart = pStart+pCode-offCode;

        /* Now we RETURN to new job's address after we put him
        on his new stack. */

#asm
        MOV EAX, _pStack
        MOV EBX, _sStack
        ADD EAX, EBX
        SUB EAX, 4
        MOV ESP, EAX
        MOV EBP, EAX
        PUSH 18h
        MOV EAX, _pStart
        PUSH EAX
        RETF                ;We are history!
#endasm
    }

    if (ercE) {     /* Something failed loading the job (Chain or Exit) */

        /* In case there is no job to run or a fatal error has happened
        we send a message (ISendMsg) to the monitor status
        task with our TSSExch and the Error. Then he will WIPE US OUT!
        We use ISend (vice send) so he can't run before we get to
        the exchange otherwise we will be placed back on the readyQueue!
        */

        ISendMsg(KillExch, ExchE, ercE);  /* ISend clears ints! */
#asm
        STI
#endasm
        SetPriority(31);
```

```
        WaitMsg(ExchE, BogusMsg);
        while(1);                       /*In case we get scheduled again*/
        /* We are NO MORE */

    }
}

/********************** End of Module ****************/
```

23

Debugger Code

The debugger initially began as a crude memory dumper built into the operating system. It has grown substantially, but is still very immature. While I wrote the operating system, small pieces of the debugger were added as they were needed.

The debugger is a separate task, but it is not entered directly from the breakpoint address as an interrupt task. Breakpoints, which are all debug exceptions, are set up to execute an interrupt procedure that does some rather tricky manipulation of the debugger task state segment (TSS). The two major things that are changed are the pointer to the JCB and the CR3 register in the debugger's TSS. They are made to match those of the interrupted task. This allows the debugger to operate in the memory context of the task that was interrupted.

Debugger Interrupt Procedure

This code excerpt is taken from the file EXCEPT.ASM, which contains handlers for all system exceptions. You will note that even though the two exception procedures are the same, I have left them as individual procedures. I did this because on occasion I had to modify one individually, and I ended up duplicating them several times after combining them. See Listing 23.1.

Listing 23.1. Exception handlers for entering the debugger.

```
;==================== Debugger Single Step (Int 1) ======================

PUBLIC IntDbgSS:

        POP dbgOldEIP           ; Get EIP of offender for debugger
        POP dbgOldCS            ;
        POP dbgOldEFlgs         ;
        JMP EnterDebug          ;Enter debugger

;==================== Debugger Entry (Int 3) ============================

PUBLIC IntDebug:

        POP dbgOldEIP           ; Get EIP of offender for debugger
        POP dbgOldCS            ; Get CS
        POP dbgOldEFlgs         ; Get Flags
        JMP EnterDebug          ;Enter debugger


;EnterDebug
;
; This piece of code sets up to enter the debugger.  If we get here,
; one of the exceptions has activated and has done a little work based
; on which exception it was, then it jumped here to enter the debugger.
; This code effectively replaces the interrupted task with the debugger
; task (without going through the kernel).  First we copy the Page Dir
; entry from the current job into the debuggers job, then copy the CR3
; register from the current TSS into the debugger's TSS.  This makes the
```

```
; debugger operate in the current task's memory space.  All of the debugger's
; code and data are in OS protected pages (which are shared with all tasks),
; so this is OK to do even if the offending task referenced a bogus address.
; Next, we save the current pRunTSS and place the debugger's TSS in
; pRunTSS, then jump to the debugger's selector.  This switches tasks.
;
EnterDebug:
        PUSH EAX                    ;we MUST save caller's registers
        PUSH EBX                    ; and restore them before the
        PUSH EDX                    ; task switch into the debugger

        MOV EAX, pRunTSS            ;pRunTSS -> EAX
        MOV EBX, [EAX+TSS_CR3]      ;current CR3 -> EBX
        MOV EDX, OFFSET DbgTSS      ;pDebuggerTSS -> EDX
        MOV [EDX+TSS_CR3], EBX      ;CR3 -> DebuggerTSS

        MOV EAX, [EAX+TSS_pJCB]     ;pCrntJCB -> EAX
        MOV EDX, [EDX+TSS_pJCB]     ;pDebuggerJCB -> EDX
        MOV EBX, [EAX+JcbPD]        ;CrntJob Page Dir -> EBX
        MOV [EDX+JcbPD], EBX        ;Page Dir -> Debugger JCB

        MOV EAX, pRunTSS            ;Save the current pRunTSS
        MOV DbgpTSSSave, EAX
        MOV EAX, OFFSET DbgTSS      ;Install Debugger's as current
        MOV pRunTSS, EAX            ;Set Dbgr as running task

        MOV BX, [EAX+Tid]
        MOV TSS_Sel, BX             ;Set up debugger selector

        POP EDX                     ;make his registers right!
        POP EBX
        POP EAX

        JMP FWORD PTR [TSS]         ;Switch tasks to debugger

        ;When the debugger exits, we come here

        PUSH dbgOldEFlgs            ;Put the stack back the way it was
        PUSH dbgOldCS               ;
        PUSH dbgOldEIP              ;
        IRETD                       ;Go back to the caller
```

Debugger Source Listing

Listing 23.2 presents the debugger code in its entirety, with the exception of the disassembler. You'll see that much of it is display "grunt work."

The debugger also has its own read-keyboard call in the keyboard service to allow debugging of portions of the keyboard code.

Listing 23.2. Debugger data and code.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE TSS.INC
.INCLUDE RQB.INC
.INCLUDE JOB.INC

.ALIGN DWORD

;External near Variables

EXTRN rgTmrBlks DD
EXTRN pJCBs     DD
EXTRN RdyQ      DD
EXTRN rgPAM     DB

;debugger variables and buffers

PUBLIC DbgpTSSSave  DD 0        ;TO save the TSS we interrupted
PUBLIC dbgFAULT     DD 0FFh     ;0FFh is NO FAULT
PUBLIC dbgFltErc    DD 0
PUBLIC dbgOldEIP    DD 0
PUBLIC dbgOldCS     DD 0
PUBLIC dbgOldEflgs  DD 0

DbgVidSave  DD 0            ;To save interrupted video user
dbgBuf      DB '00000000'   ;General use buffers
dbgBuf2     DB '0000000000'

.ALIGN DWORD
cbBufLen2   DD 0            ;Active bytes in buf2
dbgKeyCode  DD 0            ;For ReadDbgKbd
NextEIP     DD 0            ;For Disassem display (next ins to be executed)
dbgGPdd1    DD 0            ;General purpose DDs (used all over)
dbgGPdd2    DD 0            ;
dbgGPdd3    DD 0            ;
dbgGPdd4    DD 0            ;
dbgGPdd5    DD 0            ;
dbgGPdd6    DD 0            ;flag 1 = Msg, 0 = Exch

dbgNextAdd  DD 0            ;Address we are setting as next
dbgCrntAdd  DD 0            ;Address of instructions we are displaying
dbgDumpAdd  DD 0            ;Address we are dumping

dbgX        DD 0            ;For line and column display coordination
dbgY        DD 0

dbgBPAdd    DD 0            ;Crnt Breakpoint Linear addresses

dbgfDumpD   DB 0            ;Boolean- are we dumping DWORDS?
fDbgInit    DB 0            ;Has Dbg Video been initialized?

dbgCRLF     DB 0Dh, 0Ah     ;CR LF for Dump etc...
dbgChar     DB 0

dbgMenu     DB 'SStep',0B3h,'SetBP',0B3h,'ClrBP',0B3h,'CS:EIP  '
```

```
            DB 'Exchs',0B3h,'Tasks',0B3h,'     ',0B3h,'CrntAddr'
            DB 'DumpB',0B3h,'DumpD',0B3h,'     ',0B3h,'AddInfo '
dbgSpace    DB '       '
dbgCont     DB 'ENTER to continue, ESC to quit.'
dbgClear    DB '                                          '  ;40 bytes
dbgAsterisk DB 42

;               01234567890123456789012345678901234567890123456789012345678901234
dbgExchMsg  DB 'Exch      Owner     dMsgLo    dMsgHi              Task'

;               01234567890123456789012345678901234567890123456789012345678901234
dbgTaskMsg  DB 'Task#     Job       pJCB      pTSS      Priority      '

;For Debugger entry conditions
dbgFltMsg   DB 'FATAL Processor Exception/Fault: '
sdbgFltMsg  DD $-dbgFltMsg

    ;Register display text

dbgTxt00    DB 0B3h,'TSS: '
dbgTxt01    DB 0B3h,'EAX: '
dbgTxt02    DB 0B3h,'EBX: '
dbgTxt03    DB 0B3h,'ECX: '
dbgTxt04    DB 0B3h,'EDX: '
dbgTxt05    DB 0B3h,'ESI: '
dbgTxt06    DB 0B3h,'EDI: '
dbgTxt07    DB 0B3h,'EBP: '
dbgTxt08    DB 0B3h,' SS: '
dbgTxt09    DB 0B3h,'ESP: '
dbgTxt10    DB 0B3h,' CS: '
dbgTxt11    DB 0B3h,'EIP: '
dbgTxt12    DB 0B3h,' DS: '
dbgTxt13    DB 0B3h,' ES: '
dbgTxt14    DB 0B3h,' FS: '
dbgTxt15    DB 0B3h,' GS: '
dbgTxt16    DB 0B3h,'EFL: '
dbgTxt17    DB 0B3h,'CR0: '
dbgTxt18    DB 0B3h,'CR2: '
dbgTxt19    DB 0B3h,'CR3: '
dbgTxt20    DB 0B3h,'Erc: '

dbgTxtAddr  DB 'Linear address: '

    ;For Important Address Info Display

dbgM0       DB 'IDT:  '
dbgM1       DB 'GDT:  '
dbgM2       DB 'RQBs: '
dbgM3       DB 'TSS1: '
dbgM4       DB 'TSS3: '
dbgM5       DB 'LBs:  '
dbgM6       DB 'RdyQ: '
dbgM7       DB 'JCBs: '
dbgM8       DB 'SVCs: '
dbgM9       DB 'Exch: '
dbgPA       DB 'PAM:  '
dbgMB       DB 'aTmr: '
```

continues

Listing 23.2. continued

```
;==================== End Data, Begin Code ==========================

.CODE

EXTRN DDtoHex NEAR
EXTRN HexToDD NEAR
EXTRN _disassemble NEAR
EXTRN ReadDbgKbd NEAR

PUBLIC DbgTask:

        MOV EAX, OFFSET DbgVidSave  ;Save number of vid we interrupted
        PUSH EAX
        CALL FWORD PTR _GetVidOwner

        STI

        PUSH 2
        CALL FWORD PTR _SetVidOwner     ;Dbgr is Job 2

        CMP fDbgInit, 0
        JNE DbgInitDone
        CALL FWORD PTR _ClrScr
        MOV fDbgInit, 1

DbgInitDone:

        MOV EAX, DbgpTSSSave

        ;When a fault or debug exception occurs, the values of
        ;the Instruction Pointer, Code Seg, and flags are not the
        ;way they were when the exception fired off because of the
        ;interrupt procedure they entered to get to the debugger.
        ;We make them the same by putting the values we got from
        ;the stack (entering the debugger) into the caller's TSS.
        ;
        MOV EBX,dbgOldEflgs         ;Store correct flags
        MOV [EAX+TSS_EFlags],EBX    ;EAX still has DbgpTSSSave
        MOV EBX,dbgOldCS            ;Store correct CS
        MOV [EAX+TSS_CS],BX
        MOV EBX,dbgOldEIP           ;Store correct EIP
        MOV [EAX+TSS_EIP],EBX
        ;
        ;NOTE: The "book" says the TF flag is reset by the processor
        ; when the handler is entered. This only applies if
        ; the handler is a procedure (NOT a task).  The debugger
        ; is always entered as a procedure, (we change the tasks)
        ; so we shouldn't have to reset it.  But we do...
        ; I guess I'm not reading it right or ROD SERLING LIVES!
        ;
        MOV EBX,[EAX+TSS_EFlags]    ;Reset TF in case single stepping
        AND EBX,0FFFFFEFFh
        MOV [EAX+TSS_EFlags],EBX

        ;We set the FAULT variable based on which interrupt
        ;procedure was entered.
```

```
        CMP DWORD PTR dbgFAULT,0FFh ;Was the dbgr entered on a FAULT?
        JE dbg000                   ;NO
        ;
        ;NOTE: Must eventually add SS/ESP for a change in CPL on faults!!!
        ;REF - See page 3-4 System Software Writer's Guide

        XOR EAX, EAX
        PUSH EAX                    ;Display fault message and
        PUSH EAX                    ;   number at 0 ,0
        CALL FWORD PTR _SetXY

        LEA EAX,dbgFltMsg
        PUSH EAX
        PUSH sdbgFltMsg
        PUSH 40h                    ;Color Black on RED
        CALL FWORD PTR _TTYOut

        MOV EAX,dbgFAULT
        PUSH EAX
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        LEA EAX,dbgBuf
        PUSH EAX
        PUSH 8
        PUSH 70h
        CALL FWORD PTR _TTYOut
        MOV DWORD PTR dbgFAULT, 0FFh                ;reset fault indicator

        LEA EAX,dbgCRLF
        PUSH EAX
        PUSH 2
        PUSH 07h                    ;Color White on black
        CALL FWORD PTR _TTYOut
        XOR EAX, EAX
        MOV dbgX, EAX               ;Reset X & Y to 0,1
        INC EAX
        MOV dbgY, EAX

;================================================================
dbg000:
        CALL DbgRegVid              ;Display BackLink's Register values
        CALL dbgDispMenu            ;Display menu
        PUSH dbgX                   ;Back to where we were
        PUSH dbgY
        CALL FWORD PTR _SetXY


        ;Display Instruction at CS:EIP
        MOV EBX,DbgpTSSSave         ;Get USER pUserTSS
        MOV EAX, [EBX+TSS_EIP]
        MOV dbgCrntAdd, EAX
        CALL dbgShowBP
        PUSH EAX
        CALL _disassemble           ;This puts the instruction on the line
        MOV NextEIP, EAX
```

continues

Listing 23.2. continued

```
        CALL dbgCheckScroll         ;Fall through to keyboard loop

;=============================================================
        ;Now we read the keyboard
dbg00:
        MOV EAX, OFFSET dbgKeyCode
        PUSH EAX
        CALL ReadDbgKbd             ;
        MOV EAX, dbgKeyCode
        AND EAX, 0FFh               ;Lop off key status bytes

        CMP EAX, 1Bh                ;ESCAPE (Exit)
        JE dbgExit

        CMP EAX, 0Fh                ;Single Step (F1)
        JNE dbg02
        MOV EBX,DbgpTSSSave         ;Get USER pUserTSS
        MOV ECX,[EBX+TSS_EFlags]    ;
        OR ECX,00000100h            ;Set TF in flags for single step
        MOV [EBX+TSS_EFlags],ECX
        JMP dbgExit
dbg02:
        CMP EAX, 10h                ;Set BP (Current Address - F2)
        JNE dbg03
        MOV EBX, dbgCrntAdd         ;Address displayed
        MOV dbgBPAdd, EBX           ;Save it so we know where BP is
        MOV DR0, EBX                ;Move into BreakPoint Reg 0
        XOR EAX, EAX
        MOV DR6, EAX
        MOV EAX, 00000002h          ;BP0 Set global
        MOV DR7, EAX
        JMP dbg00                   ;
dbg03:
        CMP EAX, 11h                ;Clear BP (Current Address - F3)
        JNE dbg04
        XOR EAX, EAX
        MOV dbgBPAdd, EAX           ;Clear BP saved address
        MOV DR7, EAX
        JMP dbg00                   ;
dbg04:
        CMP EAX, 12h                ;Return to CS:EIP (F4)
        JNE dbg05
        MOV EBX,DbgpTSSSave         ;Get USER pUserTSS
        MOV EAX, [EBX+TSS_EIP]
        MOV dbgCrntAdd, EAX
        CALL dbgShowBP
        PUSH EAX
        CALL _disassemble           ;This puts the instruction on the line
        MOV NextEIP, EAX
        CALL dbgCheckScroll         ;See if we need to scroll up
        JMP dbg00                   ;
dbg05:
        CMP EAX, 13h                ;Display Exchanges (F5)
        JNE dbg06
        CALL dbgDispExchs
        JMP dbg000                  ;Full display
```

```
dbg06:
        CMP EAX, 14h                ;Task array display (F6)
        JNE dbg07
        CALL dbgDispTasks
        JMP dbg000                  ;
dbg07:
        CMP EAX, 15h                ;Not used yet
        JNE dbg08
        JMP dbg00                   ;
dbg08:
        CMP AL, 16h                 ;Set Disassembly Address (F8)
        JNE dbg09
        CALL dbgSetAddr             ;Sets NextEIP
        PUSH dbgX                   ;Back to where we were
        PUSH dbgY
        CALL FWORD PTR _SetXY
        MOV EAX, NextEIP
        MOV dbgCrntAdd, EAX
        CALL dbgShowBP
        PUSH EAX
        CALL _disassemble           ;This puts the instruction on the line
        MOV NextEIP, EAX
        CALL dbgCheckScroll         ;See if we need to scroll up
        JMP dbg00                   ;
dbg09:
        CMP AL, 17h                 ;Memory Dump Bytes (F9)
        JNE dbg10
        MOV BL, 00
        MOV dbgfDumpD, BL
        CALL dbgDump                ;
        JMP dbg000
dbg10:
        CMP AL, 18h                 ;Memory Dump DWORDS (F10)
        JNE dbg12
        MOV BL, 0FFh
        MOV dbgfDumpD, BL
        CALL dbgDump                ;
        JMP dbg000
dbg12:
        CMP AL, 01Ah                ;Info Address dump (F12)
        JNE dbg13
        CALL DbgInfo                ;
        JMP dbg00

dbg13:  CMP AL, 02h                 ;Display next Instruction (Down Arrow)
        JNE dbg14
        MOV EAX, NextEIP
        MOV dbgCrntAdd, EAX
        CALL dbgShowBP
        PUSH EAX
        CALL _disassemble           ;This puts the instruction on the line
        MOV NextEIP, EAX
        CALL dbgCheckScroll         ;See if we need to scroll up
        JMP dbg00

dbg14:
        JMP dbg00               ;GO back for another key
```

continues

Listing 23.2. continued

```
DbgExit:

        LEA EAX,dbgX            ;Query XY
        PUSH EAX
        LEA EAX,dbgY
        PUSH EAX
        CALL FWORD PTR _GetXY

        PUSH DbgVidSave
        CALL FWORD PTR _SetVidOwner ;Change screens back

        MOV EAX, DbgpTSSSave        ;Return saved pRunTSS
        MOV pRunTSS, EAX
        MOV BX, [EAX+Tid]
        MOV TSS_Sel, BX             ;Set up caller's TSS selector

        JMP FWORD PTR [TSS]

        ;Next time we enter the debugger task it will be here!
        JMP DbgTask                 ;Back to beginning

;=============================================================

dbgShowBP:
        ;This compares the current breakpoint to the address
        ;we are about to display. If they are the same,
        ;we put up an asterisk to indicate it's the breakpoint
        ;EAX must be preserved

        MOV EBX, dbgCrntAdd         ;Address displayed
        MOV ECX, dbgBPAdd           ;BP Address
        CMP EBX, ECX
        JZ dbgShowBP1
        RETN
dbgShowBP1:
        PUSH EAX                    ;Save EAX across call
        PUSH OFFSET dbgAsterisk     ;3 params to TTYOut
        PUSH 01h
        PUSH 47h                    ;Reverse Vid WHITE on RED
        CALL FWORD PTR _TTYOut
        POP EAX
        RETN

;=============================================================
;This sets up and calls to display all of the register
;information on the right side of the debugger video display

DbgRegVid:
        MOV EBX,DbgpTSSSave         ;EBX MUST be DbgpTSSSave
        MOV ECX,00                  ;TSS Display
        MOV ESI,OFFSET DbgTxt00
        XOR EAX,EAX
        MOV AX,[EBX+TSSNum]         ;Number of this TSS
        CALL DispRegs

        MOV ECX,01                  ;EAX Display
```

```
MOV ESI,OFFSET DbgTxt01
MOV EAX,[EBX+TSS_EAX]
CALL DispRegs

MOV ECX,02                  ;EBX Display
MOV ESI,OFFSET DbgTxt02
MOV EAX,[EBX+TSS_EBX]
CALL DispRegs

MOV ECX,03                  ;ECX Display
MOV ESI,OFFSET DbgTxt03
MOV EAX,[EBX+TSS_ECX]
CALL DispRegs

MOV ECX,04                  ;EDX Display
MOV ESI,OFFSET DbgTxt04
MOV EAX,[EBX+TSS_EDX]
CALL DispRegs

MOV ECX,05                  ;ESI Display
MOV ESI,OFFSET DbgTxt05
MOV EAX,[EBX+TSS_ESI]
CALL DispRegs

MOV ECX,06                  ;EDI Display
MOV ESI,OFFSET DbgTxt06
MOV EAX,[EBX+TSS_EDI]
CALL DispRegs

MOV ECX,07                  ;EBP Display
MOV ESI,OFFSET DbgTxt07
MOV EAX,[EBX+TSS_EBP]
CALL DispRegs

MOV ECX,08                  ;SS Display
MOV ESI,OFFSET DbgTxt08
XOR EAX,EAX
MOV AX,[EBX+TSS_SS]
CALL DispRegs

MOV ECX,09                  ;ESP Display
MOV ESI,OFFSET DbgTxt09
MOV EAX,[EBX+TSS_ESP]
CALL DispRegs

MOV ECX,10                  ;CS Display
MOV ESI,OFFSET DbgTxt10
XOR EAX,EAX
MOV AX,[EBX+TSS_CS]
CALL DispRegs

MOV ECX,11                  ;EIP Display
MOV ESI,OFFSET DbgTxt11
MOV EAX,[EBX+TSS_EIP]
CALL DispRegs
MOV ECX,12                  ;DS Display
MOV ESI,OFFSET DbgTxt12
```

continues

Listing 23.2. continued

```
        XOR EAX,EAX
        MOV AX,[EBX+TSS_DS]
        CALL DispRegs
        MOV ECX,13                  ;ES Display
        MOV ESI,OFFSET DbgTxt13
        XOR EAX,EAX
        MOV AX,[EBX+TSS_ES]
        CALL DispRegs
        MOV ECX,14                  ;FS Display
        MOV ESI,OFFSET DbgTxt14
        XOR EAX,EAX
        MOV AX,[EBX+TSS_FS]
        CALL DispRegs
        MOV ECX,15                  ;GS Display
        MOV ESI,OFFSET DbgTxt15
        XOR EAX,EAX
        MOV AX,[EBX+TSS_GS]
        CALL DispRegs
        MOV ECX,16                  ;EFlags Display
        MOV ESI,OFFSET DbgTxt16
        MOV EAX,[EBX+TSS_EFlags]
        CALL DispRegs
        MOV ECX,17                  ;CR0 Display
        MOV ESI,OFFSET DbgTxt17
        MOV EAX,CR0
        CALL DispRegs
        MOV ECX,18                  ;CR2 Display
        MOV ESI,OFFSET DbgTxt18
        MOV EAX,CR2
        CALL DispRegs
        MOV ECX,19                  ;CR3 Display
        MOV ESI,OFFSET DbgTxt19
        MOV EAX,CR3
        CALL DispRegs
        MOV ECX,20                  ;Fault Error Code Display
        MOV ESI,OFFSET DbgTxt20
        MOV EAX,dbgFltErc
        CALL DispRegs
        RETN
;=============================================================================
;
; This is for Debugger Register display
; Call with: EAX loaded with value to display (from TSS reg)
;        ECX loaded with number of text line to display on
;        ESI loaded with EA of text line to display
;   We save all registers cause the vid calls don't

DispRegs:
        PUSHAD

        PUSH EAX            ;Save number to display

        PUSH 66
        PUSH ECX
        CALL FWORD PTR _SetXY
```

```
        PUSH ESI
        PUSH 05h
        PUSH 07h
        CALL FWORD PTR _TTYOut

        POP EAX             ;Get number back for display

        PUSH EAX
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH OFFSET dbgBuf
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _TTYOut
        POPAD
        RETN
;=============================================
; This displays the debugger function key menu

dbgDispMenu:
        PUSH 0              ;Display Debugger FKey Menu
        PUSH 24
        CALL FWORD PTR _SetXY

        LEA EAX,dbgMenu
        PUSH EAX
        PUSH 78
        PUSH 70h
        CALL FWORD PTR _TTYOut

        PUSH 25
        PUSH 24
        CALL FWORD PTR _SetXY

        LEA EAX,dbgSpace
        PUSH EAX
        PUSH 1
        PUSH 07h
        CALL FWORD PTR _TTYOut

        PUSH 51
        PUSH 24
        CALL FWORD PTR _SetXY

        LEA EAX,dbgSpace
        PUSH EAX
        PUSH 1
        PUSH 07h
        CALL FWORD PTR _TTYOut

        RETN
;============================
;Allows the user to pick the currently displayed address

dbgSetAddr:
        PUSH 0                      ;Goto Query Line
```

continues

Listing 23.2. continued

```
        PUSH 23                     ;
        CALL FWORD PTR _SetXY

        LEA EAX, dbgTxtAddr
        PUSH EAX
        PUSH 16
        PUSH 07h
        CALL FWORD PTR _TTYOut
        CMP EAX, 0
        JNE DumpDone

        LEA EAX, DbgBuf2            ;
        PUSH EAX                    ;pEdString
        PUSH cbBufLen2              ;Crnt size
        PUSH 8                      ;Max size
        LEA EAX, cbBufLen2          ;
        PUSH EAX                    ;ptr to size returned
        LEA EAX, dbgChar            ;
        PUSH EAX                    ;ptr to char returned
        PUSH 70h                    ;Black On White
        CALL FWORD PTR _EditLine    ;Ignore error if any

        MOV AL, dbgChar             ;did they exit with CR?
        CMP AL, 0Dh
        JNE dbgSetAddrDone

        LEA EAX, dbgBuf2            ;Convert String to DD
        PUSH EAX                    ;ptr to string
        LEA EAX, dbgNextAdd
        PUSH EAX                    ;ptr to destination DD
        PUSH cbBufLen2              ;length of string
        CALL HexToDD                ;dbgDumpAdd has address to dump!
        CMP EAX, 0
        JNE dbgSetAddrDone

        MOV EAX, dbgNextAdd
        MOV NextEIP, EAX
dbgSetAddrDone:
        CALL dbgClearQuery
        RETN                        ;Go home...
;============================
;Queries user for address then dumps data to screen

dbgDump:
        PUSH 0                      ;Goto Query Line
        PUSH 23                     ;
        CALL FWORD PTR _SetXY

        LEA EAX, dbgTxtAddr
        PUSH EAX
        PUSH 16
        PUSH 07h
        CALL FWORD PTR _TTYOut
        CMP EAX, 0
        JNE DumpDone
```

```
        LEA EAX, DbgBuf2        ;
        PUSH EAX                ;pEdString
        PUSH cbBufLen2          ;Crnt size
        PUSH 8                  ;Max size
        LEA EAX, cbBufLen2      ;
        PUSH EAX                ;ptr to size returned
        LEA EAX, dbgChar        ;
        PUSH EAX                ;ptr to char returned
        PUSH 70h                ;Black On White
        CALL FWORD PTR _EditLine    ;Ignore error if any

        MOV AL, dbgChar         ;did they exit with CR?
        CMP AL, 0Dh
        JE dbgDoDump
        CALL dbgClearQuery
        RETN                    ;Go home...

dbgDoDump:
        LEA EAX, dbgBuf2        ;Convert String to DD
        PUSH EAX                ;ptr to string
        LEA EAX, dbgDumpAdd
        PUSH EAX                ;ptr to destination DD
        PUSH cbBufLen2          ;length of string
        CALL HexToDD            ;dbgDumpAdd has address to dump!
        CMP EAX, 0
        JNE DumpDone

        CALL FWORD PTR _ClrScr

dbgDump00:
        MOV DWORD PTR dbgGPdd1, 24  ;line counter begins at 24
dbgDump01:
        MOV DWORD PTR dbgGPdd3, 4   ;number of quads per line
        PUSH dbgDumpAdd     ;convert address to text
        LEA EAX, dbgBuf
        PUSH EAX
        CALL DDtoHex

        LEA EAX, dbgBuf
        PUSH EAX
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _TTYOut
        CMP EAX, 0
        JNE DumpDone
dbgDump02:
        MOV DWORD PTR dbgGPdd2, 6   ;byte offset begins at 6
        LEA EAX, dbgSpace
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut
        CMP EAX, 0
        JNE DumpDone
        MOV EBX, dbgDumpAdd ;get dump address
        MOV EAX, [EBX]      ;Get what it's pointing to
        PUSH EAX            ;make it a DD Text
```

continues

Listing 23.2. continued

```
        LEA EAX, dbgBuf
        PUSH EAX
        CALL DDtoHex
        MOV AL, dbgfDumpD   ;Dumping DWORDS
        CMP AL, 0
        JE DumpB            ;NO - go to display bytes

        LEA EAX, dbgBuf     ;Yes display Quad
        PUSH EAX
        PUSH 8
        PUSH 07
        CALL FWORD PTR _TTYOut
        JMP DumpDin

dumpB:
        LEA EAX, dbgBuf         ;Display First byte
        ADD EAX, dbgGPdd2
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut  ;ignore error

        LEA EAX, dbgSpace       ;Display 1 spaces
        PUSH EAX
        PUSH 1
        PUSH 07h
        CALL FWORD PTR _TTYOut
        DEC dbgGPdd2            ;point to second 2 bytes
        DEC dbgGPdd2

        LEA EAX, dbgBuf         ;display 2st byte
        ADD EAX, dbgGPdd2
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut  ;ignore error

        LEA EAX, dbgSpace       ; display 1 space
        PUSH EAX
        PUSH 1
        PUSH 07h
        CALL FWORD PTR _TTYOut

        DEC dbgGPdd2
        DEC dbgGPdd2

        LEA EAX, dbgBuf          ;display 3rd byte
        ADD EAX, dbgGPdd2
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut  ;ignore error

        LEA EAX, dbgSpace       ;a space
        PUSH EAX
        PUSH 1
```

```
        PUSH 07h
        CALL FWORD PTR _TTYOut
        DEC dbgGPdd2
        DEC dbgGPdd2

        LEA EAX, dbgBuf         ;display 4th byte
        ADD EAX, dbgGPdd2
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut  ;ignore error
DumpDin:
        INC dbgDumpAdd
        INC dbgDumpAdd
        INC dbgDumpAdd
        INC dbgDumpAdd
        DEC dbgGPdd3            ;done with 4 quads??
        JNZ dbgDump02           ;NO - go back for next 4 bytes

        LEA EAX, dbgSpace       ;Yes - Display 2 spaces
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut

        LEA EAX,dbgX            ;Query XY
        PUSH EAX
        LEA EAX,dbgY
        PUSH EAX
        CALL FWORD PTR _GetXY

        PUSH dbgX               ;Put 16 TEXT chars on right
        PUSH dbgY
        MOV EAX, dbgDumpAdd
        SUB EAX, 16
        PUSH EAX
        PUSH 16
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;ignore error

        LEA EAX, dbgCRLF        ;Do CR/LF
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut
        ;
        DEC dbgGPdd1        ;23lines yet??
        JNZ dbgDump01       ;NO
        ;
        LEA EAX, dbgCont    ;"Continue" Text
        PUSH EAX
        PUSH 31             ;size of "cont text"
        PUSH 07h
        CALL FWORD PTR _TTYOut
dbgDump03:
        MOV EAX, OFFSET dbgKeyCode
        PUSH EAX
```

continues

Listing 23.2. continued

```
        CALL ReadDbgKbd
        MOV EAX, dbgKeyCode
        AND EAX, 0FFh               ;Lop off key status bytes
        CMP EAX, 0
        JE dbgDump03
        CMP EAX, 1Bh            ;Escape (Quit??)
        JE DumpDone

        LEA EAX, dbgCRLF        ;Do CR/LF
        PUSH EAX
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut
        JMP dbgDump00

        ;
 DumpDone:
        CALL FWORD PTR _ClrScr
        MOV DWORD PTR dbgX, 0
        MOV DWORD PTR dbgY, 0
        RETN
;==========================

;Displays exchanges that are allocated along with
;messages or tasks that may be waiting at them

dbgDispExchs:

        MOV DWORD PTR dbgGPdd2, 0       ;Exch# we are on
dbgDE00:
        CALL FWORD PTR _ClrScr
        PUSH 0                  ;Col
        PUSH 0                  ;Line for labels
        PUSH OFFSET dbgExchMsg  ;
        PUSH 54                 ;Length of message
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;
        MOV DWORD PTR dbgGPdd1, 1           ;line we are on

        ;First we do the exchange on the current line
dbgDE01:
        PUSH dbgGPdd2           ;Convert Exch number for display
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH 0                  ;Col
        PUSH dbgGPdd1           ;Line we are on
        PUSH OFFSET dbgBuf      ;ExchNum
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        ;Then we do the Exch Owner (Job Number) next to it

        MOV EAX, dbgGPdd2       ; Exch number
```

```
        MOV EDX, sEXCH          ; Compute offset of Exch in rgExch
        MUL EDX                 ;
        MOV EDX,prgExch         ; Add offset of rgExch => EAX
        ADD EAX,EDX             ; EAX now pts to Exch
        MOV dbgGPdd3, EAX       ; pExch into save variable
        MOV EBX, [EAX+Owner]    ; pJCB of Owner into EBX
        XOR EAX, EAX            ; Clear for use as JobNum
        OR EBX, EBX             ; pNIL? (No owner if so)
        JZ dbgDE03
        MOV EAX, [EBX+JobNum]   ;
dbgDE03:
        PUSH EAX                ;Convert Job Number
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH 10                 ;Col
        PUSH dbgGPdd1           ;Line
        PUSH OFFSET dbgBuf      ;
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        MOV dbgGPdd5, 0         ;Set pNextMsg to 0

        ;See if there is a first message

        MOV EAX, dbgGPdd3       ;pExch -> EAX
        MOV EBX, [EAX+EHead]    ;pMsg -> EBX
        OR EBX, EBX             ;Is is NIL (no msg or task)?
        JZ dbgDE13              ;Yes. Go to next Exch

        MOV EBX, [EAX+fEMsg]    ;MsgFlag -> EBX
        OR EBX, EBX             ;Is is 1 (a message)?
        JZ dbgDE05              ;No, Go check for tasks
        MOV EBX, [EAX+EHead]    ;pMsg -> EBX

        ;Display Messages
dbgDE04:
        MOV DWORD PTR dbgGPdd6, 0   ;Flag to indicate we are doing messages
        MOV EAX, [EBX+NextLB]   ;For next msg in chain (if it exists)
        MOV dbgGPdd5, EAX       ;Save for loop
        MOV EAX, [EBX+DataHi]   ;Get dMsg1
        MOV EDX, [EBX+DataLo]   ;Get dMsg2
        PUSH EDX                ;Save dMsg2

        PUSH EAX                ;Convert dMsg1
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH 20                 ;Col
        PUSH dbgGPdd1           ;Line
        PUSH OFFSET dbgBuf      ;
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        POP EDX                 ;Get dMsg2 back
```

continues

Listing 23.2. continued

```
        ;Could have left it on stack, but would be confusing later...
        ;"simplicity of maintenance is as important as simplicity of design"

        PUSH EDX                ;Convert dMsg2
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH 30                     ;Col
        PUSH dbgGPdd1               ;Line
        PUSH OFFSET dbgBuf          ;
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;
        JMP  dbgDE07                ;Next line please

        ;See if there are tasks waiting
dbgDE05:
        MOV DWORD PTR dbgGPdd6, 1   ;Flag to indicate we are doing tasks
        MOV DWORD PTR dbgGPdd5, 0   ;Clear pNextTask
        MOV EAX, dbgGPdd3       ;pExch -> EAX
        MOV EBX, [EAX+EHead]    ;pTSS -> EBX
        OR EBX, EBX             ;Is is 0 (no TSS)?
        JZ dbgDE07              ;
dbgDE06:
        MOV EAX, [EBX+NextTSS]
        MOV dbgGPdd5, EAX       ;Save ptr to next task if it exists
        XOR EAX, EAX
        MOV AX, [EBX+TSSNum]    ;Get Number of Task at exch

        PUSH EAX                ;Convert Task Number
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH 50                 ;Col
        PUSH dbgGPdd1           ;Line
        PUSH OFFSET dbgBuf      ;
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

dbgDE07:
        INC dbgGPdd1            ;Next line
        CMP DWORD PTR dbgGPdd1, 23      ;23 lines yet?
        JB dbgDE09              ;No
        ;
dbgDE08:
        PUSH 0                  ;Col
        PUSH 24                 ;Line
        PUSH OFFSET dbgCont     ;
        PUSH 31                 ;length of Cont string
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        MOV EAX, OFFSET dbgKeyCode
        PUSH EAX
```

```
        CALL ReadDbgKbd
        MOV EAX, dbgKeyCode
        AND EAX, 0FFh               ;Lop off key status bytes
        CMP EAX, 1Bh                ;Escape (Quit??)
        JE dbgDEDone

        CMP DWORD PTR dbgGPdd2, nDynEXCH    ; Number of dynamic exchanges
        JAE dbgDEDone               ; All Exchs displayed

        CALL FWORD PTR _ClrScr      ;
        PUSH 0                      ;Col
        PUSH 0                      ;Line for labels
        PUSH OFFSET dbgExchMsg      ;
        PUSH 54                     ;Length of message
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;
        MOV DWORD PTR dbgGPdd1, 1   ;line. We are on line 1 again

dbgDE09:
        MOV EBX, dbgGPdd5           ;Set up to loop for next msg/task
        XOR EBX, EBX                ;Another pointer in the link?
        JZ dbgDE13                  ;No
        MOV EAX, dbgGPdd6           ;
        OR EAX, EAX                 ;NonZero if we are doing tasks
        JNZ dbgDE06                 ;Tasks
        JMP dbgDE04                 ;Messages
dbgDE13:
        INC dbgGPdd2                ; Exch number
        CMP DWORD PTR dbgGPdd2, nDynEXCH    ; Number of dynamic exchanges
        JAE dbgDE08                 ; Go back for prompt (to pause)
        JMP dbgDE01                 ; Back to display new exch num
dbgDEDone:
        CALL FWORD PTR _ClrScr
        MOV DWORD PTR dbgX, 0
        MOV DWORD PTR dbgY, 0
        RETN

;===========================
;Displays Tasks that are active along with
;pertinent address info about them

dbgDispTasks:
        MOV DWORD PTR dbgGPdd2, 1       ;Task# we are on
dbgDT00:
        CALL FWORD PTR _ClrScr
        PUSH 0                  ;Col
        PUSH 0                  ;Line for labels
        PUSH OFFSET dbgTaskMsg  ;
        PUSH 54                 ;Length of message
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;
        MOV DWORD PTR dbgGPdd1, 1           ;line we are on

        ;First we get pTSS and see if it is valid
        ;If so, we get all the data BEFORE we display it
        ;If not, we increment TSS number and go back
        ;for the next one
```

continues

Listing 23.2. continued

```
dbgDT01:
        ;We get pJCB out of TSS and get JobNum from JCB

        MOV EAX, dbgGPdd2       ; Task number
        CMP EAX, 1              ; Is this TSS 1 (Static memory)
        JNE dbgDT02
        MOV EBX, OFFSET MonTSS  ;
        JMP SHORT dbgDT04
dbgDT02:
        CMP EAX, 2              ; Is this TSS 2 (Static memory)
        JNE dbgDT03
        MOV EBX, OFFSET DbgTSS  ;
        JMP SHORT dbgDT04
dbgDT03:
        MOV EBX, pDynTSSs
        DEC EAX                 ;Make TSS Num offset in dynamic array
        DEC EAX                 ;of TSSs
        DEC EAX
        MOV ECX, 512
        MUL ECX
        ADD EBX, EAX            ;EBX points to TSS!
dbgDT04:
        ;EBX has pTSS of interest
        MOV dbgGPdd5, EBX       ;Save pTSS for display
        XOR ECX, ECX            ;
        MOV CL, [EBX+Priority]  ;Priority of this task
        MOV dbgGPdd6, ECX
        MOV EAX, [EBX+TSS_pJCB] ;EAX has pJCB
        OR EAX, EAX             ;NON zero means it's valid
        JNZ dbgDT05
        MOV EAX, dbgGPdd2       ;Not used, go for next Task number
        INC EAX
        CMP EAX, nTSS           ;
        JE dbgDT06
        MOV dbgGPdd2, EAX
        JMP SHORT dbgDT01       ;back for next one
dbgDT05:
        ;EAX now pJCB
        MOV dbgGPdd4, EAX       ;Save pJCB for display
        MOV EBX, [EAX]          ;Job number is first DD in JCB
        MOV dbgGPdd3, EBX

        PUSH dbgGPdd2           ;Convert TSS number for display
        PUSH OFFSET dbgBuf
        CALL DDtoHex
        PUSH 0                  ;Col
        PUSH dbgGPdd1           ;Line we are on
        PUSH OFFSET dbgBuf      ;TaskNum
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        PUSH dbgGPdd3           ;Convert and display Job number
        PUSH OFFSET dbgBuf      ;
        CALL DDtoHex
        PUSH 10                 ;Col
```

```
        PUSH dbgGPdd1           ;Line we are on
        PUSH OFFSET dbgBuf      ;
        PUSH 8                  ;Size
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        PUSH dbgGPdd4           ;Convert and display pJCB
        PUSH OFFSET dbgBuf      ;
        CALL DDtoHex
        PUSH 20                 ;Col
        PUSH dbgGPdd1           ;Line we are on
        PUSH OFFSET dbgBuf      ;
        PUSH 8                  ;Size
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        PUSH dbgGPdd5           ;Convert and display pTSS
        PUSH OFFSET dbgBuf      ;
        CALL DDtoHex
        PUSH 30                 ;Col
        PUSH dbgGPdd1           ;Line we are on
PUSH OFFSET dbgBuf      ;
        PUSH 8                  ;Size
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        PUSH dbgGPdd6           ;Convert and display Priority
        PUSH OFFSET dbgBuf      ;
        CALL DDtoHex
        PUSH 40                 ;Col
        PUSH dbgGPdd1           ;Line we are on
PUSH OFFSET dbgBuf      ;
        PUSH 8                  ;Size
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        MOV EAX, dbgGPdd2       ;go for next Task number
        INC EAX
        CMP EAX, nTSS           ;
        JE dbgDT06
        MOV dbgGPdd2, EAX       ;Save it for the top

        INC dbgGPdd1            ;Next line
        CMP DWORD PTR dbgGPdd1, 23      ;23 lines yet?
        JAE dbgDT06             ;Yes, continue prompt
        JMP dbgDT01             ;No, go back for next
dbgDT06:
        PUSH 0                  ;Col
        PUSH 24                 ;Line
        PUSH OFFSET dbgCont     ;
        PUSH 31                 ;length of Cont string
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;

        MOV EAX, OFFSET dbgKeyCode
        PUSH EAX
        CALL ReadDbgKbd
        MOV EAX, dbgKeyCode
```

continues

Listing 23.2. continued

```
        AND EAX, 0FFh               ;Lop off key status bytes
        CMP EAX, 1Bh                ;Escape (Quit??)
        JE dbgDTDone
        JMP dbgDT00                 ;Back for next screen
dbgDTDone:
        CALL FWORD PTR _ClrScr
        MOV DWORD PTR dbgX, 0
        MOV DWORD PTR dbgY, 0
        RETN

;=============================================================================
; This is for Debugger Address Info display
; Call with:
;       EAX loaded with address to display (Linear Address)
;       ESI loaded with EA of text line to display
;   We save all registers cause the vid calls don't
;=============================================================================
DispAddr:
        PUSHAD
        PUSH EAX            ;Save number to display

        PUSH ESI                ;ptr to line
        PUSH 06h                ;Length of line
        PUSH 07h                ;Vid Attribute
        CALL FWORD PTR _TTYOut  ;Do it

        POP EAX                ;Get number back for display

        PUSH EAX
        PUSH OFFSET dbgBuf
        CALL DDtoHex

        PUSH OFFSET dbgBuf
        PUSH 8
        PUSH 07h
        CALL FWORD PTR _TTYOut

        PUSH OFFSET dbgCRLF
        PUSH 2
        PUSH 07h
        CALL FWORD PTR _TTYOut

        CALL dbgCheckScroll

        POPAD
        RETN
;=============================================
;DbgInfo - Displays important linear address for the OS

DbgInfo:
        MOV ESI,OFFSET DbgM0    ;IDT
        LEA EAX, IDT
        CALL DispAddr
        MOV ESI,OFFSET DbgM1    ;GDT
        LEA EAX, GDT
        CALL DispAddr
```

```
        MOV ESI,OFFSET DbgM2    ;RQBs
        MOV EAX, pRQBs
        CALL DispAddr
        MOV ESI,OFFSET DbgM3    ;MonTSS
        MOV EAX, OFFSET MonTSS
        CALL DispAddr
        MOV ESI,OFFSET DbgM4    ;pTSS3
        MOV EAX, pDynTSSs
        CALL DispAddr
        MOV ESI,OFFSET DbgM5    ;LBs
        LEA EAX, rgLBs
        CALL DispAddr
        MOV ESI,OFFSET DbgM6    ;RdyQ
        LEA EAX, RdyQ
        CALL DispAddr
        MOV ESI,OFFSET DbgM7    ;JCBs
        MOV EAX, pJCBs
        CALL DispAddr
        MOV ESI,OFFSET DbgM8    ;SVCs
        LEA EAX, rgSVC
        CALL DispAddr
        MOV ESI,OFFSET DbgM9    ;Exchs
        MOV EAX, prgExch
        CALL DispAddr
        MOV ESI,OFFSET DbgPA    ;PAM (Page Allocation map)
        LEA EAX, rgPAM
        CALL DispAddr
        MOV ESI,OFFSET DbgMB    ;Timer Blocks
        LEA EAX, rgTmrBlks
        CALL DispAddr
        RETN

;=====================================================
;All of the debugger text is displayed in a window
;between columns 0 and 66, and line 0 to 24. The other
;areas are reserved for the menu, query line,
;and the register display.
;This checks to see if the cursor is on line 23.
;If so, we scroll up the text area by one line.

dbgCheckScroll:
        LEA EAX,dbgX            ;Query XY (See what line and Col)
        PUSH EAX
        LEA EAX,dbgY
        PUSH EAX
        CALL FWORD PTR _GetXY
        CMP DWORD PTR dbgY, 23  ;Are we at bottom (just above menu)??
        JB  dbgNoScroll         ;No, go back for next key

        PUSH 0                  ;Yes, Scroll test area (Col 0-64, Line 0-24)
        PUSH 0
        PUSH 66                 ;Columns 0-65
        PUSH 24                 ;Lines 0-23
        PUSH 1                  ;fUp (1)
        CALL FWORD PTR _ScrollVid

        PUSH 0                  ;Got to Column 0, Line 22
        PUSH 22
```

continues

Listing 23.2. continued

```
        CALL FWORD PTR _SetXY
dbgNoScroll:
        RETN
;
;=====================================================
;Clear the query line (Line 23, 40 chars)

dbgClearQuery:
        PUSH 0                  ;Col 0, Line 23
        PUSH 23
        PUSH OFFSET dbgClear
        PUSH 40
        PUSH 07h
        CALL FWORD PTR _PutVidChars ;ignore error
        RETN

;==================== module end =====================
```

24

Selected Device Driver Code

This chapter contains the source code for two MMURTL device drivers. Comments, in addition to those in the code itself, precede sections of the code in order to explain the purpose of an entire section. The code comments should suffice for the detailed explanations of otherwise confusing code.

The two device drivers are for the IDE disk drives and the RS232 asynchronous communications device (UARTS).

Unlike large portions of the rest of the MMURTL code, many device drivers are written in C. Even some of the ISRs are done in C, although I would rather have done them in assembler. Time was the deciding factor.

IDE Disk Device Driver

The IDE (Integrated Drive Electronics) device driver was actually one of the easiest drivers to write. All the hardware commands are well documented, and they are also fairly compatible with the MFM hard disk controllers. They were designed that way. In fact, this device driver should work with MFM drives, but I haven't owned any for a couple of years and can't find any to test it. If you have MFM drives, you're on your own, but I think you're OK.

One of the things you'll note is that I don't depend on CMOS RAM locations to provide the hard disk drive geometry. I found so many variations in the different ROMs that I gave up and actually read the disk itself to find out. This seems to be the best, and maybe the only, dependable way to do it.

You will find these little `#define` statements at the top of almost every C source file I have. They simply save some typing and help my brain realize what is signed and unsigned. You'll see that I sometimes slip back to the long-hand notation (`unsigned long int`, for example), but I try not to. You may also notice that almost everything in MMURTL is unsigned anyway. This eliminates a lot of confusion for me.

The CM32 C compiler requires ANSI prototypes for functions. All the operating system calls are prototyped here. There are no include files, so you don't have to go digging to see what I'm doing. I feel this is the best way to try to pass information in a source file to someone; otherwise I would have just included the standard MMURTL header files. See Listing 24.1.

Listing 24.1. IDE device driver data and externals.

```
#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char

/* MMURTL OS PROTOTYPES */
```

```
extern far AllocExch(U32  *pExchRet);
extern far U32  InitDevDr(U32  dDevNum,
                          S8  *pDCBs,
                          U32  nDevices,
                          U32  dfReplace);

extern far U32  UnMaskIRQ(U32  IRQNum);
extern far U32  MaskIRQ(U32  IRQNum);
extern far U32  SetIRQVector(U32  IRQNum, S8  *pIRQ);
extern far U32  EndOfIRQ(U32  IRQNum);
extern far U32  SendMsg(U32  Exch, U32  msg1, U32  msg2);
extern far U32  ISendMsg(U32  Exch, U32  msg1, U32  msg2);
extern far U32  WaitMsg(U32  Exch, U32  *pMsgRet);
extern far U32  CheckMsg(U32  Exch, U32  *pMsgRet);
extern far U32  Alarm(U32  Exch, U32  count);
extern far U32  KillAlarm(U32  Exch);
extern far U32  Sleep(U32  count);
extern far void MicroDelay(U32  us15count);
extern far void OutByte(U8 Byte, U16 wPort);
extern far void OutWord(U16 Word, U16 wPort);
extern far U8 InByte(U16 wPort);
extern far U16 InWord(U16 wPort);
extern far U8 ReadCMOS(U16 Address);
extern far void CopyData(U8 *pSource, U8 *pDestination, U32 dBytes);
extern far InWords(U32 dPort, U8 *pDataIn, U32 dBytes);
extern far OutWords(U32 dPort, U8 *pDataOut, U32 dBytes);
```

While writing the operating system, I needed a method to get data to the screen. The monitor has a function that works very similarly to `printf` in C. It's called `xprintf()`. Any section of the operating system code included at build time can use this function for displaying troubleshooting information. You must simply keep in mind that it writes to the video screen for the job that called the device driver. See Listing 24.2.

Listing 24.2. Continuation of IDE driver data.

```
/* Near External for troubleshooting */

extern long xprintf(char *fmt, ...);


/* LOCAL PROTOTYPES */

U32  hdisk_setup(void);
static void interrupt hdisk_isr(void);  /* The HD interrupt function */
static U32  hd_format_track(U32 dLBA, U32 dnBlocks);
static void hd_reset(void);
static U32  send_command(U8  parm);
static U32  hd_wait (void);
static U32  check_busy(void);
static U32  hd_seek(U32 dLBA);
static U32  hd_recal(U8 drive);
static U32  hd_write(U32 dLBA, U32 dnBlocks, U8 *pDataOut);
```

continues

Listing 24.2. continued

```
static U32  hd_read(U32 dLBA, U32 dnBlocks, U8 *pDataIn);
static U32  hd_status(U8 LastCmd);
static U32  setupseek(U32 dLBA, U32 nBlks);
static U32  hd_init(U8 drive);
static U32 ReadSector(U32 Cylinder, U32 HdSect, U8 *pDataRet);

/* The following 3 calls are required in every MMURTL device driver */

static U32  hddev_op(U32  dDevice,
            U32  dOpNum,
            U32  dLBA,
            U32  dnBlocks,
            U8  *pData);

static U32  hddev_stat(U32  dDevice,
              S8 * pStatRet,
              U32  dStatusMax,
              U32  *pdSatusRet);

static U32  hddev_init(U32  dDevNum,
              S8  *pInitData,
              U32   sdInitData);

/* LOCAL DEFINITIONS */

#define ok 0

/* Error Codes to return */

#define ErcNoMsg        20
#define ErcNotInstalled 504

#define ErcBadBlock     651
#define ErcAddrMark     652
#define ErcBadECC       653
#define ErcSectNotFound 654
#define ErcNoDrive0     655
#define ErcNotSupported 656
#define ErcBadHDC       658
#define ErcBadSeek      659
#define ErcHDCTimeOut   660
#define ErcOverRun      661
#define ErcBadLBA       662
#define ErcInvalidDrive 663
#define ErcBadOp        664
#define ErcBadRecal     665
#define ErcSendHDC      666
#define ErcNotReady     667
#define ErcBadCmd       668
#define ErcNeedsInit    669
#define ErcTooManyBlks  670     /* The controller can only do 128 max */
#define ErcZeroBlks     671     /* 0 Blocks not allowed for this cmd */
#define ErcWriteFault   672     /* WriteFault bit set... bummer */

#define ErcMissHDDInt   675

#define ErcHDDMsgBogus  676
```

```
#define ErcHDDIntMsg    677
#define ErcHDDAlarmMsg  678

/* Commands accepted by this HD driver */

#define CmdNull     0
#define CmdRead     1
#define CmdWrite    2
#define CmdVerify   3
#define CmdFmtBlk   4
#define CmdFmtTrk   5
#define CmdSeekTrk  6
#define CmdSetMedia 7   /* Not used unless mountable */
#define CmdResetHdw 8   /* Used to reset controller hardware */

/* CmdReadSect is the only device specific call in the IDE/MFM hard
   disk device driver.  This allows you to read ONE sector
   specified by Cylinder, head and Sector number.
   Cylinder is HiWord of dLBA in DeviceOp call,
   Head is LoWord of dLBA in DeviceOp call, and
   Sector number is LowWord in dnBlocks.
*/

#define CmdReadSect 256 /* only device specific call in HDD */

/* HDC port definitions */

#define HD_PORT 0x1f0

/* When writing to the port+X (where X =):
     0 - write data       (1F0h - 16 bit)
     1 - pre-comp         (1F1h)
     2 - sector count     (1F2h)
     3 - sector number    (1F3h)
     4 - low cyl          (1F4h)
     5 - high cyl         (1F5h)
     6 - size/drive/head  (1F6h)
     7 - command register (1F7h)

When reading from the port+X (where X =):
     0 - read data       (1F0h - 16 bit)
     1 - error register  (1F1h)
     2 - sector count    (1F2h)
     3 - sector number   (1F3h)
     4 - low cyl         (1F4h)
     5 - high cyl        (1F5h)
     6 - size/drive/head (1F6h)
     7 - status register (1F7h)
*/

#define HD_REG_PORT 0x3f6

/* This is a byte wide write only control port
   that allows reset and defines some special
   characteristics of the hard drives.
    Bit     Desc
    0       Not used
    1       Not used
```

continues

Listing 24.2. continued

```
    2       Reset Bit - Set, wait 50us, then Reset
    3       Mucho Heads Flag. Set = More than 8 heads
    4       Not used
    5       Not used
    6       Disable retries
    7       Disable retries (same as six, either one set)
*/

/* HDC Status Register Bit Masks (1F7h) */

#define BUSY        0x80  /* busy.. can't talk now! */
#define READY       0x40  /* Drive Ready  */
#define WRITE_FAULT 0x20  /* Bad news */
#define SEEKOK      0x10  /* Seek Complete */
#define DATA_REQ    0x08  /* Sector buffer needs servicing */
#define CORRECTED   0x04  /* ECC corrected data was read */
#define REV_INDEX   0x02  /* Set once each disk revolution */
#define ERROR       0x01  /* data address mark not found */

/* HDC Error Register Bit Masks (1F1h) */

#define BAD_SECTOR  0x80  /* bad block */
#define BAD_ECC     0x40  /* bad data ecc */
#define BAD_IDMARK  0x10  /* id not found */
#define BAD_CMD     0x04  /* aborted command */
#define BAD_SEEK    0x02  /* trk 0 not found on recalibrate, or bad seek */
#define BAD_ADDRESS 0x01  /* data address mark not found */


/* HDC internal command bytes (HDC_Cmd[7]) */

#define HDC_RECAL      0x10   /* 0001 0000 */
#define HDC_READ       0x20   /* 0010 0000 */
#define HDC_READ_LONG  0x22   /* 0010 0010 */
#define HDC_WRITE      0x30   /* 0011 0000 */
#define HDC_WRITE_LONG 0x32   /* 0011 0010 */
#define HDC_VERIFY     0x40   /* 0100 0000 */
#define HDC_FORMAT     0x50   /* 0101 0000 */
#define HDC_SEEK       0x70   /* 0111 0000 */
#define HDC_DIAG       0x90   /* 1001 0000 */
#define HDC_SET_PARAMS 0x91   /* 1001 0001 */
```

You may notice that I don't initialize any variables in structures because that would place them in a different location in the data segment. If I did one member, I would have to do them all to ensure they would be contiguous. The operating system requires the DCB and status record fields to be contiguous in memory. See Listing 24.3.

Listing 24.3. Continuation of IDE driver data.

```
/* L O C A L   D A T A */

static U8  hd_Cmd[8];       /* For all 8 command bytes */
```

```
static U8  fDataReq;        /* Flag to indicate is fDataRequest is active */
static U8  statbyte;        /* From HDC status register last time it was read */

static U8  hd_control;      /* Current control byte value */
static U8  hd_command;      /* Current Command */
static U8  hd_drive;        /* Current Physical Drive, 0 or 1 */
static U8  hd_head;     /* Calculated from LBA - which head */
static U8  hd_nsectors; /* Calculated from LBA - n sectors to read/write */
static U8  hd_sector;       /* Calculated from LBA - Starting sector */

/* Current type drive 0 & 1 found in CMOS or Set by caller. */
/* Current number of heads, cylinders, and sectors set by caller */

static U8  hd0_type;
static U8  hd0_heads;
static U8  hd0_secpertrk;
static U16 hd0_cyls;

static U8  hd1_type;
static U8  hd1_heads;
static U8  hd1_secpertrk;
static U16 hd1_cyls;

#define sStatus 64

static struct statstruct
 {
  U32 erc;
  U32 blocks_done;
  U32 BlocksMax;
  U8 fNewMedia;
  U8 type_now;      /* current fdisk_table for drive selected */
  U8 resvd0[2];     /* padding for DWord align  */
  U32 nCyl;         /* total physical cylinders */
  U32 nHead;        /* total heads on device    */
  U32 nSectors;     /* Sectors per track        */
  U32 nBPS;         /* Number of bytes per sect.  32 bytes out to here.*/

  U32 LastRecalErc0;
  U32 LastSeekErc0;
  U8  LastStatByte0;
  U8  LastErcByte0;
  U8  fIntOnReset;  /* Interrupt was received on HDC_RESET */
  U8  filler0;

  U32 LastRecalErc1;
  U32 LastSeekErc1;
  U8  LastStatByte1;
  U8  LastErcByte1;
  U8  ResetStatByte;    /* Status Byte immediately after RESET */
  U8  filler1;

  U32 resvd1[2];    /* out to 64 bytes */
  };

static struct statstruct hdstatus;
static struct statstruct HDStatTmp;
```

continues

Listing 24.3. continued

```
static struct dcbtype
{
    S8   Name[12];
    S8   sbName;
    S8   type;
    S16  nBPB;
    U32  last_erc;
    U32  nBlocks;
    S8  *pDevOp;
    S8  *pDevInit;
    S8  *pDevSt;
    U8   fDevReent;
    U8   fSingleUser;
    S16  wJob;
    U32  OS1;
    U32  OS2;
    U32  OS3;
    U32  OS4;
    U32  OS5;
    U32  OS6;
    };

static struct dcbtype hdcb[2];      /* two HD device control blocks */

/* Exch and msgs space for HD ISR */

static U32 hd_exch;

static U32 hd_msg;
static U32 hd_msg2;

static long HDDInt;
```

The `hdisk_setup()` function is called from the monitor to initialize the driver. In a loadable driver, this would be the `main()` section of the C source file. See Listing 24.4.

Listing 24.4. IDE device driver code.

```
U32  hdisk_setup(void)
{
U32  erc;

  /* first we set up the 2 DCBs in anticipation of calling InitDevDr */

    hdcb[0].Name[0]  = 'H';
    hdcb[0].Name[1]  = 'D';
    hdcb[0].Name[2]  = '0';
    hdcb[0].sbName   = 3;
    hdcb[0].type     = 1;        /* Random */
    hdcb[0].nBPB     = 512;
    hdcb[0].nBlocks  = 524288;   /* largest disk handled - 2Gb disks*/
    hdcb[0].pDevOp   = &hddev_op;
    hdcb[0].pDevInit = &hddev_init;
```

```
    hdcb[0].pDevSt   = &hddev_stat;

    hdcb[1].Name[0]  = 'H';
    hdcb[1].Name[1]  = 'D';
    hdcb[1].Name[2]  = '1';
    hdcb[1].sbName   = 3;
    hdcb[1].type     = 1;           /* Random */
    hdcb[1].nBPB     = 512;
    hdcb[1].nBlocks  = 524288;  /* largest device handled - 2Gb disks*/
    hdcb[1].pDevOp   = &hddev_op;
    hdcb[1].pDevInit = &hddev_init;
    hdcb[1].pDevSt   = &hddev_stat;

/* These are defaulted to non zero values to
   ensure we don't get a divide by zero during initial calculations
   on the first read.
*/

    hd0_type =  ReadCMOS(0x19); /* read this but don't use it */
    hd0_heads = 16;          /* Max */
    hd0_secpertrk = 17;      /* most common */
    hd0_cyls = 1024;         /* Max */

    hd1_type =  ReadCMOS(0x1A);
    hd1_heads = 16;
    hd1_secpertrk = 17;
    hd1_cyls = 1024;

    erc = AllocExch(&hd_exch);      /* Exhange for HD Task to use */

    SetIRQVector(14, &hdisk_isr);
    UnMaskIRQ(14);

/* Documentation lists the fixed disk types at CMOS 11h and 12h,
   and also shows them at 19h and 1Ah.  We don't actually read them
   because they are not dependable. They vary from BIOS to BIOS.
   We have to make this sucker work the hard way.
*/

/* Reset the HDC - hd_reset resets the controller (which controls
   both drives). We have to do it once, then try both physical drives.
   If the second drive is not there, some controllers will lock-up
   (the el-cheapos).  In this case we have to reset it again so it
   will work.  It seems like a lot of work, but to make it function
   with the widest range of IDE and MFM controllers this is the
   only way I have found that works.
*/

    hd_reset();     /* no error is returned */

/* Now we attempt to select and recal both drives.
The driver MUST be able to recal the first physical drive
   or the Driver won't inititalize.
*/

    erc = hd_recal(0);          /* try to recal */
    if (erc)
    {                   /* try one more time! */
```

continues

Listing 24.4. continued

```
        hd_reset();
        erc = hd_recal(0);          /* try to recal */
        if (erc)
        {
            hdcb[0].last_erc = erc;
            hd0_type = 0;           /* Must not be a valid drive */
            return(ErcNoDrive0);
        }
    }

    /* if we got here, drive 0 looks OK and the controller is
       functioning.  Now we try drive 1 if type > 0.
    */

    if (hd1_type)
    {
        erc = hd_recal(1);  /* try to recal if CMOS says it's there */
        if (erc)
        {
            hdcb[1].last_erc = erc;
            hd1_type = 0;           /* Guess it's not a valid drive */

            if (!erc)

        /* We must redo drive 0 cause some cheap controllers lockup
        on us if drive 1 is not there.  They SHOULD simply return
        a Bad Command bit set in the Error register, but they don't. */

            hd_reset();
            erc = hd_recal(0);          /* recal drive 0 */
         hdcb[0].last_erc = erc;
        }
    }

    return(erc = InitDevDr(12, &hdcb, 2, 1));

}

/*************************************************************
 Reset the HD controller.  This should only be called by
 DeviceInit or hdisk_setup.  This resets the controller
 and reloads parameters for both drives (if present) and
 attempts to recal them.
*************************************************************/

static void hd_reset(void)
{
U32 i;
    UnMaskIRQ(14);              /*  enable the IRQ */
    OutByte(4, HD_REG_PORT);    /*  reset the controller */
    MicroDelay(4);              /*  Delay 60us */

    /* bit 3 of HD_REG must be 1 for access to heads 8-15 */
    /* Clear "MUCHO" heads bit, and clear the reset bit */

    OutByte(hd_control & 0x0f, HD_REG_PORT);
```

```
    Sleep(20);      /* 200ms - seems some controllers are SLOW!! */
    i = CheckMsg(hd_exch, &hd_msg);     /* Eat Int if one came back  */

    hdstatus.ResetStatByte = statbyte;  /* The ISR gets statbyte */

    if (i) hdstatus.fIntOnReset = 1;
    else hdstatus.fIntOnReset = 0;

}

/*************************************************************
 The ISR is VERY simple. It just waits for an interrupt, gets
 the single status byte from the controller (which clears the
 interrupt condition) then sends an empty message to the
 exchange where the HD Driver task will be waiting.
 This tells the HD task currently running that it's got
 some status to act on!
*************************************************************/
static void interrupt hdisk_isr(void)
{
    statbyte = InByte(HD_PORT+7);
    HDDInt = 1;
    ISendMsg(hd_exch, 0xfffffff0, 0xfffffff0);
    EndOfIRQ(14);
}

/*************************************************************
 This checks the HDC controller to see if it's busy so we can
 send it commands or read the rest of the registers.
 We will wait up to 3 seconds then error out.
 The caller should call check_busy and check the error.
 If it's 0 then the controller became ready in less than
 3 seconds.  ErcNotReady will be returned otherwise.
 It leaves the status byte in the global statbyte.
*************************************************************/

static U32  check_busy(void)
{
S16 count;

  count = 0;
  while (count++ < 60)
  {
    statbyte = InByte(HD_PORT+7);
    if ((statbyte & BUSY) == 0) return(ok);
    Sleep(5);  /* 50ms shots */
  }
  return(ErcNotReady);   /* controller out to lunch! */
}

/*************************************************************
  This sends the SetParams command to the controller to set
  up the drive geometry (nHeads, nSectors, etc.).
*************************************************************/

static U32  hd_init(U8 drive)
{
```

continues

Listing 24.4. continued

```
U32 erc;
  /* set max heads, sectors and cylinders */
  if (drive == 0)
  {                                /* Drive 0 */
    hd_Cmd[2] = hd0_secpertrk;                    /* sector count */
    hd_Cmd[6] = (drive << 4) ¦ ((hd0_heads-1) & 0x0f) ¦ 0xa0; /* hds & drv */
  }
  else
  {                                        /* Drive 1 */
    hd_Cmd[2] = hd1_secpertrk;                    /* sector count */
    hd_Cmd[6] = (drive << 4) ¦ ((hd1_heads-1) & 0x0f) ¦ 0xa0; /* hds & drv */
  }
  hd_Cmd[1] = 0;
  hd_Cmd[3] = 0;
  hd_Cmd[4] = 0;                                  /* cyl = 0 for init */
  hd_Cmd[5] = 0;                                  /* cyl = 0 for init */

  erc = send_command(HDC_SET_PARAMS);       /* Send the command */
  erc = hd_wait();                          /* wait for interrupt */
  if (!erc)
      erc = hd_status(HDC_SET_PARAMS);
  return(erc);
}
```

The `hd_wait()` function has a good example of using the `Alarm()` function. You expect the hard disk controller to come back in a short period of time by sending you a message from the ISR. You also set the `Alarm()` function so you get a message at that exchange, even if the controller goes into Never-Never Land. See Listing 24.5.

Listing 24.5. Continuation of IDE driver code.

```
/*****************************************
Wait for the hardware interrupt to occur.
Time-out and return if no interrupt.
*****************************************/

static U32  hd_wait(void)
{
U32  erc;

    /* Set alarm for 3 seconds */

    HDDInt = 0;
    KillAlarm(hd_exch);          /* kill any pending alarm */

    erc = Alarm(hd_exch, 300);  /* Set it up again */
    if (erc)
        return(erc);         /* bad problem */

    erc = WaitMsg(hd_exch, &hd_msg);

    KillAlarm(hd_exch);
```

```
    if (hd_msg != 0xfffffff0)
    {       /* HD interrupt sends fffffff0 */
        if (HDDInt)
          return(ErcMissHDDInt);
        else
            return(ErcHDCTimeOut);      /* Alarm sends 0xffffffff      */
    }
    else
    {
        KillAlarm(hd_exch);
        return(ok);
    }
}

/*****************************************
    Recalibrate the drive.
*****************************************/

static U32  hd_recal(U8 drive)
{
U32 erc;

  hd_Cmd[6] = (drive << 4) ¦ (hd_head & 0x0f) ¦ 0xa0;
  erc = send_command(HDC_RECAL);
  if (!erc)
    erc = hd_wait();                        /* wait for interrupt */
  if (!erc)
     erc = hd_status(HDC_RECAL);
  if (drive)
    hdstatus.LastRecalErc1 = erc;
  else
    hdstatus.LastRecalErc0 = erc;
  return(erc);
}


/*****************************************
    Send the command to the controller.
    Clear the Exchange of any left over
    alarm or int messages before we
    send a command.
*****************************************/

static U32  send_command(U8  Cmd)
{
U32 erc, msg[2];

    while (CheckMsg(hd_exch, &msg) == 0);   /* Empty it */

    /* bit 3 of HD_REG must be 1 for access to heads 8-15 */
    if (hd_head > 7)
    {
      hd_control ¦= 0x08;
      OutByte(hd_control, HD_REG_PORT);     /*  set bit for head > 7 */
      hd_control &= 0xf7;
    }
    erc = check_busy();
    if (!erc) OutByte(hd_Cmd[1], HD_PORT+1);
```

continues

Listing 24.5. continued

```
    if (!erc) erc = check_busy();
    if (!erc) OutByte(hd_Cmd[2], HD_PORT+2);
    if (!erc) erc = check_busy();
    if (!erc) OutByte(hd_Cmd[3], HD_PORT+3);
    if (!erc) erc = check_busy();
    if (!erc) OutByte(hd_Cmd[4], HD_PORT+4);
    if (!erc) erc = check_busy();
    if (!erc) OutByte(hd_Cmd[5], HD_PORT+5);
    if (!erc) erc = check_busy();
    if (!erc) OutByte(hd_Cmd[6], HD_PORT+6);
    if (!erc) erc = check_busy();
    if (!erc) OutByte(Cmd, HD_PORT+7);
    return(erc);
}

/*************************************************************
 This sets up the cylinder, head and sector variables for all
 commands that require them (read, write, verify, format, seek).
 nBlks  CANNOT be greater than the hardware can handle. For
 IDE/MFM controllers this is 128 sectors.
 The calculated values are placed in the proper command byte
 in anticipation of the command being sent.
*************************************************************/

static U32  setupseek(U32 dLBA, U32 nBlks)
{
 U32  j;
 U16  cyl;

  if (nBlks > 256) return ErcTooManyBlks;
  if (nBlks == 0) return ErcZeroBlks;

  hd_nsectors = nBlks;
  if (hd_nsectors == 256) hd_nsectors = 0;   /* 0==256 for controller */

  if (hd_drive == 0)
  {     /* drive 0 */

    cyl = dLBA / (hd0_heads * hd0_secpertrk);
    j = dLBA % (hd0_heads * hd0_secpertrk);     /* remainder */

    /* we now know what cylinder, calculate head and sector */

    hd_head = j / hd0_secpertrk;
    hd_sector = j % hd0_secpertrk + 1; /* sector number start at 1 !!! */

  }
  else
  {                     /* drive 1 */

    cyl = dLBA / (hd1_heads * hd1_secpertrk);
    j = dLBA % (hd1_heads * hd1_secpertrk);     /* remainder */

    /* We now know what cylinder. Calculate head and sector */

    hd_head = j / hd1_secpertrk;
```

```
    hd_sector = j % hd1_secpertrk + 1; /* sector number start at 1 !!! */
  }

  hd_Cmd[2] = nBlks;                    /* How many sectors */
  hd_Cmd[3] = hd_sector;                /* Which sector to start on */
  hd_Cmd[4] = cyl & 0xff;               /* cylinder lobyte */
  hd_Cmd[5] = (cyl >> 8) & 0xff;        /* cylinder hibyte */
  hd_Cmd[6] = (hd_drive << 4) | (hd_head & 0x0f) | 0xa0;

  return ok;
}


/*****************************************************
  Move the head to the selected track (cylinder).
*****************************************************/

static U32  hd_seek(U32 dLBA)
{
U32 erc;

  erc = setupseek(dLBA, 1);                 /* sets up for implied seek */
  if (!erc) erc = send_command(HDC_SEEK);   /* Not implied anymore... */
  if (!erc) erc = hd_wait();                /* wait for interrupt */
  if (!erc) erc = hd_status(HDC_SEEK);
  hdstatus.LastSeekErc0 = erc;
  return(erc);
}


/*****************************************************
 Called to read status and errors from the controller
 after an interrupt generated by a command we sent.
 The error checking is based on the command that we sent.
 This is done because certain bits in the status and error
 registers are actually not errors, but simply indicate
 status or indicate an action we must take next.
 ZERO returned indicates no errors for the command status
 we are checking.
*****************************************************/

static U32  hd_status(U8  LastCmd)
{
U32 erc;
U8  statbyte, errbyte;

  /* We shouldn't see the controller busy. After all,
     he interrupted us with status.
  */

  erc = check_busy();       /* puts status byte into global StatByte */
  if (!erc)
    statbyte = InByte(HD_PORT+7);
  else return(erc);

  if (hd_drive)
      hdstatus.LastStatByte1 = statbyte;
```

continues

Listing 24.5. continued

```
  else
      hdstatus.LastStatByte0 = statbyte;

  if ((statbyte & ERROR) == 0)
  {     /* Error bit not set in status reg */
    erc = ok;  /* default */

    switch (LastCmd)
    {
      case HDC_READ:
      case HDC_READ_LONG:
      case HDC_WRITE:
      case HDC_WRITE_LONG:
      case HDC_SEEK:
      case HDC_RECAL:
        if (statbyte & WRITE_FAULT) erc = ErcWriteFault;
        else
            if ((statbyte & SEEKOK) == 0) erc = ErcBadSeek;
        break;
      case HDC_SET_PARAMS:
      case HDC_VERIFY:
      case HDC_FORMAT:
      case HDC_DIAG:
        break;
      default:
        break;
    }
    return(erc);
    }
  else
  {
    erc = check_busy();
    if (!erc)
      errbyte = InByte(HD_PORT+1);
    else return(erc);

    if (hd_drive)
        hdstatus.LastErcByte1 = errbyte;
    else
        hdstatus.LastErcByte0 = errbyte;

    if (errbyte & BAD_ADDRESS) erc = ErcAddrMark;
    else if (errbyte & BAD_SEEK) erc = ErcBadSeek;
    else if (errbyte & BAD_CMD) erc = ErcBadCmd;
    else if (errbyte & BAD_IDMARK) erc = ErcSectNotFound;
    else if (errbyte & BAD_ECC) erc = ErcBadECC;
    else if (errbyte & BAD_SECTOR) erc = ErcBadBlock;
    else erc = ErcBadHDC; /* no error bits found but should have been! */
  }
 return erc;
}

/*************************************************************
 This is called for the DeviceOp code Read.
 This reads 1 or more whole sectors from the calculated values
 in hd_head, hd_sector, and hd_cyl
```

```
*************************************************************/

static U32  hd_read(U32 dLBA, U32 dnBlocks, U8 *pDataRet)
{
U32 erc, nleft, nBPS;

  nBPS = hdcb[hd_drive].nBPB;           /* From nBytesPerBlock in DCB */
  nleft = dnBlocks;
  erc = setupseek(dLBA, dnBlocks);      /* sets up for implied seek */
  if (!erc) erc = send_command(HDC_READ);

  while ((nleft) && (!erc))
  {
      erc = hd_wait();                  /* wait for interrupt */
      if (!erc)
        erc = hd_status(HDC_READ);
      if (!erc)     /* && (statbyte & DATA_REQ))  */
      {
        InWords(HD_PORT, pDataRet, nBPS);
        pDataRet+=nBPS;
        --nleft;
      }
  }
  return(erc);
}

/*************************************************************
 This is called for the DeviceOp code Write.
 This writes 1 or more whole sectors from the calculated values
 in hd_head, hd_sector, and hd_cyl
*************************************************************/

static U32  hd_write(U32 dLBA, U32 dnBlocks, U8 *pDataOut)
{
U32 erc, nSoFar, nBPS;

  nBPS = hdcb[hd_drive].nBPB;           /* From n BytesPerBlock in DCB */
  nSoFar = 0;
  erc = setupseek(dLBA, dnBlocks);      /* sets up for implied seek */
  erc = send_command(HDC_WRITE);
  erc = check_busy();           /* No INT occurs for first sector of write */

  if ((!erc) && (statbyte & DATA_REQ))
  {
    OutWords(HD_PORT, pDataOut, nBPS);
    pDataOut+=nBPS;
    nSoFar++;
  }

  while ((nSoFar < dnBlocks ) && (erc==ok))
  {
      erc = hd_wait();                  /* wait for interrupt */
      if (erc==ok) erc = hd_status(HDC_WRITE);
      if ((erc==ok) && (statbyte & DATA_REQ))
      {
          OutWords(HD_PORT, pDataOut, nBPS);
          pDataOut+=nBPS;
```

continues

Listing 24.5. continued

```
            nSoFar++;
        }
    }
    if (!erc) erc = hd_wait();              /* wait for final interrupt */
    if (!erc) erc = hd_status(HDC_WRITE);

    return(erc);
}


/*************************************************************
 This formats the track beginning at the block address given
 in dLBA. dLBA must always be a multiple of the number of
 sectors per track minus 1 for the disk type.
*************************************************************/

static U32  hd_format_track(U32 dLBA, U32 dnBlocks)
{
U32  erc;
    erc = setupseek(dLBA, dnBlocks);        /* sets up for implied seek */
    erc = send_command(HDC_FORMAT);
    erc = hd_wait();                        /* wait for interrupt */
    if (erc==ok)
        erc = hd_status(HDC_FORMAT);
    return(erc);
}

/*****************************************************************
   ReadSector is the only device specific call in the IDE/MFM hard
   disk device driver.  This allows you to read ONE sector
   specified by Cylinder, head and Sector number.
   Cylinder is LoWord of dLBA in DeviceOp call,
   Head is LoWord of dnBlocks in DeviceOp call, and
   Sector number is HiWord in dnBlocks.
*****************************************************************/

static U32 ReadSector(U32 Cylinder, U32 HdSect, U8 *pDataRet)
{
U32 erc;
U16  cyl;

    cyl = Cylinder;
    hd_head = HdSect & 0xffff;
    hd_sector = (HdSect >> 16) & 0xffff;

/*  For testing
 xprintf("\r\nCYL %d, HD %d, SEC %d\r\n", cyl, hd_head, hd_sector);
*/

    hd_Cmd[2] = 1;                          /* How many sectors */
    hd_Cmd[3] = hd_sector;                  /* Which sector to start on */
    hd_Cmd[4] = cyl & 0xff;                 /* cylinder lobyte */
    hd_Cmd[5] = (cyl >> 8) & 0xff;          /* cylinder hibyte */
    hd_Cmd[6] = (hd_drive << 4) ¦ (hd_head & 0x0f) ¦ 0xa0;

    erc = send_command(HDC_READ);
    erc = hd_wait();                    /* wait for interrupt */
```

```
  if (!erc) erc = hd_status(HDC_READ);
  if (!erc)
    InWords(HD_PORT, pDataRet, 512);
  return(erc);
}
```

Chapter 8, "Programming Interfaces," discussed device driver interfaces; there I provided an overview of MMURTL's device driver interface. The three calls that all MMURTL device drivers must provide are next. See Listing 24.6.

Listing 24.6. Continuation of IDE driver code.

```
/*****************************************
Called for all device operations.  This
assigns physical device from logical number
that outside callers use. For Hard disk,
12=0 and 13=1. This will check to make sure a
drive type is assigned and check to see if
they are going to exceed max logical blocks.
*****************************************/

static U32  hddev_op(U32  dDevice,
             U32  dOpNum,
             U32  dLBA,
             U32  dnBlocks,
             U8  *pData)
{
U32  erc;

 hdstatus.blocks_done = 0;  /* Reset values in Status record */

 erc = 0;

 /* Set drive internal drive number */

 if (dDevice == 12)
    hd_drive = 0;
 else
    hd_drive = 1;

 /* Check to see if we have a leftover interrupt message from last
    command.  If so then we eat it (and do nothing) */

  CheckMsg(hd_exch, &hd_msg);       /* Ignore error */

 if (hd_drive==0)
 {
    if (hd0_type==0)
        erc = ErcInvalidDrive;
 }
 else
 {
    if (hd1_type==0)
        erc = ErcInvalidDrive;
```

continues

Listing 24.6. continued

```
}

/* make sure they don't exceed max blocks */

if (!erc)
  if (dLBA > hdcb[hd_drive].nBlocks) erc = ErcBadLBA;

if (!erc)
{

  switch(dOpNum)
  {
     case(CmdNull):
       erc = ok;                    /* Null Command */
       break;
     case(CmdRead):                                    /* Read */
       erc = hd_read(dLBA, dnBlocks, pData);
       break;
     case(CmdWrite):                                   /* Write */
       erc = hd_write(dLBA, dnBlocks, pData);
       break;
     case(CmdVerify):                                  /* Verify */
       erc = ErcNotSupported;

       /* hd_verify is not supported in this version of the driver */
       /*      erc = hd_verify(dLBA, dnBlocks, pData); */
       break;
     case(CmdSeekTrk):                                 /* Seek Track */
       erc = hd_seek(dLBA);
       break;
     case(CmdFmtTrk):                                  /* Format Track */
       erc = hd_format_track(dLBA, dnBlocks);
       break;
     case(CmdResetHdw):                                /* Reset Ctrlr */
       hd_reset();
       erc = 0;
       break;
     case(CmdReadSect):                                /* Read Sector(s) */
       erc = ReadSector(dLBA, dnBlocks, pData);
       break;
     default:
       erc = ErcBadOp;
       break;
   }
 }
 hdcb[hd_drive].last_erc = erc;          /* update DCB erc */
 return(erc);
}

/*****************************************
Called for indepth status report on ctrlr
and drive specified. Returns 64 byte block
of data including current drive geometery.
This is called by the PUBLIC call DeviceStat!
*****************************************/
```

```
static U32 hddev_stat(U32  dDevice,
              S8 * pStatRet,
              U32  dStatusMax,
              U32  *pdStatusRet)
{
U32 i;

 /* Set status for proper device */

 if (dDevice == 12)
 {
    hdstatus.erc = hdcb[0].last_erc;
    hdstatus.type_now = hd0_type;
    hdstatus.nCyl = hd0_cyls;
    hdstatus.nHead = hd0_heads;
    hdstatus.nSectors = hd0_secpertrk;
    hdstatus.nBPS = hdcb[0].nBPB;
 }
 else
 {
    hdstatus.erc = hdcb[1].last_erc;
    hdstatus.type_now = hd1_type;
    hdstatus.nCyl = hd1_cyls;
    hdstatus.nHead = hd1_heads;
    hdstatus.nSectors = hd1_secpertrk;
    hdstatus.nBPS = hdcb[1].nBPB;
 }

 /* Calculate size of status to return. Return no more than asked for! */

 if (dStatusMax <= sStatus) i = dStatusMax;
 else i = sStatus;

 CopyData(&hdstatus, pStatRet, i);      /* copy to their status block */

 *pdStatusRet = i;                      /* tell em how much it was */

 return ok;
}

/*****************************************
Called to reset the hard disk controller
and set drive parameters.  The Initdata
is a copy of the 64 byte status block
that is read from status.  The caller
normally reads the block (DeviceStat),
makes changes to certain fields and
calls DeviceInit pointing to the block
for the changes to take effect.
This should ONLY be called once for each HD
to set its parameters before it is used
the first time after the driver is loaded,
or after a fatal error is received that
indicates the controller may need to be
reset (multiple timeouts etc.).
The DCB values are updated if this is
successful.
```

continues

Listing 24.6. continued

```
This is called by the PUBLIC call DeviceInit.
*****************************************/

static U32  hddev_init(U32  dDevice,
             S8  *pInitData,
             U32   sdInitData)

{
U32 erc, i;

  erc = 0;

 /* Read the init status block in */

 if (sdInitData > sStatus) i = sStatus;     /* no more than 64 bytes! */
 else i = sdInitData;

 CopyData(pInitData, &HDStatTmp, i);        /* copy in their init data */

 /* Set internal drive number */
 if (dDevice == 12) hd_drive=0;
 else hd_drive = 1;

 if (hd_drive==0)
 {
    hd0_type  = HDStatTmp.type_now;
    if (hd0_type)
    {
      hd0_cyls  = HDStatTmp.nCyl;
      hd0_heads = HDStatTmp.nHead;
      hd0_secpertrk = HDStatTmp.nSectors;
    }
    else erc = ErcInvalidDrive;
 }
 else
 {
    hd1_type  = HDStatTmp.type_now;
    if (hd1_type)
    {
        hd1_cyls  = HDStatTmp.nCyl;
        hd1_heads = HDStatTmp.nHead;
        hd1_secpertrk = HDStatTmp.nSectors;
    }
    else erc = ErcInvalidDrive;
 }

/* If no error, initialize it and recal */

 if (!erc) erc = hd_init(hd_drive);
 if (!erc) erc = hd_recal(hd_drive);

/* If no error, update corresponding DCB values */

 if (!erc)
 {
    hdcb[hd_drive].nBPB = HDStatTmp.nBPS;
```

```
    hdcb[hd_drive].last_erc = 0;
    hdcb[hd_drive].nBlocks =
              HDStatTmp.nCyl * HDStatTmp.nSectors * HDStatTmp.nHead;
 }

 hdcb[hd_drive].last_erc = erc;          /* update DCB erc */

 return(erc);
}
```

The RS-232 Asynchronous Device Driver

The RS-232 driver is designed to drive two channels with 8250, 16450, or 16550 UARTs. It needs more work, such as complete flow control for XON/XOFF and CTS/RTS; but other than that, it is a fully functional driver.

You may notice the differences between the IDE driver, which is a block-oriented random device, and this driver, which is a sequential byte-oriented device. They are two different animals with the same basic interface.

Unlike the IDE disk device driver, a header file is included in the RS232 driver source file. This header file was also designed to be used by programs that use the driver, as well as by the driver itself. This same header file is used in the sample communications program in Chapter 16, "MMURTL Sample Software."

RS232.h

The header file defines certain parameters to functions, as well as explains all the error or status codes that the driver will return.

The C structure for the status record is defined here and will be needed to initialize, as well as status, the driver. See Listing 24.7.

Listing 24.7. RS-232.h header file.

```
#define MIN_BAUD        1501
#define MAX_BAUD        384001
#define NO_PAR             0
#define EV_PAR             1
#define OD_PAR             2

#define ErcRecvTimeout  800     /* Recv Buffer Empty */
#define ErcXmitTimeout  801     /* Xmit Buffer never Emptied */
#define ErcRcvBufOvr    802     /* Receive buffer overrun */
#define ErcBadPort      803     /* Invalid port on OpenCommC */
#define ErcRcvBufOvr    805     /* Buffer full!!! */
#define ErcNotOpen      807     /* Channel not open... */
```

continues

Listing 24.7. continued

```
#define ErcChannelOpen  809     /* It's already open... */
#define ErcNotOwner     812     /* It's opened by someone else... */
#define ErcBadBaud      820     /* 150-38400       */
#define ErcBadParity    821     /* 0, 1 or 2       */
#define ErcBadDataBits  822     /* Must be 5-8     */
#define ErcBadStopBits  823     /* Must be 1 or 2 */
#define ErcBadIOBase    824     /* if 0            */
#define ErcBadCommIRQ   825     /* < 3             */
#define ErcBadInitSize  827     /* At least 40 bytes for this version */


struct statRecC{
  unsigned long commJob;    /* Owner of this comms port, 0 for not in use */
  unsigned long LastErc;    /* Result of last device operation */
  unsigned long LastTotal;  /* Total bytes moved in last operation */
  unsigned long Baudrate;   /* Baudrate for this port, 150 - 38400 */
unsigned char parity;     /* Parity for this port, 0=none, 1=even, 2=odd */
  unsigned char databits;   /* nDatabits for this port, 5-8 */
  unsigned char stopbits;   /* stop bits for this port, 1 or 2 */
  unsigned char IRQNum;     /* IRQNum for this channel */
  unsigned long IOBase;     /* IO base address for hardware */
  unsigned long XBufSize;   /* Size of Xmit buffer */
  unsigned long RBufSize;   /* Size of Recv Buffer */
  unsigned long XTimeOut;   /* Xmit Timeout in 10ms increments */
  unsigned long RTimeOut;   /* Recv Timeout in 10ms increments */
  unsigned long resvd[6];   /* out to 64 bytes */
  };

/* Device Driver interface commands (Op numbers) */

#define CmdReadRec   1          /* Read one or more bytes */
#define CmdWriteRec  2          /* Write one or more bytes */
#define CmdOpenC     10         /* Open Comm Channel */
#define CmdCloseC    11         /* Close Comm Channel */
#define CmdDiscardRcv 12        /* Trash input buffer */
#define CmdSetRTO    13         /* Set Recv timeout 10ms incs in dLBA */
#define CmdSetXTO    14         /* Set Xmit timeout 10ms incs in dLBA */
#define CmdSetDTR    15         /* Set DTR (On) */
#define CmdSetRTS    16         /* Set CTS (On) */
#define CmdReSetDTR 17          /* Set DTR (On) */
#define CmdReSetRTS 18          /* Set CTS (On) */
#define CmdBreak     19         /* Send BREAK (10ms incs in dLBA) */
#define CmdGetDC     20         /* Returns byte TRUE to pData if CD ON */
#define CmdGetDSR    21         /* Returns byte TRUE to pData if DSR ON */
#define CmdGetCTS    22         /* Returns byte TRUE to pData if CTS ON */
#define CmdGetRI     23         /* Returns byte TRUE to pData if RI ON */
#define CmdReadB     31         /* Recv a single byte */
#define CmdWriteB    32         /* Xmit a single byte */
```

The Device Driver, RS-232.C

In the following code, you again will notice the shorthand for the rather lengthy C-type declarations. Further down in the file you will notice that commands are defined 1 through 32.

The only commands that are shared with the default device driver commands are for reading and writing blocks of data. These are commands 1 and 2. All other commands are specific to this driver. See Listing 24.8.

Listing 24.8. RS-232 device driver code.

```
#define U32 unsigned long
#define S32 long
#define U16 unsigned int
#define S16 int
#define U8 unsigned char
#define S8 char
#define TRUE   1
#define FALSE  0

#include "RS232.h"

/* MMURTL OS Prototypes */

extern far U32 AllocExch(U32 *pExchRet);

extern far U32 InitDevDr(U32 dDevNum,
                         S8  *pDCBs,
                         U32 nDevices,
                         U32 dfReplace);

extern far U32 AllocOSPage(U32 nPages, U8 **ppMemRet);
extern far U32 DeAllocPage(U8 *pOrigMem, U32 nPages);

extern far U32 UnMaskIRQ(U32 IRQNum);
extern far U32 MaskIRQ(U32 IRQNum);
extern far U32 SetIRQVector(U32 IRQNum, S8  *pIRQ);
extern far U32 EndOfIRQ(U32 IRQNum);
extern far U32 SendMsg(U32 Exch, U32 msg1, U32 msg2);
extern far U32 ISendMsg(U32 Exch, U32 msg1, U32 msg2);
extern far U32 WaitMsg(U32 Exch, S8  *pMsgRet);
extern far U32 CheckMsg(U32 Exch, S8  *pMsgRet);
extern far U32 GetTimerTick(U32 *pTickRet);
extern far U32 Alarm(U32 Exch, U32 count);
extern far U32 KillAlarm(U32 Exch);
extern far U32 Sleep(U32 count);
extern far void MicroDelay(U32 us15count);
extern far void OutByte(U8 Byte, U16 wPort);
extern far U8   InByte(U16 wPort);
extern far void CopyData(U8 *pSource, U8 *pDestination, U32 dBytes);

extern far long GetJobNum(long *pJobNumRet);

/* local prototypes. These will be called from the device driver interface. */

static U32 comdev_stat(U32  dDevice,
             S8  *pStatRet,
             U32 dStatusMax,
             U32 *pdStatusRet);

static S32 comdev_init(U32  dDevice,
```

continues

Listing 24.8. continued

```
            S8  *pInitData,
            U32  sdInitData);

static U32 comdev_op(U32 dDevice,
           U32 dOpNum,
           U32 dLBA,
           U32 dnBlocks,
           U8  *pData);


#define CmdReadRec   1              /* Read one or more bytes */
#define CmdWriteRec  2              /* Write one or more bytes */

#define CmdOpenC     10             /* Open Comm Channel */
#define CmdCloseC    11             /* Close Comm Channel */
#define CmdDiscardRcv 12
#define CmdSetRTO    13
#define CmdSetXTO    14
#define CmdSetDTR    15             /* Set DTR (On) */
#define CmdSetRTS    16             /* Set CTS (On) */
#define CmdReSetDTR 17              /* Set DTR (On) */
#define CmdReSetRTS 18              /* Set CTS (On) */
#define CmdBreak     19
#define CmdGetDC     20
#define CmdGetDSR    21
#define CmdGetCTS    22
#define CmdGetRI     23

#define CmdReadB     31
#define CmdWriteB    32

/* The following definitions are used to identify, set
   and reset signal line condition and functions.
*/

#define CTS          0x10        /* Clear To Send       */
#define DSR          0x20        /* Data Set Ready      */
#define RI           0x40        /* Ring Indicator      */
#define CD           0x80        /* Carrier Detect      */

#define DTR          0x01        /* Data Terminal Ready */
#define RTS          0x02        /* Request To Send     */
#define OUT2         0x08        /* Not used */

/* Values from IIR register */

#define MDMSTAT      0x00
#define NOINT        0x01
#define TXEMPTY      0x02
#define RVCDATA      0x04
#define RCVSTAT      0x06

/* For errors returned from Comms calls see RS-232.h */

#define SSENDBUF 4096       /* 1 Page Send Buf Default */
#define SRECVBUF 4096       /* 1 Page Recv Buf Default */
```

```
static U32  recv_timeout[2] = 1;    /* 10ms intervals */
static U32  xmit_timeout[2] = 10;   /* 10ms intervals */

/* variables for ISRs */

static U8   f16550[2];
static U8   stat_byte[2];
static U8   mstat_byte[2];
static U8   int_id[2];
static U8   fExpectInt[2];

static U32 recv_error[2];   /* NON-ZERO = an error has occurred */

static U8   *pSendBuf[2];   /* pointers to Xmit bufs */
static U32  head_send[2];   /* Next char to send */
static U32  tail_send[2];   /* Where next char goes in buf */
static U32  cSendBuf[2];    /* Count of bytes in buf */
static U32  sSendBuf[2];    /* Size of buffer (allocated) */

static U8   *pRecvBuf[2];   /* pointers to Recv bufs */
static U32  head_recv[2];   /* Next char from chip */
static U32  tail_recv[2];   /* Next char to read for caller */
static U32  cRecvBuf[2];    /* Count of bytes in buf */
static U32  sRecvBuf[2];    /* Size of buffer (allocated) */

static U8   control_byte[2] = 0;

 /* array of registers from port base for each channel */

static U16      THR[2];     /* Transmitter Holding Register */
static U16      IER[2];     /* Interrupt Enable Register */
static U16      IIR[2];     /* Interrupt Id Register */
static U16      FCR[2];     /* FIFO control for 16550 */
static U16      LCR[2];     /* Line Control Register */
static U16      MCR[2];     /* Modem Control Register */
static U16      LSR[2];     /* Line Status Register */
static U16      MSR[2];     /* Modem Status Register */
static U16      DLAB_LO[2]; /* same address as THR  */
static U16      DLAB_HI[2];   /* same address as IER  */
```

I saw register bits defined like those in Listing 24.9 in some documentation I saw while working on an AM65C30 USART. I liked it very much because it was easy to read in a bit-wise fashion. You can easily see the break-out of the bits and follow what is being tested or written throughout the source code.

Listing 24.9. RS-232 device driver code continued.

```
/* Complete description of Register bits follows:

    THR — TX data, RX data, Divisor Latch LSB

    IER — Interrupt Enable, Divisor Latch MSB
```

continues

Listing 24.9. continued

```
                7 6 5 4 3 2 1 0
                | | | | | | | |  \_ Data Available
                | | | | | | |  \___ Xmit Holding Reg Empty
                | | | | | |  \_____ Receiver Line Status
                | | | | |  \_______ Modem Status
\ _ _ _\_________ Always 0000
```

IIR — Interrupt Identification Register

```
                7 6 5 4 3 2 1 0
                | | | | \_\_\_\__________ 0001 = no interrupt
                | | | |                    0110 = rcvr status
                | | | |                    0100 = rcvd data
                | | | |                    0010 = THR empty
                | | | |                    0000 = Modem status
                | | | |                    1101 = Rcv Fifo Timeout (16550)
                | | |  \_________ = 0.
                | |  \___________ = 0.
                 \ \_____________ = (16550 only)
                                    00 = FIFOs disabled
                                    non-zero = 16550 enabled
```

FCR — FIFO Control Register (16550 only)
This is a write only port at the same address as the IIR on an 8250/16450

```
                7 6 5 4 3 2 1 0
                | | | | | | | |  \__= 1 = FIFO Enable
                | | | | | | |  \___ = 1 = Recv FIFO Reset
                | | | | | |  \_____ = 1 = Xmit FIFO Reset
                | | | | |  \_______ = (DMA) 0 = Single char, 1 = Multichar
                | | |  \_________ = 0.
                | |  \___________ = 0.
                \__\_____________ = FIFO Rcv Trigger Level
                                    00 - 1 char
                                    01 - 2 chars
                                    10 - 4 chars
                                    11 - 8 chars
```

LCR — Line Control Register

```
                7 6 5 4 3 2 1 0
                | | | | | | | |  \_ Word Length Select Bit 0.
                | | | | | | |  \___ Word Length Select Bit 1.
                | | | | | |  \_____ Number Stop Bits (0=1, 1=2)
                | | | | |  \_______ Parity Enable
                | | |  \_________ Even Parity Select
                | |  \___________ Stick Parity
                |  \_____________ Set Break
                 \_______________ Divisor Latch Access Bit
```

```
    MCR — Modem Control Register

                  7 6 5 4 3 2 1 0
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦  \_ Data Terminal Ready
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦  \___ Request to Send
                  ¦ ¦ ¦ ¦ ¦ ¦  \_____ Out 1
                  ¦ ¦ ¦ ¦ ¦  \_______ Out 2  (= 1 to enable ints.)
                  ¦ ¦ ¦ ¦  \_________ Loop
\ _ _\____________ = Always 0

    LSR — Line Status Register

                  7 6 5 4 3 2 1 0
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦  \_ Data Ready
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦  \___ Overrun Error
                  ¦ ¦ ¦ ¦ ¦ ¦  \_____ Parity Error
                  ¦ ¦ ¦ ¦ ¦  \_______ Framing Error
                  ¦ ¦ ¦ ¦  \_________ Break interrupt
                  ¦ ¦ ¦  \___________ Transmitter Holding Reg Empty
                  ¦ ¦  \_____________ Transmitter Shift Reg Empty
                  ¦  \_______________ Recv FIFO Error (16550 Only)

    MSR — Modem Status Register

                  7 6 5 4 3 2 1 0
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦  \_ Delta Clear to Send
                  ¦ ¦ ¦ ¦ ¦ ¦ ¦  \___ Delta Data Set Ready
                  ¦ ¦ ¦ ¦ ¦ ¦  \_____ Trailing Edge Ring Indicator
                  ¦ ¦ ¦ ¦ ¦  \_______ Delta Rx Line Signal Detect
                  ¦ ¦ ¦ ¦  \_________ Clear to Send
                  ¦ ¦ ¦  \___________ Data Set Ready
                  ¦ ¦  \_____________ Ring Indicator
                  ¦  \_______________ Receive Line Signal Detect

*/
```

The status record for each driver is specific to that particular driver. The members of this structure are defined in the header file RS232.h. It is documented in the preceding section in this chapter. See Listing 24.10.

Listing 24.10. RS-232 device driver code continued.

```
/* Record for 64 byte status and init record */
/* This structure is peculiar to the comms driver */

#define sStatus 64
static struct statRecC comstat[2];
static struct statRecC *pCS;
```

This is the same device control block (DCB) record structure used in all MMURTL device drivers. Note that there are two DCBs. There is one for each channel, and they are contiguously defined, as is required by MMURTL. See Listing 24.11.

Listing 24.11. RS-232 device driver code continued.

```
static struct dcbtype
{
    S8   Name[12];
    S8   sbName;
    S8   type;
    S16  nBPB;
    U32  last_erc;
    U32  nBlocks;
    S8  *pDevOp;
    S8  *pDevInit;
    S8  *pDevSt;
    S8   fDevReent;
    S8   fSingleUser;
    S16  wJob;
    U32  OS1;
    U32  OS2;
    U32  OS3;
    U32  OS4;
    U32  OS5;
    U32  OS6;
    };

static struct dcbtype comdcb[2];        /* Two RS-232 ports */


/* THE COMMS INTERRUPT FUNCTION PROTOTYPES */

static void interrupt comISR0(void);
static void interrupt comISR1(void);
```

Just as with the IDE hard disk device driver, the following initialization routine is called once to set up the driver. The routine calls `InitDevDr()` after it has completed filling out the device control blocks and setting up the interrupts. This means it's ready. See Listing 24.12.

Listing 24.12. RS-232 device driver code continued.

```
/*********************************************************
    This is called ONCE to initialize the 2 default
    comms channels with the OS device driver interface.
    It sets up defaults for both channels to 9600, 8N1.
*********************************************************/

U32  coms_setup(void)
{
U32  erc;
```

```
/* first we set up the 2 DCBs in anticipation of calling InitDevDr */

  comdcb[0].Name[0]  = 'C';
  comdcb[0].Name[1]  = 'O';
  comdcb[0].Name[2]  = 'M';
  comdcb[0].Name[2]  = '1';
  comdcb[0].sbName   = 4;
  comdcb[0].type     = 2;          /* Sequential */
  comdcb[0].nBPB     = 1;          /* 1 byte per block */
  comdcb[0].nBlocks  = 0;          /* 0 for Sequential devices */
  comdcb[0].pDevOp   = &comdev_op;
  comdcb[0].pDevInit = &comdev_init;
  comdcb[0].pDevSt   = &comdev_stat;

  comdcb[1].Name[0]  = 'C';
  comdcb[1].Name[1]  = 'O';
  comdcb[1].Name[2]  = 'M';
  comdcb[1].Name[2]  = '2';
  comdcb[1].sbName   = 4;
  comdcb[1].type     = 2;          /* Sequential */
  comdcb[1].nBPB     = 1;          /* 1 byte per block */
  comdcb[1].nBlocks  = 0;          /* 0 for Sequential devices */
  comdcb[1].pDevOp   = &comdev_op;
  comdcb[1].pDevInit = &comdev_init;
  comdcb[1].pDevSt   = &comdev_stat;

  /* Set default comms params in stat records */

  comstat[0].Baudrate = 9600;
  comstat[0].parity   = 0;    /* none */
  comstat[0].databits = 8;
  comstat[0].stopbits = 1;
  comstat[0].XTimeOut = 100;
  comstat[0].RTimeOut = 2;
  comstat[0].IOBase = 0x3F8;
  comstat[0].IRQNum = 4;
  comstat[0].XBufSize = 4096;
  comstat[0].RBufSize = 4096;
  sSendBuf[0] = 4096;
  sRecvBuf[0] = 4096;

  comstat[1].Baudrate = 9600;
  comstat[1].parity   = 0;    /* none */
  comstat[1].databits = 8;
  comstat[1].stopbits = 1;
  comstat[1].XTimeOut = 100;
  comstat[1].RTimeOut = 2;
  comstat[1].IOBase = 0x2F8;
  comstat[1].IRQNum = 3;
  comstat[1].XBufSize = 4096;
  comstat[1].RBufSize = 4096;
  sSendBuf[1] = 4096;
  sRecvBuf[1] = 4096;

  MaskIRQ(4);
  MaskIRQ(3);
```

continues

Listing 24.12. continued

```
    SetIRQVector(4, &comISR0);  /* COM1 */
    SetIRQVector(3, &comISR1);  /* COM2 */

    return(erc = InitDevDr(5, &comdcb, 2, 1));

}
```

The following function does all the work for the interrupt service routine functions, one for each channel. This code must be re-entrant for two channels. If you want to add channels, you must expand the variable arrays that this works with. Right now, there are two items for each array. Four channels would be very easy to implement.

I broke my own rule on doing ISRs in assembler. I would have done this in assembler if I had had the time. Assembler would be more efficient. Just the same, I haven't had any problems with it, even at higher baud rates. See Listing 24.13.

Listing 24.13. RS-232 device driver code continued.

```
/*********************************************************
  This does the grunt work for each of the two ISRs
  This MUST remain reentrant for two ISRs.
*********************************************************/

static void handleISR(U32 i)        /* i is the device */
{
U8 *pRBuf, *pXBuf;

pRBuf = pRecvBuf[i];
pXBuf = pSendBuf[i];

while (TRUE)
{
    int_id[i] = InByte (IIR[i]);        /* Get ID Byte from IIR */
    switch (int_id[i])
    {
        case RCVSTAT:
            stat_byte[i] = InByte(LSR[i]);  /* clear error conditions */
            break;

        case RVCDATA:
            if (cRecvBuf[i] == sRecvBuf[i])
            {       /* Overflow!! */
                recv_error[i] = ErcRcvBufOvr;       /* Do not put in buf */
                InByte (THR[i]);                    /* trash the byte */
            }
            else
            {
#asm
    CLI
#endasm
```

```
                pRBuf[head_recv[i]] =
                    InByte (THR[i]);    /* Get the byte */
                ++cRecvBuf[i];
                if (++head_recv[i] == SRECVBUF)
                    head_recv[i] = 0;
#asm
    STI
#endasm
                recv_error[i] = 0;
            }
            break;

        case TXEMPTY:
#asm
    CLI
#endasm
            if (cSendBuf[i])
            {
                OutByte(pXBuf[tail_send[i]], THR[i]);  /* Send the byte */
                if (++tail_send[i] == sSendBuf[i])
                    tail_send[i] = 0;
                --cSendBuf[i];
                fExpectInt[i] = TRUE;
            }
            else
                fExpectInt[i] = FALSE;
#asm
    STI
#endasm
        break;

        case MDMSTAT:
        mstat_byte[i] = InByte (MSR[i]);   /* Get Modem Status */
        break;

        case NOINT:
        stat_byte[i] = InByte(LSR[i]);  /* clear error conditions */
        default:
        return;
    }
}
}


static void interrupt comISR0(void)
{
;       /* ; needed if asm is first in function */
#asm
    STI
#endasm
    handleISR(0);
    EndOfIRQ(4);
    return;
}


static void interrupt comISR1(void)
{
```

continues

Listing 24.13. continued.

```
;       /* ; needed if asm is first in function */
#asm
    STI
#endasm
    handleISR(1);
    EndOfIRQ(3);
    return;
}

/*****************************************/


static long ReadByteC(U32 device, unsigned char *pByteRet)
{
U32 counter;
U8 *pRBuf;

    pRBuf = pRecvBuf[device];
    if (recv_error[device]) return (recv_error[device]);

    if (cRecvBuf[device])
    {
        *pByteRet = pRBuf[tail_recv[device]];
        if (++tail_recv[device] == sRecvBuf[device])
            tail_recv[device] = 0;
        --cRecvBuf[device];
        return (0);
    }

    counter = comstat[device].RTimeOut;     /* set up for timeout */
    while (counter--)
    {
        Sleep(1);
        if (cRecvBuf[device])
        {
            *pByteRet = pRBuf[tail_recv[device]];
            if (++tail_recv[device] == sRecvBuf[device])
                tail_recv[device] = 0;
            --cRecvBuf[device];
            return (0);
        }
    }
    return (ErcRecvTimeout);
}

/*****************************************/


static long ReadRecordC(U32 device,
                unsigned char *pDataRet,
                unsigned int  sDataMax,
                unsigned int *pcbRet)
{
int  erc, cb;

    erc = 0;
```

```
    cb = 0;
    while ((cb < sDataMax) && (!erc))
    {
        erc = ReadByteC(device, pDataRet++);
        if (!erc) ++cb;
    }
    *pcbRet = cb;       /* tell em how many bytes */
    return (erc);
}


/*****************************************/
static long WriteByteC(U32 device, unsigned char b)
{

U32 erc, counter;
U8 *pXBuf;

    erc = 0;
    pXBuf = pSendBuf[device];
    counter = comstat[device].XTimeOut;     /* set up for timeout */

    while (cSendBuf[device] == sSendBuf[device])
    {
        Sleep(1);
        counter--;
        if  (!counter)
            return (ErcXmitTimeout);        /* never got sent */
    }

#asm
    CLI
#endasm

    if (!fExpectInt[device])
    {           /* Xmit buf empty, send ourself */
        OutByte(b, THR[device]);
        fExpectInt[device] = TRUE;
        }
    else
    {
        pXBuf[head_send[device]] = b;
        if (++head_send[device] == sSendBuf[device])
            head_send[device]  = 0;
        ++cSendBuf[device];             /* one more in buf */
    }
#asm
    STI
#endasm
    return (erc);
}


/*****************************************/
static long WriteRecordC(U32 device,
                unsigned char *pSendData,
```

continues

Listing 24.13. continued

```
                unsigned int cbSendData)
{
int erc;

    erc = 0;
    while ((cbSendData) && (!erc))
    {
        erc = WriteByteC(device, *pSendData++);
        --cbSendData;
    }
    return (erc);
}


/*****************************************/

static long DiscardRecvC(U32 device)
{
U32 saveto, erc;
U8  b;

    saveto = comstat[device].RTimeOut;
    comstat[device].RTimeOut = 1;
    erc = 0;
    while (!erc)
        erc = ReadByteC(device, &b);
    comstat[device].RTimeOut = saveto;
    return (0);
}



/*****************************************
 This sets comms params prior to opening, or
 while a channel is in use.
*****************************************/

static U32 SetParams(U32 device)

{
U32  divisor, speed;
U8   c, parity, bits, stop_bit, temp;

    parity = comstat[device].parity;
    bits = comstat[device].databits;
    stop_bit = comstat[device].stopbits;
    speed = comstat[device].Baudrate;

    /* Set up baud rate */

    divisor = 115200/speed;

#asm
    CLI
#endasm
    c=InByte (LCR[device]);
```

```
    OutByte  ((c ¦ 0x80), LCR[device]);
    OutByte  ((divisor & 0x00ff), DLAB_LO[device]);
    OutByte  (((divisor>>8) & 0x00ff), DLAB_HI[device]);
    OutByte  (c, LCR[device]);
#asm
    STI
#endasm

    /* set comms params */

    temp = bits - 5;
    temp ¦= ((stop_bit == 1) ? 0x00 : 0x04);

    switch (parity)
    {
       case NO_PAR : temp ¦= 0x00; break;
       case OD_PAR : temp ¦= 0x08; break;
       case EV_PAR : temp ¦= 0x18; break;
    }

#asm
    CLI
#endasm
    OutByte (temp, LCR[device]);
#asm
    STI
#endasm

    return (0);
}


/*****************************************
 This allocates buffers, sets up the ISR
 and IRQ values and open the channel for use.
*****************************************/

static U32  OpenCommC(U32 device)

{
U32  erc;
U16  port_base;
U8   c;

    if (comstat[device].commJob)
        return(ErcChannelOpen);

    GetJobNum(&comstat[device].commJob);

    erc = AllocOSPage(comstat[device].XBufSize/4096,
                      &pSendBuf[device]);

    if (!erc)
    {
        erc = AllocOSPage(comstat[device].RBufSize/4096,
                          &pRecvBuf[device]);

        if (erc)  /* get rid of Xmit buf if we can't recv */
```

continues

Listing 24.13. continued

```
            DeAllocPage(pSendBuf[device],
                        comstat[device].XBufSize/4096);
    }

    if (erc)
    {
        comstat[device].commJob = 0;
        return (erc);
    }

    port_base = comstat[device].IOBase;

    /* Set up buffer variables for this port */

    cSendBuf[device] = 0;
    head_send[device] = 0;
    tail_send[device] = 0;

    cRecvBuf[device] = 0;
    head_recv[device] = 0;
    tail_recv[device] = 0;
    recv_error[device] = 0;


    THR[device]     = port_base;
    IER[device]     = port_base + 1;
    IIR[device]     = port_base + 2;
    FCR[device]     = port_base + 2;
    LCR[device]     = port_base + 3;
    MCR[device]     = port_base + 4;
    LSR[device]     = port_base + 5;
    MSR[device]     = port_base + 6;
    DLAB_HI[device] = port_base + 1;
    DLAB_LO[device] = port_base;

    InByte(THR[device]);    /* reset any pending ints on chip */
    InByte(LSR[device]);

#asm
    CLI
#endasm
    control_byte[device] = RTS ¦ DTR ¦ OUT2;
    OutByte(control_byte[device], MCR[device]); /* Mod Ctrl Reg   */
    OutByte(0x0F, IER[device]);                 /* Int Enable Reg */

    /* See if we have a 16550 and set it up if we do!! */

    OutByte(0x03, FCR[device]);
    c = InByte(IIR[device]);
    if (c & 0xC0)       /* we have a 16550 and it's set to go! */
        f16550[device] = 1;
    else
        f16550[device] = 0;     /* 8250 or 16450 */

#asm
    STI
```

```
#endasm

    SetParams(device);

    UnMaskIRQ(comstat[device].IRQNum);
    return (0);
}

/*****************************************
 This closes the port, sets the owner to 0
 and deallocates the buffers.
*****************************************/

static int  CloseCommC (U32 device)
{
U32 erc;

    MaskIRQ(comstat[device].IRQNum);
    OutByte(0, MCR[device]);
    OutByte(0, IER[device]);
    erc = DeAllocPage(pSendBuf[device],
                  comstat[device].XBufSize/4096);
    erc = DeAllocPage(pRecvBuf[device],
                  comstat[device].RBufSize/4096);
    comstat[device].commJob = 0;
    return (erc);
}
```

Just as with the IDE hard disk driver (and all other drivers in MMURTL), three functions provide the interface from the outside. They are the last functions defined in this file. Their offsets (entry points) have been defined in the device control block for each of the two devices. This was done before calling `InitDevDr()`.

This driver is re-entrant, and therefore all data must be associated with the particular comms port you are addressing. See Listing 24.14.

Listing 24.14. RS-232 device driver code continued.

```
/*****************************************
Called for all device operations.  This
assigns physical device from logical number
that outside callers use. For RS-232, 5=0
and 6=1.
*****************************************/

static U32 comdev_op(U32 dDevice,
           U32 dOpNum,
           U32 dLBA,
           U32 dnBlocks,
           U8  *pData)
{
U32 erc;
U32 Job, device;
```

continues

Listing 24.14. continued

```
U8 c;

/* Set internal drive number */
/* 5    RS-232 1    COM1    (OS built-in) */
/* 6    RS-232 2    COM2    (OS built-in) */

    if (dDevice == 5)
        device = 0;
    else
        device = 1;

    GetJobNum(&Job);

    if ((!comstat[device].commJob) && (dOpNum != CmdOpenC))
        return(ErcNotOpen);

    if (comstat[device].commJob)
    {
        if ((comstat[device].commJob != Job) &&
            (Job != 1))
            return(ErcNotOwner);
    }

    erc = 0;        /* default error */

    switch(dOpNum)
    {

        case(0):
            break;              /* Null Command */
        case CmdReadB:
            erc = ReadByteC(device, pData);
            break;
        case CmdWriteB:
            erc = WriteByteC(device, *pData);
            break;
        case CmdReadRec:
            erc = ReadRecordC(device, pData, dnBlocks,
                        &comstat[device].LastTotal);
            break;
        case CmdWriteRec:
            erc = WriteRecordC(device, pData, dnBlocks);
            break;
        case CmdSetRTO:
            comstat[device].RTimeOut = dLBA;        /* 10ms intervals */
            break;
        case CmdSetXTO:
            comstat[device].XTimeOut = dLBA;        /* 10ms intervals */
            break;
        case CmdOpenC:
            erc =  OpenCommC(device);
            break;
        case CmdCloseC:
            erc =  CloseCommC(device);
            break;
        case CmdDiscardRcv:
```

```
            erc = DiscardRecvC(device);
            break;
        case CmdSetDTR:
            control_byte[device] |= DTR;
            OutByte(control_byte[device], LCR[device]);
            break;
        case CmdSetRTS:
            control_byte[device] |= RTS;
            OutByte(control_byte[device], MCR[device]);
            break;
        case CmdReSetDTR:
            control_byte[device] &= ~DTR;
            OutByte(control_byte[device], LCR[device]);
        case CmdReSetRTS:
            control_byte[device] &= ~RTS;
            OutByte(control_byte[device], MCR[device]);
            break;
        case CmdBreak:
            c = InByte(LCR[device]);
            OutByte((c | 0x40), LCR[device]);
            Sleep(dLBA);
            OutByte(c, LCR[device]);
            break;
        case CmdGetDC:
            *pData = mstat_byte[device] & CD;
            break;
        case CmdGetDSR:
            *pData = mstat_byte[device] & DSR;
            break;
        case CmdGetCTS:
            *pData = mstat_byte[device] & CTS;
            break;
        case CmdGetRI:
            *pData = mstat_byte[device] & RI;
            break;
        default:
            break;
    }

    comstat[device].LastErc = erc;
    return(erc);
}


/*****************************************
Called for status report on comms channel.
Returns 64 byte block for channel specified.
This is called by the PUBLIC call DeviceStat
*****************************************/

static U32 comdev_stat(U32  dDevice,
             S8  *pStatRet,
             U32 dStatusMax,
             U32 *pdStatusRet)
{
U32 i, device;

    /* Set internal device number */
```

continues

Listing 24.14. continued

```
    if (dDevice == 5)
        device = 0;
    else device = 1;

    if (dStatusMax > 64)
        i = 64;
    else
        i = dStatusMax;

    if (!device)
    {
        CopyData(&comstat[0], pStatRet, i);     /* copy the status data */
    }
    else
    {
        CopyData(&comstat[1], pStatRet, i);     /* copy the status data */
    }

    *pdStatusRet = dStatusMax;      /* give em the size returned */

    return(0);
}

/*****************************************
Called to set parameters for the comms
channels prior to opening or while in use.
If an invalid value is passed in, all params
remain the same as before.
Some comms channel params may not be changed
while the channel is in use.
This is called by the PUBLIC call DeviceInit.
*****************************************/

static S32 comdev_init(U32  dDevice,
            S8  *pInitData,
            U32  sdInitData)

{
U32  erc, Xbufsize, Rbufsize, device;
U32  speed, XTO, RTO;
U16  port_base;
U8   parity, bits, stop_bit, IRQNUM;

    if (dDevice == 5)
        device = 0;             /* Set internal device number */
    else
        device = 1;

    if (sdInitData < 40)
        return(ErcBadInitSize);

    pCS = pInitData;

    /* Get the callers new params */

    speed     = pCS->Baudrate;  /* Non Volatile */
```

```
parity    = pCS->parity;    /* Non Volatile */
bits      = pCS->databits;  /* Non Volatile */
stop_bit  = pCS->stopbits;  /* Non Volatile */
XTO       = pCS->XTimeOut;  /* Non Volatile */
RTO       = pCS->RTimeOut;  /* Non Volatile */

port_base = pCS->IOBase;
Xbufsize  = pCS->XBufSize;
Rbufsize  = pCS->RBufSize;
IRQNUM    = pCS->IRQNum;

/* Non Volatile params can be set whether or not the
   channel is open. Do these first and return errors. */

if ((speed > MAX_BAUD) || (speed < MIN_BAUD))
   return (ErcBadBaud);

if ((parity < NO_PAR) || (parity > OD_PAR))
    return (ErcBadParity);

if ((bits < 5) || (bits > 8))
    return (ErcBadDataBits);

if ((stop_bit < 1) || (stop_bit > 2))
    return (ErcBadStopBits);

if (!XTO)  XTO = 1;
if (!RTO)  RTO = 1;

comstat[device].Baudrate = speed;
comstat[device].parity   = parity;
comstat[device].databits = bits;
comstat[device].stopbits = stop_bit;
comstat[device].XTimeOut = XTO;
comstat[device].RTimeOut = RTO;

/* If we got here, the params are OK.  Now we check
   to see if the channel is open and call SetParams
   if so.  The channel is open if the JobNumber
   in the commstat record is NON-ZERO.
*/

if (comstat[device].commJob)
{  /* Channel Open! */
    SetParams(device);
}

/* Channel is not open so we check and set rest of params */

else
{

    if (!port_base)
        return (ErcBadIOBase);
    if (IRQNUM < 3)
        return (ErcBadCommIRQ);

    /* We now round up buffer sizes to whole pages */
```

continues

Listing 24.14. continued

```
        Xbufsize = Xbufsize/4096 * 4096;
        if (Xbufsize % 4096) Xbufsize+=4096; /* another page */


        Rbufsize = Rbufsize/4096 * 4096;
        if (Rbufsize % 4096) Rbufsize+=4096; /* another page */


        comstat[device].IOBase = port_base;
        comstat[device].IRQNum = IRQNUM;
        comstat[device].XBufSize = Xbufsize;
        comstat[device].RBufSize = Rbufsize;

        /* Local copies so we don't work from a structure in ISR */

        sSendBuf[device] = Xbufsize;    /* Size of buffer (allocated) */
        sRecvBuf[device] = Rbufsize;    /* Size of buffer (allocated) */

        erc = 0;
    }

 return(erc);
}
```

25

Keyboard Code

The keyboard code is probably one of the most complicated source files, second only to the file system. The keyboard code is complicated because it has a little bit of everything in it. It contains an interrupt service routine, multilevel lookup tables, a complete system service with an additional task, and several hardware handling calls.

Even though the keyboard is technically a device, it does not use the standard MMURTL device-driver interface. There are several reasons for this. The first is that the keyboard was one of the very first devices coded on the system. The second reason is that the keyboard is only an input device and is always shared among applications. Another reason is that the keyboard I/O ports, which actually go to an 8042 (or equivalent) microprocessor, are used for a wide variety of things. The keyboard is a device found on almost all system boards on PC-AT ISA-compatible platforms.

Keyboard Data

The data portion of the file is much larger than for most of the files in MMURTL. It has several look-up tables used to translate scan codes from what was supposed to be a standardized system.

The systems are not as standardized as I once thought. Some of them support all of the modes for the IBM PC-AT, and some of them support only two out of three. I stuck with the most common scan code set, which was the original scan set 2 for the PC-AT. I have the original PC-AT documentation and PS/2 documentation. Some of the ISA machines support PS/2 scan sets, but only the original scan set 2 was supported on all of the systems I tested. Hence, I use it to support the widest array of machines possible.

There are three tables that provide the translation. The first provides the initial translation from the raw scan codes; the second is for the scan codes that are preceded by the E0 hex escape sequence; and the third is for shifted keystrokes. See Listing 25.1.

Listing 25.1. Keyboard data and tables.

```
.DATA                       ;Begin Keyboard Data
.INCLUDE MOSEDF.INC
.INCLUDE RQB.INC
.INCLUDE TSS.INC
.ALIGN DWORD

EXTRN ddVidOwner    DD

;The keyboard service is designed to return the complete status
;of the keyboard shift, control, alt, and lock keys as well as
;the key in the buffer that the state applies to.
;
;This is done by using two buffers. The first is a "RAW" key buffer
;that holds keys from the interrupt service routine which contains
;all raw keystrokes including shift keys going up and down.
;This information is needed to build coherent keystroke information
```

```
; for an application.  This buffer is 32 bytes long.
;
;The second buffer is 256 bytes and contains the translated information
;in the form of DWords (a 4 byte quantity times 64).  This DWord contains the
;following information:
;
;Low byte  (bits 0-7)   Partially translated application keystroke
;Next byte (bits 8-15)  Shift State (Ctrl, Shift, Alt)
;Next byte (bits 16-23) Lock State (CAPS, Num, Scroll)
;Hi Byte   (bits 24-31) Key Source (Bit 1 Set = Key From Numeric Pad)
;
KBDSvcName  DB 'KEYBOARD'

KbdMainExch DD 0            ;Used by the Kbd Process
KbdHoldExch DD 0            ;Used for requestors that don't own the keyboard
KbdGlobExch DD 0            ;Used to hold Global CTRL-ALT requests
KbdWaitExch DD 0            ;Used for Kbd Wait-For-Key requests
KbdTempExch DD 0            ;Used to rifle thru requests

dGlobalKey  DD 0            ;Global Key Found, else 0
dfPrefix    DD 0
KbdMsgBuf1L DD 0            ;Message buffers for Kbd Service
KbdMsgBuf1H DD 0

KbdMsgBuf2L DD 0
KbdMsgBuf2H DD 0

KbdOwner        DD 1        ;Current owner of Kbd (Mon is default)
KbdAbortJob     DD 0        ;Job that is aborting
KbdCancelJob    DD 0        ;Job that is cancelling global request

rgbKbdBuf   DB 20h DUP (0) ;32 byte RAW buffer
dKbdCnt     DD 0
pKbdIn      DD OFFSET rgbKbdBuf     ;ptr to next char going in
pKbdOut     DD OFFSET rgbKbdBuf     ;ptr to next char going out
;
bKbdMode    DB 0        ;ZERO for Cooked, 1 for RAW
fKBDInitDone DB 0           ;Set true when KBD Service is up and running
;
KbdState    DB 0        ;(See State Masks below)
KbdLock     DB 0        ;(See Lock Masks below)
;
rgdKBBuf    DD 40h DUP (0)  ;64 Dwords for translated key buffer
dKBCnt      DD 0
pKBIn       DD OFFSET rgdKBBuf  ;ptr to codes comming in
pKBOut      DD OFFSET rgdKBBuf  ;ptr to codes going out
;
; These "masks" are for keyboard states that change with special keys:
; They are BIT OFFSETS and NOT MASKS for logical operations!!!!!

CtrlLeftBit    EQU 0
CtrlRiteBit    EQU 1
ShftLeftBit    EQU 2
ShftRiteBit    EQU 3
AltLeftBit     EQU 4
AltRiteBit     EQU 5
;
```

continues

Listing 25.1. continued

```
; Mask to tell if one of the 3 states exist (Ctrl, Shift, Alt)
CtrlDownMask    EQU 00000011b
ShftDownMask    EQU 00001100b
AltDownMask     EQU 00110000b
;
; BIT OFFSETS
CpLockBit       EQU 2
NmLockBit       EQU 1
ScLockBit       EQU 0
; MASKS
CpLockMask      DB 00000100b
NmLockMask      DB 00000010b
ScLockMask      DB 00000001b

;
; The following special keys are processed by the Keyboard Task and handled
; as follows:
; NUMLOCK  - Lights NumLock LED and processes keys accordingly
; SHIFT    - Sets shift flag and processes keys accordingly
; CTRL     - Sets Ctrl flag
; CAPSLOCK - Lights CapsLock LED and processes keys accordingly
; ALT      - Sets Alt flag.
; SCRLLOCK - Lights ScrollLock LED and flag
;
; This table is used to translate all active editing keys from
; the raw value provided by the hardware.
;
;                                   SHIFT
;                                   Value
KbdTable DB 0;          00
DB  01Bh      ;  Esc    01
DB  031h      ;  1      02      21h  !
DB  032h      ;  2      03      40h  @
DB  033h      ;  3      04      23h  #
DB  034h      ;  4      05      24h  $
DB  035h      ;  5      06      25h  %
DB  036h      ;  6      07      5Eh  ^
DB  037h      ;  7      08      26h  &
DB  038h      ;  8      09      2Ah  *
DB  039h      ;  9      0A      28h  (
DB  030h      ;  0      0B      29h  )
DB  02Dh      ;  -      0C      5Fh  _
DB  03Dh      ;  =      0D      2Bh  +
DB  008h      ;  BkSpc  0E
DB  009h      ;  TAB    0F
DB  071h      ;  q      10      51h
DB  077h      ;  w      11      57h
DB  065h      ;  e      12      45h
DB  072h      ;  r      13      52h
DB  074h      ;  t      14      54h
DB  079h      ;  y      15      59h
DB  075h      ;  u      16      55h
DB  069h      ;  i      17      49h
DB  06Fh      ;  o      18      4Fh
DB  070h      ;  p      19      50h
DB  05Bh      ;  [      1A      7Bh
```

```
DB  05Dh    ;  ]       1B     7Dh
DB  00Dh    ;  CR      1C
DB  0h      ;  LCtrl   1D    Special handling
DB  061h    ;  a       1E     41h
DB  073h    ;  s       1F     53h
DB  064h    ;  d       20     44h
DB  066h    ;  f       21     46h
DB  067h    ;  g       22     47h
DB  068h    ;  h       23     48h
DB  06Ah    ;  j       24     4Ah
DB  06Bh    ;  k       25     4Bh
DB  06Ch    ;  l (L)   26     4Ch
DB  03Bh    ;  ;       27     3Ah
DB  027h    ;  '         28         22h
DB  060h    ;  `       29     7Eh
DB  0h      ;  LfShf   2A    Special handling
DB  05Ch    ;  \       2B     7Ch
DB  07Ah    ;  z       2C     5Ah
DB  078h    ;  x       2D     58h
DB  063h    ;  c       2E     43h
DB  076h    ;  v       2F     56h
DB  062h    ;  b       30     42h
DB  06Eh    ;  n       31     4Eh
DB  06Dh    ;  m       32     4Dh
DB  02Ch    ;  ,       33     3Ch
DB  02Eh    ;  .       34     3Eh
DB  02Fh    ;  /       35     3Fh
DB  0h      ;  RtShf   36    Special handling
DB  02Ah    ;  Num *   37     Num pad
DB  0h      ;  LAlt    38    Special handling
DB  020h    ;  Space   39
DB  0h      ;  CpsLk   3A    Special handling
DB  00Fh    ;  F1      3B
DB  010h    ;  F2      3C
DB  011h    ;  F3      3D
DB  012h    ;  F4      3E
DB  013h    ;  F5      3F
DB  014h    ;  F6      40
DB  015h    ;  F7      41
DB  016h    ;  F8      42
DB  017h    ;  F9      43
DB  018h    ;  F10     44
DB  0h      ;  NumLk   45   Special handling
DB  0h      ;  ScrLk   46   Special handling
DB  086h    ;  Num 7   47     37h     Num Home
DB  081h    ;  Num 8   48     38h     Num Up
DB  085h    ;  Num 9   49     39h     Num Pg Up
DB  0ADh    ;  Num -   4A             Num Pad
DB  083h    ;  Num 4   4B     34h     Num Left
DB  09Fh    ;  Num 5   4C     35h     Num (Extra code)
DB  084h    ;  Num 6   4D     36h     Num Right
DB  0ABh    ;  Num +   4E             Num Pad
DB  08Bh    ;  Num 1   4F     31h     Num End
DB  082h    ;  Num 2   50     32h     Num Down
DB  08Ch    ;  Num 3   51     33h     Num Pg Dn
DB  08Eh    ;  Num 0   52     30h     Num Insert
DB  0FFh    ;  Num .   53     2Eh     Num Del
```

continues

Listing 25.1. continued

```
DB  01Ch    ;  Pr Scr  54              SYS REQUEST
DB  000h    ;          55
DB  000h    ;          56
DB  019h    ;  F11     57
DB  01Ah    ;  F12     58
DB  000h    ;          59
DB  000h    ;          5A
DB  000h    ;          5B
DB  000h    ;          5C
DB  000h    ;          5D
DB  000h    ;          5E
DB  000h    ;          5F   ;The following chars are subs from table2
DB  00Eh    ;  Ins      60  Cursor pad
DB  00Bh    ;  End      61  Cursor pad
DB  002h    ;  Down     62  Cursor pad
DB  00Ch    ;  PgDn     63  Cursor pad
DB  003h    ;  Left     64  Cursor pad
DB  000h    ;          65
DB  004h    ;  Right    66  Cursor pad
DB  006h    ;  Home     67  Cursor pad
DB  001h    ;  Up       68  Cursor pad
DB  005h    ;  PgUp     69  Cursor pad
DB  07Fh    ;  Delete   6A  Cursor pad
DB  0AFh    ;  /        6B  Num Pad
DB  08Dh    ;  ENTER    6C  Num Pad
DB  0h      ;          6D
DB  0h      ;          6E
DB  0h      ;          6F
DB  0h      ;          70
DB  0h      ;          71
DB  0h      ;          72
DB  0h      ;          73
DB  0h      ;          74
DB  0h      ;          75
DB  0h      ;          76
DB  0h      ;          77
DB  0h      ;          78
DB  0h      ;          79
DB  0h      ;          7A
DB  0h      ;          7B
DB  0h      ;          7C
DB  0h      ;          7D
DB  0h      ;          7E
DB  0h      ;          7F
;
;This table does an initial character translation from the characters
;provided by the keyboard.  The Kbd translates incoming keystrokes
;from the original scan set 2 for the IBM PC-AT. All PCs are set to this
;by default.  Keys on the 101 keyboard that were common to the numeric
;keypad use a two character escape sequence begining with E0 hex.
;If we see an E0 hex we scan this table and provide the translation
;to another unique character which is looked up in the primary
;table above.  This gives us unique single characters for every key!
;
```

```
nKbdTable2 EQU 16
;
;
KbdTable2 DB 052h,  060h    ;Insert
          DB 04Fh,  061h    ;End
          DB 050h,  062h    ;Down
          DB 051h,  063h    ;Pg Down
          DB 04Bh,  064h    ;Left
          DB 04Dh,  066h    ;Rite
          DB 047h,  067h    ;Home
          DB 048h,  068h    ;Up
          DB 049h,  069h    ;Pg Up
          DB 053h,  06Ah    ;Delete
          DB 037h,  06Bh    ;Num /
          DB 01Ch,  06Ch    ;Num ENTER
          DB 038h,  070h    ;Right ALT DOWN     These are special cause we
          DB 01Dh,  071h    ;Right Ctrl DOWN    track UP & DOWN!!!
          DB 0B8h,  0F0h    ;Right ALT UP
          DB 09Dh,  0F1h    ;Right Ctrl UP

;This table provides shift level values for codes from the primary KbdTable.
;In Shift-ON state ALL keycodes are translated through this table.
;In CAPS LOCK state codes 61h to 7Ah are translated through this table
;In NUM LOCK state only codes with High Bit set are translated
;

KbdTableS DB 0; 00
DB  38h     ;   01  Up   8  Numeric pad
DB  32h     ;   02  Dn   2  Numeric pad
DB  34h     ;   03  Left 4  Numeric pad
DB  36h     ;   04  Rite 6  Numeric pad
DB  39h     ;   05  PgUp 9  Numeric pad
DB  37h     ;   06  Home 7  Numeric pad
DB  07h     ;   07
DB  08h     ;   08
DB  09h     ;   09
DB  0Ah     ;   0A
DB  31h     ;   0B  End  1  Numeric Pad
DB  33h     ;   0C  PgDn 3  Numeric pad
DB  0Dh     ;   0D
DB  30h     ;   0E  Ins  0  Numeric pad
DB  0Fh     ;   0F
DB  10h     ;   10
DB  11h     ;   11
DB  12h     ;   12
DB  13h     ;   13
DB  14h     ;   14
DB  15h     ;   15
DB  16h     ;   16
DB  17h     ;   17
DB  18h     ;   18
DB  18h     ;   19
DB  1Ah     ;   1A
DB  1Bh     ;   1B
DB  1Ch     ;   1C
```

continues

Listing 25.1. continued

```
DB  1Dh     ;   1D
DB  1Eh     ;   1E
DB  35h     ;   1F  Blnk 5  Numeric pad
DB  20h     ;   20
DB  21h     ;   21
DB  22h     ;   22
DB  23h     ;   23
DB  24h     ;   24
DB  25h     ;   25
DB  26h     ;   26
DB  22h     ;   27  '   "
DB  28h     ;   28
DB  29h     ;   29
DB  2Ah     ;   2A
DB  2Bh     ;   2B
DB  3Ch     ;   2C  ,   <
DB  5Fh     ;   2D  -   _
DB  3Eh     ;   2E  .   >
DB  3Fh     ;   2F  /   ?
DB  29h     ;   30  0   )
DB  21h     ;   31  1   !
DB  40h     ;   32  2   @
DB  23h     ;   33  3   #
DB  24h     ;   34  4   $
DB  25h     ;   35  5   %
DB  5Eh     ;   36  6   ^
DB  26h     ;   37  7   &
DB  2Ah     ;   38  8   *
DB  28h     ;   39  9   (
DB  3Ah     ;   3A
DB  3Ah     ;   3B  ;   :
DB  3Ch     ;   3C
DB  2Bh     ;   3D  =   +
DB  3Eh     ;   3E
DB  3Fh     ;   3F
DB  40h     ;   40
DB  41h     ;   41
DB  42h     ;   42
DB  43h     ;   43
DB  44h     ;   44
DB  45h     ;   45
DB  46h     ;   46
DB  47h     ;   47
DB  48h     ;   48
DB  49h     ;   49
DB  4Ah     ;   4A
DB  4Bh     ;   4B
DB  4Ch     ;   4C
DB  4Dh     ;   4D
DB  4Eh     ;   4E
DB  4Fh     ;   4F
DB  50h     ;   50
DB  51h     ;   51
DB  52h     ;   52
DB  53h     ;   53
```

```
DB  54h     ;   54
DB  55h     ;   55
DB  56h     ;   56
DB  57h     ;   57
DB  58h     ;   58
DB  59h     ;   59
DB  5Ah     ;   5A
DB  7Bh     ;   5B  [  {
DB  7Ch     ;   5C  \  |
DB  7Dh     ;   5D  ]  }
DB  5Eh     ;   5E
DB  5Fh     ;   5F
DB  7Eh     ;   60  '  ~
DB  41h     ;   61  a  A
DB  42h     ;   62  b  B
DB  43h     ;   63  c  C
DB  44h     ;   64  d  D
DB  45h     ;   65  e  E
DB  46h     ;   66  f  F
DB  47h     ;   67  g  G
DB  48h     ;   68  h  H
DB  49h     ;   69  i  I
DB  4Ah     ;   6A  j  J
DB  4Bh     ;   6B  k  K
DB  4Ch     ;   6C  l  L
DB  4Dh     ;   6D  m  M
DB  4Eh     ;   6E  n  N
DB  4Fh     ;   6F  o  O
DB  50h     ;   70  p  P
DB  51h     ;   71  q  Q
DB  52h     ;   72  r  R
DB  53h     ;   73  s  S
DB  54h     ;   74  t  T
DB  55h     ;   75  u  U
DB  56h     ;   76  v  V
DB  57h     ;   77  w  W
DB  58h     ;   78  x  X
DB  59h     ;   79  y  Y
DB  5Ah     ;   7A  z  Z
DB  7Bh     ;   7B
DB  7Ch     ;   7C
DB  7Dh     ;   7D
DB  7Eh     ;   7E
DB  2Eh     ;   7F  Del  .  Numeric Pad


KbdScvStack     DD 127 DUP(0)   ;512 byte stack for KbsSvc
KbdSvcStackTop  DD 0

;=============================================================================
;The following equates are for the hardware handling code.
;8042 Status Byte, Port Hex 0064 Read
;=============================================================================

STATUSPORT      EQU 64h
COMMANDPORT     EQU 64h
DATAPORT        EQU 60h
```

continues

Listing 25.1. continued

```
PARITYERROR      EQU 10000000b
GENERALTIMEOUT   EQU 01000000b
AUXOUTBUFFFULL   EQU 00100000b
INHIBITSWITCH    EQU 00010000b
COMMANDDATA      EQU 00001000b
SYSTEMFLAG       EQU 00000100b
INPUTBUFFFULL    EQU 00000010b
OUTPUTBUFFFULL   EQU 00000001b
```

Keyboard Code

The keyboard code is divided into several sections. The first section is the ISR; the second is a small routine that reads scan codes from the raw ISR buffers; the third is the shift translation code, followed by the keyboard service, and finally all of the support routines.

Keyboard ISR

The keyboard ISR places all raw scan codes into a 32-byte buffer. When a code is placed in the buffer, a message is sent to the keyboard service to tell it that the raw buffer has something in it. This uses `ISendMsg()`, which is used by ISRs to send messages when interrupts are disabled. See Listing 25.2.

Listing 25.2. Keyboard ISR code.

```
.CODE
;
;
;ISR for the keyboard.  This is vectored to by the processor whenever
;INT 21 fires off.  This puts the single byte from the 8042
;KBD processor into the buffer.  Short and sweet the way all ISRs
;should be... (most are not this easy though).  This also sends
;a message to the KBD Task (using ISend) when the buffer is almost
;full so it will be forced to process some of the raw keys even
;if no keyboard requests are waiting.

PUBLIC IntKeyBrd:                    ;Key Board (KB) INT 21
        PUSHAD                       ;Save all registers
        MOV ESI, pKbdIn              ;Set up pointer
        XOR EAX,EAX
        IN  AL, 60h                  ;Read byte
        MOV EBX, dKbdCnt             ;See if buffer full
        CMP EBX, 20h                 ;Buffer size
        JE KbdEnd                    ;Buffer is full - Don't save it
        MOV BYTE PTR [ESI], AL           ;Move into buf
        INC dKbdCnt                      ;One more in the buf
        INC ESI                          ;Next byte in
        CMP ESI, OFFSET rgbKbdBuf+20h    ;past end yet?
```

```
        JB KbdEnd
        MOV ESI, OFFSET rgbKbdBuf   ;Back to beginning of buffer

KbdEnd:
        MOV pKbdIn, ESI             ;Set up pointer for next time

        CMP BYTE PTR fKBDInitDone, 0  ;Service isn't ready for messages yet
        JE KbdExit

        ;ISend a msg to Keyboard task
        MOV EBX, KbdMainExch        ;Yes - ISend Message to KbdTask
        PUSH EBX                    ;exchange to send to
        PUSH 0FFFFFFFFh             ;bogus msg
        PUSH 0FFFFFFFFh             ;bogus msg
        CALL FWORD PTR _ISendMsg        ;tell him to come and get it...

KbdExit:
        PUSH 1
        CALL FWORD PTR _EndOfIRQ
        POPAD
        IRETD

;This gets one byte from the raw Kbd Buffer and returns it in AL
;Zero is returned if no key exists.
;
ReadKBDBuf:
        CLI
        MOV ESI, pKbdOut        ;Get ptr to next char to come out
        MOV EAX, dKbdCnt        ;See if there are any bytes
        CMP EAX, 0
        JE RdKBDone             ;No - Leave 0 in EAX
        DEC dKbdCnt             ;Yes - make cnt right
        XOR EAX,EAX
        MOV AL, BYTE PTR [ESI]          ;Put byte in AL
        INC ESI
        CMP ESI, OFFSET rgbKbdBuf+20h   ;past end yet?
        JB RdKBDone
        MOV ESI, OFFSET rgbKbdBuf   ;Back to beginning of buffer

RdKBDone:
        MOV pKbdOut, ESI            ;Save ptr to next char to come out
        STI
        RETN
```

Keyboard Translation

When the keyboard is notified that raw scan codes have been placed in the keyboard buffer, this routine is called to translate them and place them in the final 64-key buffer. Some interrupts may not necessarily result in a key code being sent to the final buffer. For instance, if you press the Shift key and then let it go without pressing another key, the result will be no additional key codes in the final buffer. See Listing 25.3.

Listing 25.3. Keyboard translation code.

```
; Reads and processes all bytes from the RAW keyboard buffer
; and places them and their proper state bytes and into the next DWord
; in the translated buffer if it is an edit key.

XLateRawKBD:
        CALL ReadKbdBuf
        CMP EAX, 0
        JE XLateDone            ;No

        MOV BL, bKbdMode        ;See if we are RAW... (for testing ONLY)
        CMP BL, 1
        JNE KB001               ;NO - keep going
        JMP KB029A              ;Yes, leave the key in AL for buffer

        ;Now we check to see if the byte is 0Eh which tells us
        ;this is a two key code that needs to be translated from
        ;our special table before processing.  This turns the
        ;two key code into a single code and sets a state
        ;bit to indicate this.

KB001:
        CMP dfPrefix, 1
        JE KB003
        CMP AL, 0E0h            ;Key PREFIX???
        JNE KB006               ;No
        MOV dfPrefix, 1         ;Yes, We got an E0.
        JMP XLateDone           ;

KB003:
        MOV dfPrefix, 0         ;No prefix
        MOV ESI, OFFSET KbdTable2
        MOV ECX, nKbdTable2

KB004:
        CMP BYTE PTR [ESI], AL
        JE  KB005
        INC ESI                 ;Two bytes further into table 2
        INC ESI
        DEC ECX
        JNZ KB004               ;Go to next table entry
        JMP XLateDone           ;No translation - ignore it

KB005:
        INC ESI                 ;One byte further over to get Xlate byte
        XOR EAX,EAX
        MOV AL, [ESI]           ;Fall thru to check on char...

      ;This next section checks for special keys (shift, alt, etc.)
      ;BL has SHIFT state, CL has LOCK State, AL has byte from buffer.

KB006:
        MOV dfPrefix, 0         ;No prefix
        XOR EBX, EBX
        XOR ECX, ECX
```

```
        MOV BL, KbdState        ;BL has Shift, Alt, Ctrl states
        MOV CL, KbdLock         ;BH has Num, Caps, & Scroll Lock

        CMP AL, 45h             ;Key = NumLock ?
        JNE KB007               ;NO...
        BTC ECX, NmLockBit      ;Compliment bit
        JMP KB022

KB007:
        CMP AL, 3Ah             ;Caps Lock?
        JNE KB008
        BTC ECX, CpLockBit      ;Compliment bit in BH
        JMP KB022

KB008:
        CMP AL, 46h             ;Scroll Lock?
        JNE KB009
        BTC ECX, ScLockBit      ;Compliment bit in BH
        JMP KB022

KB009:
        CMP AL, 2Ah             ;Char Left Shift On?
        JNE KB010
        BTS EBX, ShftLeftBit
        JMP KB021

KB010:
        CMP AL, 36h             ;Right Shift On?
        JNE KB011
        BTS EBX, ShftRiteBit
        JMP KB021

KB011:
        CMP AL, 0AAh            ;Left Shift Off?
        JNE KB012
        BTR EBX, ShftLeftBit
        JMP KB021

KB012:
        CMP AL, 0B6h            ;Right Shift Off?
        JNE KB013
        BTR EBX, ShftRiteBit
        JMP KB021

KB013:
        CMP AL, 1Dh             ;Left Ctrl On?
        JNE KB014
        BTS EBX, CtrlLeftBit
        JMP KB021

KB014:
        CMP AL, 71h             ;Right Ctrl On?
        JNE KB015
        BTS EBX, CtrlRiteBit
        JMP KB021
```

continues

Listing 25.3. continued

```
KB015:
        CMP AL, 09Dh            ;Left Ctrl Off?
        JNE KB016
        BTR EBX, CtrlLeftBit
        JMP KB021

KB016:
        CMP AL, 0F1h            ;Right Ctrl Off?
        JNE KB017
        BTR EBX, CtrlRiteBit
        JMP KB021

KB017:
        CMP AL, 38h             ;Left Alt On?
        JNE KB018
        BTS EBX, AltLeftBit
        JMP KB021

KB018:
        CMP AL, 70h             ;Right Alt On?
        JNE KB019
        BTS EBX, AltRiteBit
        JMP KB021

KB019:
        CMP AL, 0B8h            ;Left Alt Off?
        JNE KB020
        BTR EBX, AltLeftBit
        JMP KB021

KB020:
        CMP AL, 0F0h            ;Right Alt Off?
        JNE KB023
        BTR EBX, AltRiteBit

KB021:
        MOV KbdState, BL        ;Put Kbd Shift State back
        JMP XLateDone           ;

KB022:
        MOV KbdLock, CL         ;Put Kbd Lock State back
        CALL SetKbdLEDS         ;Set LEDs on keyboard
        JMP XLateRawKBD         ;

        ;We jumped here if it wasn't a key that is specially handled

KB023:
        TEST AL,  80h           ;Check for high bit (key-up code)
        JNZ  XLateDone          ;Go back, else fall through

        OR AL,AL                ;Zero not a valid code
        JZ XLateDone            ;Go back, else fall through
```

```
        ;If we got here, IT'S AN EDIT KEY DOWN!
        ;Now we lookup the code and do a single translation.

        AND EAX, 07Fh               ;Chop off any upper bit junk
        MOV ESI, OFFSET KbdTable    ;Set up to index table
        MOV DL, BYTE PTR [ESI+EAX]  ;Save in DL
        OR AL,AL                    ;Zero not a valid code
        JZ XLateDone                ;Go back, else fall through

        MOV CL, KbdState            ;Get Shift state
        MOV CH, KbdLock             ;Get lock state

        ;TO let the user know if the key came from the Numeric
        ;keypad we set the high bits in the first translation
        ;table for these keys.  This next piece of code tests for it
        ;and sets the low bit in DH if it its. DH is later moved
        ;into the high byte of the returned key code.

        MOV DH, 0
        TEST DL, 80h                ;High bit set?
        JZ SHORT KB024
        MOV DH, 1                   ;Indicates key came numeric pad

        ;See if shift key is down and shift all keys if it is
KB024:
        TEST CL, ShftDownMask       ;Either shift key down?
        JZ KB025                    ;No, go look for locks
        CMP DL, 21h                 ;Is key < ' '
        JB KB025                    ;Yes, look for locks
        CMP DL, 7Eh                 ;Is key > '~'
        JA KB025                    ;Yes, look for locks
        JMP SHORT KB027             ;In-range, go do the translation

KB025:  ;See if key is from Numeric Keypad (high bit will be set)
        TEST DL, 80h                ;High bit set?
        JZ KB026                    ;No
        AND CH, NmLockMask          ;Yes, is NumLock ON
        JZ KB026
        CMP DL, 0ADh                ;Do not shift DASH (-) Special Case
        JE KB026
        JMP SHORT KB027             ;Do the shift Xlation

KB026:  ;See if Caps Lock is on and if key is between 61h and 7Ah
        ;do the translation

        TEST CH, CpLockMask         ;Is CpLock ON
        JZ KB029                    ;No
        CMP DL, 61h                 ;Is key >= 'a'
        JB KB029                    ;No
        CMP DL, 7Ah                 ;Is key <= 'z'
        JA KB029                    ;No
        ;Fall through to do the translation
```

continues

Listing 25.3. continued

```
KB027:  ;Do the shift translation and leave in DL
        MOV AL, DL                  ;Put in AL
        AND EAX, 07Fh               ;Chop all above 7 bits
        MOV ESI, OFFSET KbdTableS   ;Set up to index table
        MOV DL, BYTE PTR [ESI+EAX]  ;Save in DL
        ;Fall though to put key in final buffer

       ;Place DL in the LOW byte of the DWord to go into the
       ;final buffer (the data the user will get)
       ;If the high bit is set coming from the primary
       ;translation table, this means the key was from the
       ;numeric keypad so we set the numpad bit in status

KB029:
        MOV AH, DH
        SHL EAX, 8              ;Num Pad indicator
        MOV AL, KbdState        ;Get Shift state
        MOV AH, KbdLock         ;Get lock state
        SHL EAX, 8
        AND DL, 7Fh             ;Lop of high bit (if there)
        MOV AL, DL

        ;EAX now has the buffered info for the user (Key, Shifts & Locks)
        ;Now we put it in the DWord buffer if it is NOT a GLOBAL.
        ;If global, we put it in dGlobalKey.

        TEST AH, CtrlDownMask       ;Either Ctrl Down?
        JZ KB029A                   ;No
        TEST AH, AltDownMask        ;Either Alt Down?
        JZ KB029A                   ;No

        ;It IS a global key request!
        MOV dGlobalKey, EAX         ;Save it
        JMP  XLateRawKBD            ;Back for more (if there is any)

KB029A:
        MOV  EBX, dKBCnt            ;See if buffer full
        CMP  EBX, 64                ;number of DWords in final buffer
        JE   XLateDone              ;Buffer is FULL..
        MOV  ESI, pKBIn             ;Get ptr to next IN to final buffer
        MOV  [ESI], EAX             ;Move into buf
        INC  dKBCnt                 ;One more DWord in the buf
        ADD  ESI, 4
        CMP  ESI, OFFSET rgdKBBuf+100h
        JB   KB030
        MOV  ESI, OFFSET rgdKBBuf   ;Reset to buf beginning

KB030:
        MOV  pKBIn, ESI         ;Save ptr to next in
        JMP  XLateRawKBD

XlateDone:
        XOR EAX, EAX
        RETN
```

Reading the Final Buffer

This routine is called to take a key out of the final buffer if one is available. This buffer `rgdKBBuf` holds the 32-bit fully encoded key value. See Listing 25.4.

Listing 25.4. Read keyboard buffer code.

```
; Returns a keyboard code from FINAL keyboard buffer.
; Returns zero in EAX if buffer is empty.
;
; IN :  Nothing
; OUT:  EAX has Key or 0 if none
; USED: EAX, ESI
; MODIFIES: dKBCnt, pKBOut
;
ReadKBFinal:
        MOV EAX, dKBCnt
        CMP EAX, 0
        JE KBFDone                  ;Nothing final buffer
        DEC dKBCnt                  ;One more DWord in the buf
        MOV ESI, pKBOut             ;ptr to next code out
        MOV EAX, [ESI]              ;Put it in EAX
        ADD ESI, 4                  ;Next code please...
        CMP ESI, OFFSET rgdKBBuf+100h   ;Past end of buff?
        JB  KBF02                   ;No
        MOV ESI, OFFSET rgdKBBuf    ;Yes, Reset to beginning

KBF02:
        MOV pKBOut, ESI             ;Update pKBOut

KBFDone:
        RETN
```

The Keyboard Service

`KBDServiceTask` is a complete system service. It services five codes, including `Abort (0)`. It handles requests from multiple clients; this means it must hold the requests for those jobs that do not own the keyboard.

When the keyboard is reassigned to a new job, the requests it was holding must be reevaluated to see whether the new job had an outstanding keyboard request.

This service also accepts messages from the keyboard ISR to indicate that translation of raw keyboard scan codes must be done. See Listing 25.5.

Listing 25.5. Keyboard system service code.

```
;=====================================================

; This is the keyboard Service task. It is an infinite loop
; that services requests from users of the keyboard.  It
; calls XLateRawKBD to process raw keyboard buffer data,
; and waits at the KeyBoard Service Main Exchange for users.
; When it gets a request it checks the service code and handles
; it accordingly.

KBDServiceTask:
        CALL XLateRawKBD            ;Processes RAW buffer if not empty

        CMP DWORD PTR dGlobalKey, 0
        JE  KST01                   ;No global key came in

KBDGlobal1:
        PUSH KbdGlobExch
        PUSH OFFSET KbdMsgBuf1L     ;Where to return pRqBlk
        CALL FWORD PTR _CheckMsg        ;Check to see if RqWaiting
        OR EAX, EAX                 ;Yes if ZERO
        JZ KBDGlobal2               ;Rq waiting for global
        MOV DWORD PTR dGlobalKey, 0 ;Wipe out global key (no one wants it)
        JMP KBDServiceTask          ;Start over again

KBDGlobal2:
        MOV EBX, KbdMsgBuf1L        ;pRqBlk into EBX
        MOV ESI, [EBX+pData1]       ;Ptr where to return key
        OR ESI, ESI                 ;Is it null?? (Bad news if so)
        JNZ KBDGlobal3              ;No, probably good ptr
        PUSH EBX                    ;Yes, BAD PTR. Push pRqBlk
        PUSH ErcNullPtr             ;Push error
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

KBDGlobal3:
        MOV EDX, dGlobalKey
        MOV [ESI], EDX              ;Give em the key!
        PUSH EBX                    ;Push pRqBlk
        PUSH 0                      ;Push NO ERROR
        CALL FWORD PTR _Respond
        JMP KBDGlobal1              ;Go back to see if others want it

KST01:
        CMP DWORD PTR ddVidOwner, 2 ;Debugger has video
        JNE KST01ND                 ;NOT in Debugger
        PUSH 20
        CALL FWORD PTR _Sleep           ;
        CALL XLateRawKBD            ;Processes RAW buffer
        JMP KBDServiceTask          ;Go back to the top

KST01ND:
        PUSH KbdMainExch            ;See if someones "Requesting"
        PUSH OFFSET KbdMsgBuf1L     ;
        CALL FWORD PTR _WaitMsg     ;Wait for the message
```

```
        ;If we got here we got a Request or Msg from ISR
        CMP DWORD PTR KbdMsgBuf1L, 0FFFFFFFFh   ;Is it a msg from the KBD ISR?
        JNE KST02                   ;No, jump to handle Request

        ;If we got here, ISR sent msg to us (something in the buffer)

        CALL XLateRawKBD            ;Processes RAW buffer

        CMP DWORD PTR dGlobalKey, 0
        JNE KBDGlobal1              ;A global key came in

        PUSH KbdWaitExch            ;See if owner is waiting for a key
        PUSH OFFSET KbdMsgBuf1L     ;Where to return Request or msg
        CALL FWORD PTR _CheckMsg        ;Check to see if another msg came in
        OR EAX, EAX                 ;Yes if ZERO
        JNZ KBDServiceTask          ;No Rq/Msg waiting, Go back to the top
                                    ;Fall thru to check request
KST02:
        ;If we got here we've got a Request from Main or Wait Exch
        MOV EBX, KbdMsgBuf1L        ;pRqBlk into EBX
        MOV CX, [EBX+ServiceCode]   ;Save in CX

        CMP CX, 0                   ;Job Abort Notify
        JE KSTAbort                 ;
        CMP CX, 1                   ;ReadKbd
        JE KSTRead                  ;
        CMP CX, 2                   ;ReadKbdGlobal
        JE KSTReadGlobal            ;
        CMP CX, 3                   ;CancelGlobal
        JE KSTCancelGlobal          ;
        CMP CX, 4                   ;AssignKBD
        JE KSTAssignKbd             ;
        PUSH EBX                    ;HOMEY DON'T SERVICE THAT!
        PUSH ErcBadSvcCode          ;Bad service code
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

;-----------------------------------------------------
KSTRead:
        MOV EAX, [EBX+RqOwnerJob]   ;Whose Request is it?
        CMP EAX, KbdOwner
        JE KSTRead00                ;This guy owns it!
        PUSH EBX                    ;Not the owner, so send to Hold Exch
        PUSH KbdHoldExch            ;
        CALL FWORD PTR _MoveRequest
        JMP KBDServiceTask          ;Go back to the top

KSTRead00:
        CALL ReadKBFinal            ;Get Code from Buf (Uses EAX, ESI)
        CMP EAX, 0                  ;No Key in Final Buffer
        JE KSTRead02                ;Go see if they asked to wait
        MOV ESI, [EBX+pData1]       ;Ptr where to return key
        CMP ESI, 0                  ;Is it null?? (Bad news if so)
        JNE KSTRead01               ;No, probably good ptr
        PUSH EBX                    ;Yes, BAD PTR. Push pRqBlk
        PUSH ErcNullPtr             ;Push error
```

continues

Listing 25.5. continued

```
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

KSTRead01:
        MOV [ESI], EAX              ;Give them the key code
        PUSH EBX                    ;RqHandle
        PUSH 0                      ;NO Error
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

KSTRead02:
        CMP DWORD PTR [EBX+dData0], 0   ;Wait for key?  0 in dData0 = Don't wait
        JNE KSTRead04               ;Yes
        PUSH EBX
        PUSH ErcNoKeyAvail          ;Error Code (No key to give you)
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

KSTRead04:
        PUSH EBX                    ;They opted to wait for a key
        PUSH KbdWaitExch            ;Send em to the wait exch
        CALL FWORD PTR _MoveRequest
        JMP KBDServiceTask          ;Go back to the top
;-------------------------------------------------------

KSTAbort:
        ;Respond to all requests we are holding for Job in dData0
        ;with Erc with ErcOwnerAbort. Then respond to Abort
        ;request last. Requests can be at the HoldExch, the WaitExch,
        ;or the GlobalKeyExch. We must chack all 3!
        ;Save abort job for comparison

        MOV EAX, [EBX+dData0]       ;Get aborting job number
        MOV KbdAbortJob, EAX        ;this is aborting job

KSTAbort10:                         ;Check the WaitExch
        PUSH KbdWaitExch            ;See if he was "waiting" for a key
        PUSH OFFSET KbdMsgBuf2L     ;Where to return Request
        CALL FWORD PTR _CheckMsg    ;
        OR EAX, EAX                 ;Yes (someones waiting) if ZERO
        JNZ KSTAbort20              ;No more waiters
        MOV EDX, KbdMsgBuf2L        ;pRq of holding job into EDX
        MOV EBX, [EDX+RqOwnerJob]
        CMP EBX, KbdAbortJob
        JE KSTAbort11               ;Go to respond with Erc
        PUSH EDX                    ;Else move Request to MainKbd Exch
        PUSH KbdMainExch            ; to be reevaluated
        CALL FWORD PTR _MoveRequest
        JMP KSTAbort10              ;Go back to look for more waiters

KSTAbort11:                         ;
        PUSH EDX                    ;Respond to this request
        PUSH ErcOwnerAbort          ;cause he's dead
        CALL FWORD PTR _Respond
        JMP SHORT KSTAbort10
```

```
KSTAbort20:                     ;Check HoldExch for dead job
        PUSH KbdHoldExch        ;See if anyone is on hold
        PUSH OFFSET KbdMsgBuf2L ;Where to return Request
        CALL FWORD PTR _CheckMsg ;
        OR EAX, EAX             ;Yes (someone's holding) if ZERO
        JNZ KSTAbort30          ;No more holders
        MOV EDX, KbdMsgBuf2L    ;pRq of holding job into EDX
        MOV EBX, [EDX+RqOwnerJob]
        CMP EBX, KbdAbortJob
        JE KSTAbort21           ;Go to respond with Erc
        PUSH EDX                ;Else move Request to MainKbd Exch
        PUSH KbdMainExch        ; to be reevaluated. It's not him.
        CALL FWORD PTR _MoveRequest
        JMP KSTAbort20          ;Go back to look for more holders

KSTAbort21:                     ;
        PUSH EDX                ;Respond to this request
        PUSH ErcOwnerAbort      ;cause he's dead
        CALL FWORD PTR _Respond
        JMP SHORT KSTAbort20    ;Go back to look for more holders

KSTAbort30:                     ;Check GlobalExch for dead job
        PUSH KbdGlobExch        ;See if anyone is at global
        PUSH OFFSET KbdMsgBuf2L ;Where to return Request
        CALL FWORD PTR _CheckMsg ;
        OR EAX, EAX             ;Yes (someones holding) if ZERO
        JNZ KSTAbort40          ;No more holders
        MOV EDX, KbdMsgBuf2L    ;pRq of holding job into EDX
        MOV EBX, [EDX+RqOwnerJob]
        CMP EBX, KbdAbortJob
        JE KSTAbort31           ;Go to respond with Erc
        PUSH EDX                ;Else move Request to MainKbd Exch
        PUSH KbdMainExch        ; to be reevaluated
        CALL FWORD PTR _MoveRequest
        JMP KSTAbort30          ;Go back to look for more globals

KSTAbort31:                     ;
        PUSH EDX                ;Respond to this request
        PUSH ErcOwnerAbort      ;cause he's dead
        CALL FWORD PTR _Respond
        JMP SHORT KSTAbort30

KSTAbort40:                     ;Respond to original abort Req
        MOV EBX, KbdMsgBuf1L    ;pRqBlk of original Abort Request
        PUSH EBX                ;
        PUSH 0                  ;Error Code (OK)
        CALL FWORD PTR _Respond
        JMP KBDServiceTask      ;Go back to the top
;-------------------------------------------------------

KSTReadGlobal:
        PUSH EBX                ;They want a global key
        PUSH KbdGlobExch        ;Send em to the Global exch
        CALL FWORD PTR _MoveRequest
        JMP KBDServiceTask      ;Go back to the top
;-------------------------------------------------------
```

continues

Listing 25.5. continued

```
;Assign a new owner for the keyboard. We must check to see
;if the new owner had any keyboard requests on hold.
;We do this by sending all the requests that were on hold
;back to the main exchange to be reevaluated.
;Then we respond to the original request.

KSTAssignKBD:
        ;Change owner of Kbd
        MOV EAX, [EBX+dData0]       ;Get new owner
        CMP EAX, KbdOwner
        JNE KSTAssign01             ;New Owner!
        PUSH EBX                    ;Same owner
        PUSH 0                      ;Error Code (OK)
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top

KSTAssign01:
        MOV KbdOwner, EAX           ;Set new owner

KSTAssign02:                        ;Move all waiters to main exch
        PUSH KbdWaitExch            ;See if anyone is "waiting" for a key
        PUSH OFFSET KbdMsgBuf2L     ;Where to return Request
        CALL FWORD PTR _CheckMsg    ;
        OR EAX, EAX                 ;Yes (someones waiting) if ZERO
        JNZ KSTAssign03             ;No more waiters
        MOV EDX, KbdMsgBuf2L        ;pRq into EDX
        PUSH EDX                    ;Move Request to MainKbd Exch
        PUSH KbdMainExch            ; to be reevaluated
        CALL FWORD PTR _MoveRequest
        JMP KSTAssign02             ;Go back to look for more waiters

KSTAssign03:                        ;Waiters have been moved, Respond to Req
        PUSH KbdHoldExch            ;See if anyone is on hold
        PUSH OFFSET KbdMsgBuf2L     ;Where to return Request
        CALL FWORD PTR _CheckMsg       ;
        OR EAX, EAX                 ;Yes if ZERO
        JNZ KSTAssign04             ;No more holders
        MOV EDX, KbdMsgBuf2L        ;pRq into EDX
        PUSH EDX                    ;Move Request to MainKbd Exch
        PUSH KbdMainExch            ; to be reevaluated
        CALL FWORD PTR _MoveRequest
        JMP KSTAssign03             ;Go back to look for more holders

KSTAssign04:                        ;Holders have been moved, Respond to Req
        MOV EBX, KbdMsgBuf1L        ;pRqBlk of original Assign Request
        PUSH EBX                    ;
        PUSH 0                      ;Error Code (OK)
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top
;-----------------------------------------------

KSTCancelGlobal:
        ;Rifle thru Global Exch and respond with ErcNoKeyAvail
        ;to those with the same JobNum as dData0
        MOV EAX, [EBX+dData0]   ;Save Job that is cancelling global request
        MOV KbdCancelJob, EAX   ;
```

```
KSTCancel10:                        ;Check GlobalExch for canceled job
        PUSH KbdGlobExch            ;See if anyone is at global
        PUSH OFFSET KbdMsgBuf2L     ;Where to return Request
        CALL FWORD PTR _CheckMsg    ;
        OR EAX, EAX                 ;Yes (someones holding) if ZERO
        JNZ KSTCancel20             ;No more globals
        MOV EDX, KbdMsgBuf2L        ;pRq of holding job into EDX
        MOV EBX, [EDX+RqOwnerJob]
        CMP EBX, KbdCancelJob
        JE KSTCancel11              ;Go to respond with Erc
        PUSH EDX                    ;Else move Request to MainKbd Exch
        PUSH KbdMainExch            ; to be reevaluated
        CALL FWORD PTR _MoveRequest
        JMP KSTCancel10             ;Go back to look for more globals

KSTCancel11:                            ;
        PUSH EDX                    ;Respond to this request
        PUSH ErcNoKeyAvail          ;cause he cancelled it!
        CALL FWORD PTR _Respond
        JMP SHORT KSTCancel10       ;Back to check for more

KSTCancel20:                        ;Respond to original cancel Req
        MOV EBX, KbdMsgBuf1L        ;pRqBlk of original Request
        PUSH EBX                    ;
        PUSH 0                      ;Error Code (OK)
        CALL FWORD PTR _Respond
        JMP KBDServiceTask          ;Go back to the top
;=============================================================
```

Keyboard Procedural Interface

When programs don't need the power of the request interface, such as multiple asynchronous requests, services can provide a blocking procedural call to make the application's job easier.

This code actually makes the request for the caller. The caller doesn't have to know anything about the request interface to use this. See Listing 25.6.

Listing 25.6. Code for blocking read keyboard call.

```
;PUBLIC blocking call to read the keyboard. This uses the
;Default TSS exchange and the stack to make the request to
;the keyboard service for the caller.  The request is a standard
;service code one (Wait On Key) request.
;If fWait is NON-ZERO, this will not return without a key unless
;a kernel/fatal error occurs.
;
;The call is fully reentrant (it has to be...).
;
; Procedural interface:
;
;   ReadKbd(pKeyCodeRet, fWait): dError
```

continues

Listing 25.6. continued

```
;
;   pKeyCodeRet is a pointer to a DWORD where the keycode is returned.
;     [EBP+16]
;   fWait is NON-ZERO to wait for a key.
;     [EBP+12]
;
; Stack Variables:
;   Hndl    [EBP-4]
;
PUBLIC __ReadKBD:
        PUSH EBP                        ; Save the Previous FramePtr
        MOV EBP,ESP                     ; Set up New FramePtr
        SUB ESP, 4                      ; One DWORD local var

        MOV EAX, OFFSET KBDSvcName  ;'KEYBOARD '
        PUSH EAX

        PUSH 1                          ;Service Code (Read Keyboard)

        MOV ECX,pRunTSS                 ;Get TSS_Exch for our use
        MOV EBX,[ECX+TSS_Exch]          ;Exchange (TSS Exch)
        PUSH EBX                        ;

        LEA EAX, [EBP-4]                ;Rq Handle (Local Var)
        PUSH EAX

        PUSH 0                          ;npSend
        MOV EAX, [EBP+16]               ;Key Code return (Their Ptr)
        PUSH EAX                        ;pData1
        PUSH 4                          ;Size of key code
        PUSH 0                          ;pData2
        PUSH 0                          ;cbData2

        XOR EAX, EAX
        CMP DWORD PTR [EBP+12], 0   ;Don't wait for Key?
        JE ReadKbd1                     ;No wait
        MOV EAX, 1                      ;Set up to wait!

ReadKbd1:
        PUSH EAX                        ;Wait value (dData0)
        PUSH 0
        PUSH 0
        CALL FWORD PTR _Request     ;make the Request

        ;The request is made. Now we call Wait!

        MOV ECX,pRunTSS                 ;Get TSS_Exch for our use
        MOV EBX,[ECX+TSS_Exch]          ;
        PUSH EBX                        ;Pass exchange (for WaitMsg)
        ADD ECX,TSS_Msg                 ;Offset of TSS msg area
        PUSH ECX
        CALL FWORD PTR _WaitMsg     ;Wait on it

        ;When we get here the caller should have the key code
        ;HOWEVER, we want to pass any errors back via EAX

        OR EAX, EAX                     ;Was there a kernel error?
```

```
        JNZ ReadKbdEnd              ;YES.... bummer
        MOV ECX,pRunTSS             ;Get TSS_Msg area so we can get error
        ADD ECX,TSS_Msg             ;Offset of TSS msg area
        MOV EBX, [ECX]              ;pRqBlk (lets look!!)
        MOV EAX, [ECX+4]            ;Service error in second DWord

ReadKbdEnd:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8                  ; Rtn to Caller & Remove Params from stack
```

Debugger Keyboard

The debugger requires a special function to read the keyboard. This allows the debugger to completely bypass the keyboard system service so it doesn't have to go through the kernel for anything. This reads keystrokes directly from the final coded buffer. See Listing 25.7.

Listing 25.7. Debugger read keyboard code.

```
;=========================================================
;Special Call for Debugger so it doesn't have to pass thru
;the kernel Request mechanism for a keystroke.
;It acts like ReadKbd with fWait set to true.
;It sucks keys directly from the Final Keyboard buffer.
;
; Procedural interface:
;
;   ReadDbgKbd(pKeyCodeRet)
;
;   pKeyCodeRet is a pointer to a DWORD where the keycode is returned.
;     [EBP+8]
;
PUBLIC ReadDbgKBD:
        PUSH EBP                    ; Save the Previous FramePtr
        MOV EBP,ESP                 ; Set up New FramePtr

RDKB0:
        CALL ReadKBFinal            ;Get Code from Buf (Uses EAX, ESI)
        OR EAX, EAX                 ;Got a key?? (non zero)
        JNZ RDKB1                   ;No. Loop back again
        PUSH 2                      ;Sleep for 20 ms
        CALL FWORD PTR _Sleep
        JMP RDKB0                   ;Check again

RDKB1:
        MOV ESI, [EBP+8]            ;Ptr where to return key
        MOV [ESI], EAX
        XOR EAX, EAX
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETN 4                  ; Rtn to Caller & Remove Params from stack

;=================================================
```

Keyboard Hardware Set Up

`InitKBD` is called very early in the operating system initialization code to set up the keyboard hardware.

The 8042 provides hardware interrupts, but you don't have to call `SetIRQVector()` because the keyboard interrupt routines address was known at build time. See Listing 25.8.

Listing 25.8. Code to initialize the keyboard hardware.

```
;This sets the Keyboard Scan Set to #2 with 8042 interpretation ON
;
PUBLIC InitKBD:

        PUSH 1                  ;KBD IRQ
        CALL FWORD PTR _MaskIRQ

        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,0FAh         ;Set ALL keys typematic/make/break
        OUT DataPort,AL     ;Send Command to KBD (not 8042)

        CALL OutBuffFull    ;Eat response
        IN AL, DATAPORT

        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,0F0h         ;Set Scan code set
        OUT DataPort,AL     ;Send Command to KBD (not 8042)

        CALL OutBuffFull    ;Eat response
        IN AL, DATAPORT


        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,02h          ;Scan set 2
        OUT DataPort,AL     ;Send Command

        CALL OutBuffFull    ;Eat response
        IN AL, DATAPORT

        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,060h         ;Set up to write 8042 command byte
        OUT COMMANDPORT,AL  ;Send Command
        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,45h          ;Enable IBM Xlate
        OUT DataPort,AL     ;Send Command

        PUSH 1              ;KBD IRQ
        CALL FWORD PTR _UnMaskIRQ

        CALL SetKbdLEDs
        RETN
;=============================================================================
```

Keyboard Service Initialization

`InitKBDService` is called from the monitor to start the keyboard service. The service is a completely separate task that runs in a loop servicing requests. This will be called only once, and it is called after the keyboard hardware and the keyboard ISR are functional. See Listing 25.9.

Listing 25.9. Keyboard service initialization.

```
PUBLIC _InitKBDService:

        ;All initial requests and messages from the ISR come to
        ;this exchange

        MOV EAX, OFFSET KbdMainExch ;Alloc Main Kbd exch for service
        PUSH EAX
        CALL FWORD PTR _AllocExch
        OR EAX, EAX                 ;Check for error on AllocExch
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        ;Requests for ReadkeyBoard (ScvCode #1) that are from jobs
        ;that do NOT currently own the keyboard get sent here using
        ;MoveRequest.

        MOV EAX, OFFSET KbdHoldExch ;Alloc Hold Kbd exch for Kbd service
        PUSH EAX
        CALL FWORD PTR _AllocExch
        OR EAX, EAX                 ;Check for error on AllocExch
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        ;Requests for ReadkeyGlobal (SvcCode #3) wait here until we
        ;get a global key from the keyboard.
        ;
        MOV EAX, OFFSET KbdGlobExch ;Alloc Global Wait exch for Kbd service
        PUSH EAX
        CALL FWORD PTR _AllocExch
        OR EAX, EAX                 ;Check for error on AllocExch
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        ;Requests for ReadkeyBoard (ScvCode #1) that are from job
        ;that currently owns the keyboard waits here if it wants
        ;to wait for a key.

        MOV EAX, OFFSET KbdWaitExch ;Alloc Hold Kbd exch for Kbd service
        PUSH EAX
        CALL FWORD PTR _AllocExch
        OR EAX, EAX                 ;Check for error on AllocExch
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        ;Used to "rifle" thru RqBlks waiting at an exchange.

        MOV EAX, OFFSET KbdTempExch ;Alloc Tmp exch for Kbd service
        PUSH EAX
        CALL FWORD PTR _AllocExch
        OR EAX, EAX                 ;Check for error on AllocExch
```

continues

Listing 25.9. continued

```
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        ;Spawn the Keyboard Service task

        MOV EAX, OFFSET KBDServiceTask
        PUSH EAX
        PUSH 7                      ;Priority
        PUSH 0                      ;fDebug
        MOV EAX, OFFSET KbdSvcStackTop
        PUSH EAX
        PUSH 1                      ;OS Job task
        CALL FWORD PTR _SpawnTask
        OR EAX, EAX                 ;Check for error on AllocExch
        JNZ InitKBDSvcEnd           ;YUP, we got bad problems

        MOV EAX, OFFSET KBDSvcName
        PUSH EAX
        PUSH KbdMainExch
        CALL FWORD PTR _RegisterSvc

InitKBDSvcEnd:
        MOV BYTE PTR fKBDInitDone, 1    ;We're UP!
        RETN
;
;============================================================================
;This tells the 8042 Controller to Disable the Keyboard device.
;============================================================================

KbdDisable:
        PUSH EAX
        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        MOV AL,0ADh     ;Set Command to "Write the 8042 Command Byte"
        OUT COMMANDPORT,AL  ;Send Command
        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        POP EAX
        RETN
```

Hardware Helpers

The rest of the routines are used for various hardware functions, such as reading or writing the 8042 or keyboard data ports, as well as setting the keyboard LEDs to the proper state. See Listing 25.10.

Listing 25.10. Low-level keyboard control code.

```
;============================================================================
; This tells the 8042 Controller to Enable the Keyboard Device.
;============================================================================

KbdEnable:
        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
```

```
        MOV AL,0AEh         ;Set Command to "Write the 8042 Command Byte"
        OUT COMMANDPORT,AL  ;Send Command
        CALL InBuffEmpty    ;Wait for Input Buffer to Empty
        RETN
;
;=============================================================================
; Waits until the 8042 Input Buffer is EMPTY
;=============================================================================

InBuffEmpty:
        PUSH EAX
        PUSH ECX
        MOV ECX,2FFFFh          ;check 128k times
IBE:
        JMP IBE1
IBE1:
        JMP IBE2
IBE2:
        IN AL,STATUSPORT        ;Read Status Byte into AL
        TEST AL,INPUTBUFFFULL   ;Test The Input Buffer Full Bit
        LOOPNZ IBE
        POP ECX
        POP EAX
        RETN

;=============================================================================
; Waits until the 8042 Output Buffer is FULL so we can read it
;=============================================================================
;
; Before calling this makes sure that the Keyboard interrupts have been
; masked so the keyboard interrupt doesn't eat the byte you're
; looking for!!
;
OutBuffFull:
        PUSH EAX
        PUSH ECX
        MOV ECX,2FFFFh
OBF:
        JMP OBF1:
OBF1:
        JMP OBF2:
OBF2:
        IN AL,STATUSPORT    ;Read Status Byte into AL
        TEST AL,OUTPUTBUFFFULL  ;Test The Output Buffer Full Bit
        LOOPZ OBF
        POP ECX
        POP EAX
        RETN

;=============================================================================
; This sets the indicators on the keyboard based on data in KbdState
;=============================================================================

SetKbdLEDs:
        PUSH EAX
```

continues

Listing 25.10. continued

```
        PUSH 1                  ;KBD IRQ
        CALL FWORD PTR _MaskIRQ

        CALL InBuffEmpty        ;Wait for Input Buffer to Empty
        MOV AL,0EDh             ;Set/Reset Status Indicators
        OUT DATAPORT,AL         ;Send KBD Command

        CALL OutBuffFull        ;Eat response
        IN AL, DATAPORT

        CALL InBuffEmpty        ;Wait for Input Buffer to Empty
        MOV AL,KbdLock          ;Get Current Lock Status Byte
        AND AL,00000111b        ;Mask all but low order 3 bits
        OUT DATAPORT,AL         ;Send KBD Command

        CALL OutBuffFull        ;Eat response
        IN AL, DATAPORT

        PUSH 1                  ;KBD IRQ
        CALL FWORD PTR _UnMaskIRQ
        POP EAX
        RETN
```

26

Video Code

The code presented in this chapter implements a VGA text-based video driver. This driver does not use the standard MMURTL device driver interface. The reason it doesn't use the interface is that it was one of the first things written for the operating system. All of my earliest testing depended on being able to see results from test programs.

Virtual Video Concept

In a multitasking operating system that supports text-based video, more than one application at a time may need to write data to or read data from the video screen. This would not be possible if all the programs running wrote to the real video screen buffer. In VGA Text mode, which is the default setup in BIOS on machines that have VGA monitors, eight hardware buffers are actually available. You could use those and switch between them, but doing so would limit you to eight active programs.

To allow as many programs as possible, you allocate a page of system memory (4096 bytes) for each *job* (program) as a virtual video buffer. This prevents future problems. If I want to add a graphical user interface, text-based programs will still run and can be displayed in a window directly from each job's buffer if needed.

The video status for each job is kept in its *job control block* (JCB). The status includes screen coordinates, video screen size, video mode, a pointer to the job's virtual buffer, and a pointer to the current active buffer.

When a new task is assigned to the video screen, the contents of its virtual buffer are copied to the real screen buffer, and the pointer to the current active buffer is changed to point to the real video screen buffer. The reverse occurs for the old job (the one that was using the real screen). The real buffer is copied into the job's virtual buffers, and the job's active pointer is changed back to its own buffer.

VGA Text Video

Many books have been written about how to control standard VGA video hardware, and the hardware seems fairly standardized as well. I won't give much detail on how it works for two reasons. First, I'm not a video "guru," and second, I let the BIOS code in the machine set up the video hardware for the standard VGA Text mode. You may want to set up the video differently in your system. If so, the shelves of your local bookstore are filled with brain dumps from people who have much more knowledge than I. You'll find implementing this code was really a simplicity issue with me.

I control the registers that deal with cursor positioning and which video buffer is used. I use only one. These registers are defined in the data section of the code listing later in this chapter.

The Video Code

You will notice, no doubt, that all of the video code in this chapter is in assembler (as is most of MMURTL). When I first started testing MMURTL, assembler was all I could work with, and I needed to see things. You can only work "blind" for so long.

A very important point for you to know is that this code is completely *re-entrant*, with the exception of those calls that actually manipulate video hardware. Only one program on the system should act as a video and keyboard manager (such as the MMURTL Monitor).

The first items in the source file are the data declarations and the INCLUDE files, which define certain constant values (all of the MMURTL source files are set up to have these items first). These include constants for the video hardware. The code in Listing 26.1 presents the initial items in the source file.

Listing 26.1. Video constants and data.

```
.DATA
.INCLUDE MOSEDF.INC
.INCLUDE JOB.INC

;Video Equates and Types
;
CRTCPort1    EQU 03D4h    ;Index port for CRTC
CRTCPort2    EQU 03D5h    ;Data port for CRTC
CRTCAddHi    EQU 0Ch      ;Register for lo byte of Video address
CRTCAddLo    EQU 0Dh      ;Register for lo byte of Video address
CRTCCurHi    EQU 0Eh      ;Register for lo byte of Cursor address
CRTCCurLo    EQU 0Fh      ;Register for lo byte of Cursor address
CRTC0C       DB 0         ;CRT Reg 0C HiByte address value
CRTC0D       DB 0         ;CRT Reg 0D LoByte address value

PUBLIC ddVidOwner    DD 1         ;JCB that currently owns video
                                  ;Default to monitor (Job 1)

;End of Data & Equates
;
;=================================================================
; BEGIN INTERNAL CODE FOR VIDEO
;=================================================================
;
.CODE
;
EXTRN GetpJCB NEAR
EXTRN GetpCrntJCB NEAR
EXTRN ReadDbgKbd NEAR
EXTRN GetCrntJobNum NEAR
```

The `InitVideo()` makes video screen 0 (zero) the default screen for standard VGA hardware. The VGA text base address is 0B8000h. This address is a constant in one of the INCLUDE

files called VGATextBase. `InitVideo()` is called only once, when the operating system is initialized. Data ports on standard VGA video hardware are accessed as an array with an index. See Listing 26.2.

Listing 26.2. Code to initialize video.

```
PUBLIC InitVideo:
        MOV AL, CRTCAddHi       ;Index of hi byte
        MOV DX, CRTCPort1       ;Index Port
        OUT DX, AL
        MOV AL, CRTC0C          ;hi byte value to send
        MOV DX, CRTCPort2       ;Data Port
        OUT DX, AL

        MOV AL, CRTCAddLo       ;Index of lo byte
        MOV DX, CRTCPort1       ;Index Port
        OUT DX, AL
        MOV AL, CRTC0D          ;lo byte value to send
        MOV DX, CRTCPort2       ;Data Port
        OUT DX, AL
        RETN

;=====================================================================
; BEGIN PUBLIC CODE FOR VIDEO
;=====================================================================
```

The `SetVidOwner(ddJobNum)` call selects the screen that you see. This call is used by the monitor in conjunction with the `SetKbdOwner()` call to change which application gets the keystrokes and video; the application should be the same for both. The internal debugger is the only code that will use this call and not move the keyboard at the same time. The debugger was given its own keyboard code so I could debug the real keyboard code. Don't even ask how I debugged the debugger keyboard code. It certainly wasn't easy. The parameter (`ddJobNum`) is the new job to get the active screen (the one to be displayed). As always, if the call is successful, `EAX` returns `0`. See Listing 26.3.

Listing 26.3. Code to set the video owner.

```
ddJobVidCV  EQU DWORD PTR [EBP+12]

PUBLIC __SetVidOwner:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV EAX, ddJobVidCV     ;
        CMP EAX, ddVidOwner     ;Already own it?
        JNE ChgVid01            ;No
        XOR EAX, EAX            ;Yes
        JMP ChgVidDone
ChgVid01:
        CALL GetpJCB                        ;Leaves ptr to new vid JCB in EAX
        CMP DWORD PTR [EAX+pVidMem] ,0      ;Got valid video memory???
        JNE ChgVid02                        ;Yes
```

```
        MOV EAX, ErcVidNum      ;NO! Give em an error!
        JMP ChgVidDone
ChgVid02:
        ;Save data on screen to CURRENT job's pVirtVid
        MOV EAX, ddVidOwner
        CALL GetpJCB
        PUSH VGATextBase        ;Source
        MOV EBX, [EAX+pVirtVid] ;Destination
        PUSH EBX
        PUSH 4000                   ;Size of video
        CALL FWORD PTR _CopyData    ;Do it!

        ;Make pVidMem same as pVirtVid for CURRENT OWNER
        MOV EAX, ddVidOwner
        CALL GetpJCB            ;Leaves ptr to new vid JCB in EAX
        MOV EBX, [EAX+pVirtVid]
        MOV [EAX+pVidMem], EBX

        ;Update current video owner to NEW owner

        MOV EAX, ddJobVidCV     ;
        MOV ddVidOwner, EAX

        ;Copy in Data from new pVirtVid

        MOV EAX, ddVidOwner
        CALL GetpJCB
        MOV EBX, [EAX+pVirtVid] ;Source
        PUSH EBX
        PUSH VGATextBase        ;Destination
        PUSH 4000               ;Size of video
        CALL FWORD PTR _CopyData    ;Do it!

        ;Make new pVidMem real video screen for new owner

        MOV EAX, ddVidOwner
        CALL GetpJCB
        MOV EBX, VGATextBase
        MOV [EAX+pVidMem], EBX

        ;Set Cursor position

        MOV EAX, ddVidOwner
        CALL GetpJCB
        MOV ECX, EAX
        MOV EBX, [ECX+CrntX]    ;Get current X for new screen
        MOV EAX, [EBX+CrntY]    ;Current Y
        CALL HardXY             ;Set it up
        XOR EAX, EAX            ;No Error
ChgVidDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 4
```

The `SetNormVid(dCharAttr)` selects the normal background attribute used by `ClrScr()` and `ScrollVid()` on the screen. The parameter `dCharAttr` (listed as `ddNormVid` in the `EQU` statement

in Listing 26.4) is the attribute value used in standard video operations on the current screen. The `ClrScr()`, `EditLine()`, and `ScrollVid()` calls use this value. It is saved in the JCB for each job. The EAX register returns zero for no error, and you'll notice that's all it can return. See Listing 26.4.

Listing 26.4. Code to set normal video attribute.

```
ddNormVid   EQU DWORD PTR [EBP+12]

PUBLIC __SetNormVid:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;pJCB -> EAX
        MOV EBX, ddNormVid      ;
        MOV [EAX+NormAttr], EBX ;
        XOR EAX, EAX
        POP EBP                 ;
        RETF 4
```

The `GetNormVid(pVidRet)` call returns the value to the normal screen attribute. This call gets the value that is set with the `SetNormVid()` call. This call expects a pointer to a byte where this value is returned. If you've noticed that you only return a byte although you passed in a word to `SetNormVid()`, you are correct. There's no mistake; the upper three bytes really aren't used (yet). See Listing 26.5.

Listing 26.5. Code to get the normal video attribute.

```
pdNormVidRet EQU DWORD PTR [EBP+12]

PUBLIC __GetNormVid:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;pJCB -> EAX
        MOV EBX, [EAX+NormAttr] ;
        MOV ESI, pdNormVidRet   ;
        MOV [ESI], BL           ;
        XOR EAX, EAX
        POP EBP                 ;
        RETF 4
```

The `GetVidOwner(pdJobNumRet)` function returns the job number currently assigned the active video screen. See Listing 26.6.

Listing 26.6. Code to get the current video owner.

```
pVidNumRet  EQU DWORD PTR [EBP+12]

PUBLIC __GetVidOwner:
```

```
        PUSH EBP                ;
        MOV EBP,ESP             ;
        MOV ESI, pVidNumRet     ;
        MOV EAX, ddVidOwner     ;
        MOV [ESI], EAX          ;
        XOR EAX, EAX            ; no error obviously
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 4
```

The `ClrScr()` call clears the screen for the executing job. If this is the active screen, the real video buffers get wiped. If not, only the caller's virtual video buffer gets cleared. This call uses the value that you set in `SetNormVid()` for the color attribute and a space for the character. See Listing 26.7.

Listing 26.7. Code to clear the screen.

```
PUBLIC __ClrScr:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX
        MOV EDI,[EBX+pVidMem]   ;EDI points to the caller's video memory
MOV EAX, [EBX+NormAttr] ;Attr
        SHL EAX, 8              ;
        MOV AL, 20h             ;
        MOV DX, AX
        SHL EAX, 16
        MOV AX, DX              ;Fill Char & Attr
        MOV ECX,0400h
        CLD
        REP STOSD
        PUSH 0
        PUSH 0
        CALL FWORD PTR _SetXY   ;Erc in EAX on Return
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF
```

The `TTYOut(pTextOut, ddTextOut, ddAttrib)` function translates the text buffer pointed to with `pTextOut` into a stream of characters placed in the caller's video buffer. The beginning x and y coordinates are those found in the caller's JCB. The number of chars of text in the buffer you are pointing to is `ddTextOut`, and `ddAttrib` is the attribute or color you want for all of the characters.

The following characters in the stream are interpreted thus (listed in hex):

- 0A—Line feed. The cursor (next active character placement) will be on the following line at column 0. If this line is below the bottom of the screen, the entire screen will be scrolled up one line, the bottom line will be blanked, and the cursor will be placed on the last line in the first column.

- 0D—Carriage return. The cursor will be moved to column 0 on the current line.
- 08—Backspace. The cursor will be moved one column to the left. If the cursor is already at column 0, Backspace will have no effect. The Backspace is nondestructive (no character values are changed).

I will eventually add 07h (Bell), 07F (Delete), and 0Ch (Form Feed). If you want to go beyond these, you may want to consider writing an ANSI device driver instead of modifying this code. Listing 26.8.

Listing 26.8. Code for TTY stream output to screen.

```
pTextOut    EQU DWORD PTR [EBP+20]
sTextOut    EQU DWORD PTR [EBP+16]
dAttrText   EQU DWORD PTR [EBP+12]

DataByte    EQU BYTE PTR [ECX]

PUBLIC __TTYOut:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX
        MOV EAX, sTextOut       ;make sure count isn't null
        OR EAX, EAX
        JZ TTYDone

TTY00:
        MOV EAX,[EBX+CrntX]     ; EAX has CrntX (col)
        MOV EDX,[EBX+CrntY]     ; EDX has CrntY (line)
        MOV ECX,pTextOut

        CMP DataByte,0Ah        ; LF?
        JNE TTY02
        INC EDX
        CMP EDX,[EBX+nLines]    ; Equal to or past the bottom?
        JB  TTY06               ; No
        JMP TTYScr              ; Yes, goto scroll
TTY02:
        CMP DataByte,0Dh        ; CR?
        JNE TTY03
        MOV EAX,0
        JMP TTY06
TTY03:
        CMP DataByte,08h        ; BackSpace?
        JNE TTY04
        CMP EAX,0
        JE  TTY04
        DEC EAX
        JMP TTY06
TTY04:
        PUSH EBX                ;Save pointer to VCB
```

```
        PUSH EAX                ;X (Param 1)
        PUSH EDX                ;Y (Param 2)
        PUSH ECX                ;pointer to text char (Param3)
        PUSH 1                  ;Param 4 (nchars)
        MOV ECX,dAttrText       ;
        PUSH ECX                ;Param 5
        CALL FWORD PTR _PutVidChars ;
        POP EBX                 ;restore ptr to VCB
        CMP EAX, 0
        JNE TTYDone
        MOV EAX, [EBX+CrntX]
        MOV EDX, [EBX+CrntY]
        INC EAX                 ;Next column
        CMP EAX,[EBX+nCols]
        JNE TTY06               ;Make cursor follow
        MOV EAX,0
        INC EDX
        CMP EDX,[EBX+nLines]    ; past the bottom?
        JNE TTY06               ; No - goto 06 else fall thru

TTYScr:
        DEC EDX                 ; back up one line
        PUSH EAX                ;Save registers (scroll eats em)
        PUSH EBX
        PUSH ECX
        PUSH EDX

        PUSH 0
        PUSH 0
        PUSH 80
        PUSH 25
        PUSH 1                  ;fUP (non zero)
        CALL FWORD PTR _ScrollVid   ;Ignore error
        POP EDX                 ;restore registers
        POP ECX
        POP EBX
        POP EAX                 ;Fall thru to

TTY06:
        PUSH EBX                ;save ptr to pJCB

        PUSH EAX
        PUSH EDX
        CALL FWORD PTR _SetXY

        POP EBX                 ;Restore ptr to VCB
        CMP EAX, 0
        JNE TTYDone
        DEC sTextOut
        JZ TTYDone
        INC pTextOut
        JMP TTY00               ; Go back for next char
TTYDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 12
```

The `PutVidAttrs(ddCol, ddLine, sChars, dAttr)` call sets screen colors (attributes) for the positions and number of characters specified, without affecting the current TTY coordinates or the character data. It is independent of the current video "stream." `PutVidAttrs()` is a fill function and does not set multiple independent attributes. See Listing 26.9.

Listing 26.9. Code to screen video attributes.

```
oADDX       EQU DWORD PTR [EBP+24] ;Param 1 COLUMN
oADDY       EQU DWORD PTR [EBP+20] ;Param 2 LINE
sADDChars   EQU DWORD PTR [EBP+16] ;Param 3 sChars
sADDColor   EQU DWORD PTR [EBP+12] ;Param 4 Attr

PUBLIC __PutVidAttrs:
        PUSH EBP                ;
        MOV EBP,ESP             ;

        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX

        MOV EDI, [EBX+pVidMem]  ;point to this VCBs video memory
        MOV EBX,oADDx           ;x Position
        SHL EBX,1               ;Times 2
        MOV EAX,oADDy           ;y Position
        MOV ECX,0A0h            ;Times 160 (char/attrs per line)
        MUL ECX                 ;Times nColumns
        ADD EAX,EBX
        CMP EAX,0F9Eh           ;Last legal posn on screen
        JBE PutAttrs00
        MOV EAX, ErcVidParam
        JMP PcADone
PutAttrs00:
        MOV ECX,sADDChars
        OR ECX, ECX
        JZ PcADone
        ADD EDI,EAX
        MOV EAX,sADDColor
        CLD
pcAMore:
        INC EDI                 ;Pass the char value
        STOSB                   ;Move Color in
        LOOP pcAMore
        XOR EAX, EAX            ;No Error!
pcADone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 16
```

The `PutVidChars(ddCol, ddLine, pChars, sChars, ddAttrib)` call places characters on the screen without affecting the current TTY coordinates or the TTY data. It is independent of the current video "stream." The parameters include the position (column and line), a pointer to the characters to be placed, the count of characters, and the attribute.

The starting position in screen memory is `(Line * 80 + (Column*2))`. You alternate between characters and attributes as you move through screen memory because this is how the video hardware interprets its buffers for display (for example, put character, then color, then character, then color, and so on). See Listing 26.10.

Listing 26.10. Code to place video characters anywhere on the screen.

```
oDDX        EQU DWORD PTR [EBP+28] ;Param 1 COLUMN
oDDY        EQU DWORD PTR [EBP+24] ;Param 2 LINE
pDDChars    EQU DWORD PTR [EBP+20] ;Param 3 pChars
sDDChars    EQU DWORD PTR [EBP+16] ;Param 4 sChars
sDDColor    EQU DWORD PTR [EBP+12] ;Param 5 Attr

PUBLIC __PutVidChars:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX

        MOV EDI, [EBX+pVidMem]  ;point to this VCBs video memory
        MOV EBX,oDDx
        SHL EBX,1               ;Times 2
        MOV EAX,oDDy
        MOV ECX,0A0h            ;Times 160
        MUL ECX                 ;Times nColumns
        ADD EAX,EBX
        CMP EAX,0F9Eh           ;Last legal posn on screen
        JBE PutChars00
        MOV EAX, ErcVidParam
        JMP PcDone
PutChars00:
        MOV ECX,sDDChars
        OR ECX, ECX
        JZ PcDone
        MOV ESI,pDDChars
        ADD EDI,EAX
        MOV EAX,sDDColor
        CLD
pcMore:
        MOVSB                   ;Move Char in
        STOSB                   ;Move Color in
        LOOP pcMore
        XOR EAX, EAX            ;No Error!
pcDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 20
```

The `GetVidChar(ddCol, ddLine, pCharRet, pAttrRet)` call returns the current character and attribute from the screen coordinates you specify. Whether or not the caller is the active video screen doesn't matter. This is because the pointer in the JCB is set to either the virtual screen or the real one, and this is the pointer you use to acquire the attribute. See Listing 26.11.

Listing 26.11. Code to get a character and attribute.

```
oGDDX       EQU DWORD PTR [EBP+24] ;Param 1 COLUMN
oGDDY       EQU DWORD PTR [EBP+20] ;Param 2 LINE
pGDDCRet    EQU DWORD PTR [EBP+16] ;Param 3 pCharRet
pGDDARet    EQU DWORD PTR [EBP+12] ;Param 4 pAttrRet

PUBLIC __GetVidChar:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX

        MOV EDI, [EBX+pVidMem]  ;point to this VCBs video memory
        MOV EBX,oGDDx
        SHL EBX,1               ;Times 2
        MOV EAX,oGDDy
        MOV ECX,0A0h            ;Times 160
        MUL ECX                 ;Times nColumns
        ADD EAX,EBX
        CMP EAX,0F9Eh           ;Last legal posn on screen
        JBE GetChar00
        MOV EAX, ErcVidParam
        JMP PcGDone
GetChar00:
        ADD EDI,EAX             ;EDI now points to char
        MOV ESI,pGDDCRet
        MOV AL, [EDI]
        MOV [ESI], AL           ;Give them the char
        INC EDI                 ;Move to Attr
        MOV ESI,pGDDARet
        MOV AL, [EDI]
        MOV [ESI], AL           ;Give them the Attr
        XOR EAX, EAX            ;No Error!
pcGDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 20
```

The `ScrollVid(ddULCol, ddULline, nddCols, nddLines, ddfUp)` function scrolls the described square area on the screen either up or down one line. If `ddfUp` is not zero, the scroll will be *up*. The line left blank is filled with `NormAttr` from JCB. The upper-left corner of the area to scroll is described by `ddULCol` and `ddULLine`. The size of the area is `nddCols` and `nddLines`.

To scroll the entire screen up one line, the parameters are `ScrollVid(0,0,80,25,1)`. In this case, the top line is lost (not really scrolled), and the bottom line are blanked. In fact, if you specified `ScrollVid(0,1,80,24,1)`, you would get the same results. See Listing 26.12.

Listing 26.12. Code to scroll an area of the screen.

```
oULX        EQU DWORD PTR [EBP+28] ;Param 1 COLUMN
oULY        EQU DWORD PTR [EBP+24] ;Param 2 LINE
nddCols     EQU DWORD PTR [EBP+20] ;Param 3 pChars
```

```
nddLines    EQU DWORD PTR [EBP+16] ;Param 4 sChars
ddfUP       EQU DWORD PTR [EBP+12] ;Param 5 Attr

PUBLIC __ScrollVid:
       PUSH EBP                ;
       MOV EBP,ESP             ;
       CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
       MOV EBX, EAX            ;Save pJCB & use in EBX
       MOV EAX, oULX
       CMP EAX, 79
       JA svErcExit
       ADD EAX, nddCols
       CMP EAX, 80
       JA svErcExit
       MOV EAX, oULY
       CMP EAX, 24
       JA svErcExit
       ADD EAX, nddLines
       CMP EAX, 25
       JA svErcExit

       CMP ddfUP, 0        ;Scroll UP?
       JNE svUP0           ;Yes... Scroll UP!

;Scroll DOWN begins

       MOV EAX, oULY       ;First line
       ADD EAX, nddLines   ;Last line
       MOV ECX, 160
       MUL ECX                 ;times nBytes per line
       MOV EDI, [EBX+pVidMem]  ;EDI points to video memory 0,0
       MOV EDX, EBX            ;Save pJCB
       ADD EDI, EAX            ;EDI is ptr to 1st dest line
       ADD EDI, oULX       ;offset into line
       ADD EDI, oULX       ;add again for attributes
       MOV ESI, EDI        ;
       SUB ESI, 160        ;ESI is 1st source line
       MOV EBX, ESI        ;Save in EBX for reload
       MOV EAX, nDDLines   ;How many lines to move
       DEC EAX             ;one less than window height
svDOWN1:
       MOV ECX, nddCols    ;How many WORDS per line to move
       REP MOVSW           ;Move a line (of WORDS!)
       MOV EDI, EBX        ;Reload Dest to next line
       MOV ESI, EDI
       SUB ESI, 160
       MOV EBX, ESI        ;Save again
       DEC EAX
       JNZ svDOWN1
       MOV EAX, [EDX+NormAttr] ;Normal video attributes!!!
       SHL EAX, 8
       MOV AL, 20h             ;Space
       MOV EDI, EBX            ;Put the last line into EDI
       MOV ECX, nddCols
       CLD
       REP STOSW
       XOR EAX, EAX        ;No error
```

continues

Listing 26.12. continued

```
        JMP svDone
                            ;No... scroll down begins
svUP0:
        MOV EAX, oULY       ;First line
        MOV ECX, 160
        MUL ECX                 ;times nBytes per line
        MOV EDI, [EBX+pVidMem]  ;EDI points to video memory 0,0
        MOV EDX, EBX            ;Save pJCB
        ADD EDI, EAX            ;EDI is ptr to 1st dest line
        ADD EDI, oULX       ;offset into line
        ADD EDI, oULX       ;add again for attributes
        MOV ESI, EDI        ;
        ADD ESI, 160        ;ESI is 1st source line
        MOV EBX, ESI        ;Save in EBX for reload
        MOV EAX, nDDLines   ;How many lines to move
        DEC EAX             ;two less than window height
svUP1:
        MOV ECX, nddCols    ;How many WORDS per line to move
        REP MOVSW           ;Move a line (of WORDS!)
        MOV EDI, EBX        ;Reload Dest to next line
        MOV ESI, EDI
        ADD ESI, 160
        MOV EBX, ESI        ;Save again
        DEC EAX
        JNZ svUP1
        MOV EAX, [EDX+NormAttr] ;Normal video attributes!!!
        SHL EAX, 8
        MOV AL, 20h             ;Space
        MOV EDI, EBX        ;Put the last line into EDI
        SUB EDI, 160
        MOV ECX, nddCols
        CLD
        REP STOSW
        XOR EAX, EAX        ;No error
        JMP svDone
svErcExit:                  ;Error exits will jump here
        MOV EAX, ErcVidParam
svDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 20
```

The `HardXY()` call is used to support positioning the cursor in hardware. When a video call that repositions the cursor is made and the caller owns the real video screen, `HardXY()` is called to ensure that the hardware follows the new setting. This supports `SetXY()` and `SetVidOwner()`.

The parameters to this call are via registers; thus no stack parameters are used. `EAX` is set with the new y position and `EBX` is set with the new x position before the call is made. See Listing 26.13.

Listing 26.13. Support code to set hardware cursor position.

```
HardXY:
        MOV ECX,80
        MUL ECX                 ; Line * 80
        ADD EAX,EBX             ; Line plus column
        MOV DX,CRTCPort1        ; Index register
        PUSH EAX
        MOV AL,CRTCCurLo
        OUT DX,AL               ; Index 0Fh for low byte
        POP EAX
        MOV DX,CRTCPort2        ; Data register
        OUT DX,AL               ; Send Low byte out
        SHR EAX,08              ; shift hi byte into AL
        PUSH EAX
        MOV DX,CRTCPort1
        MOV AL,CRTCCurHi
        OUT DX,AL               ; Index for High byte
        POP EAX
        MOV DX,CRTCPort2
        OUT DX,AL               ; Send High byte out
        RETN
```

The `SetXY(ddNewX, ddNewY)` function positions the VGA cursor (Text mode) to the x and y position specified in the parameters `ddNewX` and `ddNewY`. If the caller also happens to own the real video screen buffer (being displayed), then the hardware cursor is also repositioned by a call to the internal support call `HardXY()` from Listing 26.13. See Listing 26.14.

Listing 26.14. Code to set cursor position.

```
NewX        EQU DWORD PTR [EBP+16]
NewY        EQU DWORD PTR [EBP+12]

PUBLIC __SetXY:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX

        MOV ECX,NewX            ; Column
        MOV EDX,NewY            ; Line
        MOV [EBX+CrntX],ECX     ; This saves it in the VCB
        MOV [EBX+CrntY],EDX     ;

        CALL GetCrntJobNum      ;Leaves ptr to current JCB in EAX
        CMP EAX, ddVidOwner
        JNE GotoXYDone          ;If not on Active screen, skip it

        MOV EAX,NewY            ;Setup to call HardXY
        MOV EBX,NewX
        CALL HardXY
```

continues

Listing 26.14. continued

```
GotoXYDone:
        XOR EAX,EAX             ;No Error
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8
```

The `GetXY(pddXRet, pddYRet)` function returns the x and y cursor positions from the caller's job control block. The positions in the JCB are updated with each character placement, so this will be accurate even if the caller is not currently displayed on the real video screen. See Listing 26.15.

Listing 26.15. Code to return current cursor position.

```
pXret       EQU DWORD PTR [EBP+16]
pYret       EQU DWORD PTR [EBP+12]

PUBLIC __GetXY:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, EAX

        MOV EAX,[EBX+CrntX]     ; Column
        MOV ESI,pXret
        MOV [ESI], EAX
        MOV EAX,[EBX+CrntY]     ; Line
        MOV ESI,pYret
        MOV [ESI], EAX
        XOR EAX,EAX
QXYDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 8
```

The `EditLine(pStr, dCrntLen, dMaxLen, pdLenRet, pbExitChar, dEditAttr)` function reads a line of text from the keyboard and puts it into the string pointed to by `pStr`. If `pStr` points to a valid string (`dCrntLen > 0`), the string is displayed. The editing of this string is done at the current x and y positions. You also specify the maximum length the string can be (`dMaxLen`), where you want the exit character returned (`pbExitChar`), and finally the attribute for the text you are editing (`dEditAttr`).

The `EditLine()` function probably shouldn't be included with the video code, but rather should be a library function. I added this call while doing a lot of testing; the call came in so handy as part of the operating system code that I just left it where it was.

Display and keyboard are handled entirely inside of `EditLine()`. The following keys are recognized and handled inside:

- 08—Backspace. A destructive backspace, this moves the cursor to the left, replacing char with 20h.
- 20h–7Eh—ASCII text. Any of these places this character in the current x and y position and advances the position of the cursor.

Any other key causes `Editline()` to exit, which returns the string in its current condition and also returns the key that caused `Editline()` to exit. See Listing 26.16.

Listing 26.16. Code to edit a line on the screen.

```
pEdString       EQU DWORD PTR [EBP+32]
ddSzCrnt        EQU DWORD PTR [EBP+28]
ddSzMax         EQU DWORD PTR [EBP+24]
pddSzRet        EQU DWORD PTR [EBP+20]
pExitKeyRet     EQU DWORD PTR [EBP+16]
dEditAttr       EQU DWORD PTR [EBP+12]

;Local vars EditX and EditY hold position of first char of text
;CrntX is the cursor postion

PosnX           EQU DWORD PTR [EBP-04]
EditX           EQU DWORD PTR [EBP-08]
EditY           EQU DWORD PTR [EBP-12]
KeyCode         EQU DWORD PTR [EBP-16]

PUBLIC __EditLine:
        PUSH EBP                ;
        MOV EBP,ESP             ;
        SUB ESP, 16

        CMP ddSzCrnt, 80        ;Is it currently too long?
        JA BadEdit
        CMP ddSzMax, 80         ;Is Max len too long?
JA BadEdit
        MOV EAX, ddSzCrnt
        CMP EAX, ddSzMax        ;Is Crnt len > Max???
        JA BadEdit

        LEA EAX, EditX          ;Get current cursor posns in local vars
        PUSH EAX
        LEA EAX, EditY
        PUSH EAX
        CALL FWORD PTR _GetXY
        CMP EAX, 0
        JNE EditDone            ;Bad Erc from call

        MOV EAX, EditX
        ADD EAX, ddSzCrnt
        MOV PosnX, EAX          ;make PosnX end of string

        MOV ECX, ddSzMax
        SUB ECX, ddSzCrnt       ;ECX  how many bytes to zero
        JZ  EdLn01              ;None to zero out
```

continues

Listing 26.16. continued

```
        MOV ESI, pEdString   ;Initialize current string
        ADD ESI, ddSzCrnt    ;ESI ptr to 1st empty byte
        MOV AL, 20h          ;fill with spaces
Edln00:
        MOV [ESI], AL
        INC ESI
        LOOP Edln00

EdLn01:
        PUSH PosnX
        PUSH EditY
        CALL FWORD PTR _SetXY
        CMP EAX, 0
        JNE EditDone

EdLn02:
        PUSH EditX              ;Display current string
        PUSH EditY
        PUSH pEdString
        PUSH ddSzMax
        PUSH dEditAttr          ;Attribute they selected
        CALL FWORD PTR _PutVidChars
        CMP EAX, 0
        JNE EditDone
EdLn03:
        LEA EAX, KeyCode
        PUSH EAX
        CMP ddVidOwner, 2       ;Debugger???
        JE EdLn035
        PUSH 1                  ;Wait for a key
        CALL FWORD PTR _ReadKbd ;Get a key
        JMP SHORT EdLn036
EdLn035:
        CALL ReadDbgKbd
EdLn036:
        MOV EAX, KeyCode
        AND EAX, 07Fh
        OR  EAX, EAX
        JZ  EdLn03
        CMP EAX, 08h            ;BackSpace?
        JNE EdLn04              ;No - Next test
EdLn037:
        CMP ddSzCrnt, 0
        JE EdLn01
        DEC PosnX
        DEC ddSzCrnt
        MOV ESI, pEdString
        MOV ECX, ddSzCrnt
        MOV BYTE PTR [ESI+ECX], 20h
        JMP Edln01
EdLn04: CMP EAX, 03h            ;Left?
        JNE EdLn045             ;No - Next test
        JMP EdLn037
EdLn045:
        CMP EAX, 83h            ;Num-Left?
        JNE EdLn046             ;No - Next test
```

```
        JMP EdLn037
EdLn046:
        CMP EAX, 0Dh            ;CR?
        JNE EdLn05              ;No - Next test
        JMP EdLn07
EdLn05: CMP EAX, 1Bh            ;Escape?
        JNE EdLn06              ;No - Next test
        JMP EdLn07
EdLn06:
        CMP EAX, 7Eh            ;Is it above text?
        JA  EdLn07              ;Yes, Exit!
        CMP EAX, 20h            ;Is it below text??
        JB  EdLn07              ;Yes, Exit
        MOV ESI, pEdString      ;It's really a char!
        MOV ECX, ddSzCrnt
        MOV BYTE PTR [ESI+ECX], AL
        MOV ECX, ddSzMax
        CMP ddSzCrnt, ECX
        JAE EdLn01
        INC PosnX
        INC ddSzCrnt
        JMP EdLn01
EdLn07:
        MOV ESI,pExitKeyRet
        MOV [ESI], AL
        MOV ESI,pddSzRet
        MOV EAX, ddSzCrnt
        MOV [ESI], EAX

        PUSH EditX              ;Display current string w/Norm Attrs
        PUSH EditY
        PUSH pEdString
        PUSH ddSzMax
        CALL GetpCrntJCB        ;Leaves ptr to current JCB in EAX
        MOV EBX, [EAX+NormAttr]
        PUSH EBX                ;Normal Attribute from JCB
        CALL FWORD PTR _PutVidChars ;Ignore error (we are leaving anyway)
        XOR EAX, EAX
        JMP EditDone
BadEdit:
        MOV EAX, ErcEditParam
EditDone:
        MOV ESP,EBP             ;
        POP EBP                 ;
        RETF 24
```

Relating MMURTL's Video Code to Your Operating System

The video hardware on your platform will determine how much work you have to do to implement a video interface. From this chapter, you can see that simplicity was my goal.

In this age of graphics, you no doubt will want to experiment with both character-based and graphical interfaces. I recommend you keep the virtual character video concept in mind, because it allows both character-based and graphical interfaces to coexist on a system.

27

File System Code

The MMURTL FAT file system is truly no great accomplishment. Reverse engineering provides some satisfaction, but never the amount that an original design would.

There are better file systems to be designed than the one that came with a 15-year-old operating system, but none are as widespread as the MS-DOS FAT file system.

How MMURTL Handles FAT

The physical disk layout, as seen from the disk controller's standpoint, is as follows:

Cylinder numbers run from 0 to `nMaxCyls`–1.
Head numbers run from 0 to `nMaxheads`–1.
Sector numbers run from 1 to `nMaxSectorsPerTrack`.

Physical (Absolute) Sector Numbers

Physical sector numbers (absolute) begin at Cylinder 0, Head 0, Sector 1. As the physical sector number rolls over (`nMaxSectorsPerTrack+1`), the head number is incremented, which moves you to the next track (same cylinder, next head). When the head number rolls over (`nMaxHeads` is reached), the cylinder number is incremented.

Track and *cylinder* are not interchangeable terms in what I have said here. If you have six heads on your drive, you have six tracks per cylinder. This can be confusing because many books and documents use these terms interchangeably. You can too, so long as you know that's what you're doing.

Hidden Sectors

MS-DOS reserves a section of the physical hard disk. This area is called the hidden sectors. This area is usually the very first track on the disk (beginning at Cylinder 0, Head 0, Sector 1). The partition tables are kept at the very end of the first sector in this hidden area (offset 01BEh in the first sector, to be exact).

The partition tables are 16-byte entries describing "logical" sections of the disk that can be treated as separate drives. Usually there are no "hidden sectors" on floppy disks, nor are there any partition tables.

Logical Block Address (LBA)

MMURTL device drivers treat the entire disk as a single physical drive. The MMURTL file system reads the partition tables, then sets up the device driver to span the entire physical disk as 0 to `nMaxBlockNumbers`–1. A block number is referred to as the *Logical Block Address* (LBA) and is the value passed in to the `DeviceOp` call for the MMURTL hard/floppy disk device drivers (LBAs are used with all MMURTL devices).

Do not confuse MMURTL's LBA for the sector number in an MS-DOS logical drive. MMURTL refers to these as "logical blocks" because you still have to convert them into physical cylinder, head, and sector to retrieve the data.

MS-DOS Boot Sector

The first sector of an MS-DOS logical drive is its boot sector. Each of the MS-DOS logical partitions will have a boot sector, although only the first will be marked as bootable (if any are).

Its position on the disk is calculated from the partition table information.

File System Initialization

The MMURTL-FAT file system reads the partition table and saves the starting LBA and length of each of DOS logical disk that is found. Armed with this information, MMURTL can access each of the DOS logical disks as a separate disk drive.

To maintain some sanity, the MMURTL file system gives all of its logical drives a letter, just as does MS-DOS. MMURTL supports two floppy drives (A and B) and up to eight logical hard disks (C through J). All information on the logical drives are kept in an array of records (`Ldrvs`). This includes the logical-letter-to-physical-drive conversions.

After you have the layout of each of the partitions, you read the boot sector from the first DOS logical drive. The boot sector contains several pieces of important information about the drive geometry (numbers of heads, sectors per track, and so on), which are also placed in the logical drive structures.

After you have the drive geometry information, you set up the MMURTL device driver. This tells the device driver how many cylinders, heads, and sectors per track are on the physical disk. Until this is done, the device driver assumes a minimum drive size, and you should only read the partition table (or boot sector, if no partition table is on the disk). This provides enough information to do a `DeviceInit` call to set up proper drive geometry.

If you were building a loadable file system to replace the one that's included in MMURTL, you would call your routine to initialize the file system very early in the main program block. You must not service file system requests until this is done.

File System Listing

The file system implementation in Listing 27.1 has ample comments to describe the purpose of each function. An important concept to note is that the file system itself is a separate task that runs at a relatively high priority of 5.

At the very end of Listing 27.1, you will find a function called `InitFS()`, which is called from the Monitor to initialize the file system. All resources are allocated in this function before we spawn the new file system task and register the service with the operating system.

Listing 27.1. MS-DOS FAT-compatible file system code.

```
#define U32 unsigned long
#define U16 unsigned int
#define U8  unsigned char
#define S32 long
#define S16 int
#define S8  char
#define TRUE 1
#define FALSE 0

/********** MMURTL Public Prototypes **************/

/* From MKernel */

extern far AllocExch(long *pExchRet);
extern far U32 GetTSSExch(U32  *pExchRet);
extern far SpawnTask(char *pEntry,
                     long dPriority,
                     long fDebug,
                     char *pStack,
                     long fOSCode);
extern far long WaitMsg(long Exch, char *pMsgRet);
extern far long CheckMsg(long Exch, char *pMsgRet);
extern far long Request(unsigned char *pSvcName,
                        unsigned int  wSvcCode,
                        unsigned long dRespExch,
                        unsigned long *pRqHndlRet,
                        unsigned long dnpSend,
                        unsigned char *pData1,
                        unsigned long dcbData1,
                        unsigned char *pData2,
                        unsigned long dcbData2,
                        unsigned long dData0,
                        unsigned long dData1,
                        unsigned long dData2);

extern far long Respond(long dRqHndl, long dStatRet);

/* From MData */
extern far void CopyData(U8 *pSource, U8 *pDestination, U32 dBytes);
extern far void FillData(U8 *pDest, U32 cBytes, U8 bFill);
extern far long CompareNCS(U8 *pS1, U8 *pS2, U32 dSize);

/* From MTimer.h */
extern far long GetCMOSTime(long *pTimeRet);
extern far long GetCMOSDate(long *pTimeRet);
extern far long GetTimerTick(long *pTickRet);

/* From MVid.h */
extern far long TTYOut (char *pTextOut, long ddTextOut, long ddAttrib);
extern far long GetNormVid(long *pNormVidRet);
```

```
#include "MKbd.h"

/* From MDevDrv */
extern far U32  DeviceOp(U32  dDevice,
                         U32  dOpNum,
                         U32  dLBA,
                         U32  dnBlocks,
                         U8  *pData);

extern far U32  DeviceStat(U32  dDevice,
                           S8 * pStatRet,
                           U32  dStatusMax,
                           U32  *pdSatusRet);

extern far U32  DeviceInit(U32  dDevNum,
                           S8  *pInitData,
                           U32   sdInitData);

/* From MMemory.h */
extern far U32 AllocOSPage(U32 nPages, U8 *ppMemRet);
extern far U32 DeAllocPage(U8 *pOrigMem, U32 nPages);

/* From MJob.h */
extern far U32 GetPath(long JobNum, char *pPathRet, long *pdcbPathRet);
extern far U32 RegisterSvc(S8 *pName, U32 Exch);

/* NEAR support for debugging */

extern long xprintf(char *fmt, ...);
extern U32 Dump(unsigned char *pb, long cb);

/* File System error codes */

#define ErcOK             0      /* Alls Well */
#define ErcEOF            1      /* DUH... The END */
#define ErcBadSvcCode     32     /* Service doesn't handle that code */

#define ErcBadFileSpec    200    /* invalid file spec (not correct format)*/
#define ErcNoSuchDrive    201    /* Try another letter bozo */
#define ErcNotAFile       202    /* Open a directory?? NOT */
#define ErcNoSuchFile     203    /* No can do! It ain't there...*/
#define ErcNoSuchDir      204    /* Ain't no such dir... */
#define ErcReadOnly       205    /* You can't modify it bubba */
#define ErcNoFreeFCB      206    /* We're really hurtin... */
#define ErcBadOpenMode    207    /* Say what? Mode??? */
#define ErcFileInUse      208    /* File is open in an incompatible mode */
#define ErcNoFreeFUB      209    /* Sorry, out of File User Blocks */
#define ErcBadFileHandle  210    /* WHOAAA, bad handle buddy! */
#define ErcBrokenFile     211    /* Cluster chain broken on file */
#define ErcBadFCB         213    /* We got REAL problems... */
#define ErcStreamFile     214    /* Operation not allowed on Stream File */
#define ErcBlockFile      215    /* Operation not allowed on Block File */
#define ErcBeyondEOF      217    /* SetLFA or Read/WriteBlock beyond EOF */
#define ErcNoParTable     218    /* No partiton table found on disk!!! */
#define ErcBadFATClstr    220    /* File system screwed up (or your disk) */
#define ErcRenameDrv      222    /* They have tried to rename across Dir/Vol*/
#define ErcRenameDir      223    /* They have tried to rename across Dir/Vol*/
```

continues

Listing 27.1. continued

```
#define ErcNoMatch        224    /* No matching directory entry */

#define ErcWriteOnly      225    /* Attempt to read write-only device */
#define ErcDupName        226    /* Name exists as a file or dir already */
#define ErcNotSupported   227    /* Not supported on this file  */
#define ErcRootFull       228    /* The Root Directory is Full  */
#define ErcDiskFull       230    /* No more free CLUSTERS!!!  */

#define ErcNewMedia       605    /* for floppy mounting from FDD */

/*************** FAT Buffer control structure *********************/

/*
   The Fat structures are for keeping track of the FAT buffers.
   We never want to have more than one copy of a FAT sector in
   memory at one time, and we also never want to read one when
   it's already here (a waste of time)!  We also keep track
   of the last time it was used and deallocate the oldest (LRU -
   Least Recently Used). Initially filling out the Fat control
   structure is part of the file system initialization.  If the
   FAT sector we are in has been modified (data written to clusters
   in it & FAT updated) we write it ASAP!
   Each FAT buffer is 1 sector long, except the first one which
   is 3 sectors for floppies (FAT12 types). This is because the
   FAT12 entries span sectors!
*/

#define nFATBufs 17      /* 1 Static for floppies + 16 * 512 = 8192, 2 pages */

static struct fattype {                /* */
    U8  *pBuf;           /* points to beginning of fat buffer  */
    U32 LastUsed;        /* Tick when last used (0 = Never) */
    U32 LBASect;         /* LBA of first FAT sect in buf (where it came from) */
    U16 iClstrStart;     /* Starting cluster for each buf  */
    U8  Drive;           /* LDrive this FAT sector is from */
    U8  fModLock;        /* Bit 0 = Modified, bit 1 = Locked  */
    };

static struct fattype Fat[nFATBufs];     /* 16 bytes * 17 */

/* We read 3 sectors worth of floppy fat buf in because cluster
entries span sectors
*/

U8 FatBufA[1536];  /* floppy fat buffer */

#define FATMOD  0x01
#define FATLOCK 0x02

/*************** File Contol Block Structures (FCBs) **********/
/* One FCB is allocated and filled out for each file that is open.
   The actual directory entry structure from the disk is embedded
   in the FCB so it can be copied directly to/from the directory
   sector on the disk.
*/
```

```
#define nFCBs 128
#define sFCB  64

static struct FCB {
    S8  Name[8];        /* From here to Filesize is copy of DirEnt */
    S8  Ext[3];
    S8  Attr;           /* from MS-DOS */
    U8  Resvd1[10];     /* ????????  */
    U16 Time;           /* Only changed when created or updated */
    U16 Date;
    U16 StartClstr;     /* At least one per file!! */
    U32 FileSize;       /* last entry in FAT Dir Ent (32 bytes) */
    U32 LBADirSect;     /* LBA of directory sector this is from */
    U16 oSectDirEnt;    /* Offset in sector for the dir entry */
    U8  Ldrv;           /* Logical drive this is on (A-J, 0-9) */
    U8  Mode;           /* 0 or 1 (Read or Modify). */
    U8  nUsers;         /* Active FUBs for this file (255 MAX). 0= Free FCB */
    U8  fMod;           /* This file was modified! */
    U8  Resvd[22];      /* Out to 64 bytes */
    };

static struct FCB *paFCB;       /* a pointer to array of allocated FCBs. */
static struct FCB *pFCB;        /* pointer to one FCB */

/********************** File User Blocks *************************/

/* Each user of an open file is assigned a FUB.  The FUB number is the
   filehandle (beginning with 3). ) 0, 1 & 2 are reserved for NUL,
   KBD and VID devices.
*/

#define nFUBs 128
#define sFUB  32

/* The FUB contains information on a file related to a user's view
   of the file.  It is used to hold information on files opened
   in stream and block mode. Three important fields in the FUB are:
   LFABuf - LFA of first byte in buffer for a stream file.
   Clstr - Clstr of last block read or stream fill.
   LFAClstr - LFA of first byte in Clstr.

   LFAClstr and Clstr give us a relative starting point when
   reading a file from disk.  If we didn't save this information
   on the last access, we would have to "run" the cluster chain
   everytime we wanted to read or access a file beyond the last
   point we read.
*/

struct FUB {
    U16 Job;            /* User's Job Number. 0 if FUB is free. */
    U16 iFCB;           /* FCB number for this file (0 to nFCBs-1) */
    U32 CrntLFA;        /* Current Logical File Address (File Ptr) */
    U8  *pBuf;          /* Ptr to buffer if stream mode */
    U32 sBuf;           /* Size of buffer for Stream file in bytes */
    U32 LFABuf;         /* S-First LFA in Clstr Buffer */
    U32 LFAClstr;       /* LFA of Clstr (below). */
    U16 Clstr;          /* Last Cluster read */
```

continues

Listing 27.1. continued

```
    U8  fModified;      /* Data in buffer was modified */
    U8  fStream;        /* NonZero for STREAM mode */
    U8  Rsvd[4];        /* Pad to 32 bytes */
    };

static struct FUB *paFUB;       /* a pointer to allocated FUBs. Set up at init. */
static struct FUB *pFUB;        /* a pointer to allocated FUBs. Set up at init. */

/* Boot sector info (62 byte structure) */
struct fsbtype {
      U8  Jmp[3];
      U8  OEMname[8];
      U16 bps;
      U8  SecPerClstr;
      U16 ResSectors;
      U8  FATs;
      U16 RootDirEnts;
      U16 Sectors;
      U8  Media;
      U16 SecPerFAT;
      U16 SecPerTrack;
      U16 Heads;
      U32 HiddenSecs;
      U32 HugeSecs;
      U8  DriveNum;
      U8  Rsvd1;
      U8  BootSig;
      U32 VolID;
      U8  VolLabel[11];
      U8  FileSysType[8];       /* 62 bytes */
      };
static struct fsbtype  fsb;


/* Partition Table Entry info. 16 bytes */
struct partent {
  U8  fBootable;
  U8  HeadStart;
  U8  SecStart;
  U8  CylStart;
  U8  FATType;
  U8  HeadEnd;
  U8  SecEnd;
  U8  CylEnd;
  U32 nFirstSector;
  U32 nSectorsTotal;
  };

static struct partent partab[4];    /* 4 partition table entries 64 bytes */
static U16 partsig;

/* Bit definitions in attribute field for a directory entry */

#define ATTRNORM  0x00
#define READONLY  0x01
#define HIDDEN    0x02
```

```
#define SYSTEM    0x04
#define VOLNAME   0x08
#define DIRECTORY 0x10
#define ARCHIVE   0x20

/* Directory Entry Record, 32 bytes */

struct dirstruct {
    U8  Name[8];
    U8  Ext[3];
    U8  Attr;
    U8  Rsvd[10];
    U16 Time;
    U16 Date;
    U16 StartClstr;
    U32 FileSize;
    };

static struct dirstruct  dirent;

static struct dirstruct *pDirEnt;       /* a pointer to a dir entry */

/* When a file is opened, the filename is parsed into an array
   to facilitate searching the directory tree.  IN MS-DOS all
   dir and file names are SPACE padded (20h).  The FileSpec array
   contains the fully parsed path of the file.  For instance,
   If you were to open "A:\Dog\Food\IsGood.txt" the FileSpec
   array would look like this:
   FileSpec[0] = "DOG        "
   FileSpec[1] = "FOOD       "
   FileSpec[2] = "ISGOOD  TXT"
   FileSpec[3][0] = NULL;
   Note that the DOT is not inlcuded (it's not in the DOS directory
   either), and the next unused FileSpec entry contains NULL in the
   first byte.  SpecDepth tells us how many directories deep the
   name goes.
*/

static U8 FDrive                    /* Drive parsed from file operation */
static U8 FileSpec[7][11];          /* Hierarchy from file spec parsing */
static U8 SpecDepth;                /* Depth of parse (0=Root File) */

/* Used for Rename */
static U8 FDrive1                   /* Drive parsed from file operation */
static U8 FileSpec1[7][11];     /* Hierarchy from file spec parsing */
static U8 SpecDepth1;               /* Depth of parse (0=Root File) */

/* raw sector buffer for all kinds of stuff */

static U8  abRawSector[516];
static U8  abTmpSector[516];
static U8  abDirSectBuf[516];

/* These arrays keep track of physical drive data (0-4). */
#define nPDrvs 4

static struct phydrv {
    U32 nHeads;         /* heads per drives   */
```

continues

Listing 27.1. continued

```
    U32 nSecPerTrk;     /* Sectors per track  */
    U16 BS1Cyl;         /* Cyl of 1st boot sector on disk */
    U8  BS1Head;        /* Head of 1st boot sector on disk */
    U8  BS1Sect;        /* Sector of 1st boot sector on disk */
    }

static struct phydrv  PDrvs[nPDrvs];

/* This array of structures keeps track of logical drive data (A-J). */

#define nLDrvs 10

static struct ldrvtype {
    U32 LBA0;           /* lba for Start of LDrive (bootSect) */
    U32 LBAData;        /* lba for Start of Data Area */
    U32 LBAMax;         /* Max lba for logical drive */
    U32 LBARoot;        /* lba of the Root directory */
    U32 LBAFAT;         /* lba of first FAT */
    U16 nHeads;         /* Setup after boot sector is read */
    U16 nSecPerTrk;     /* Setup after boot sector is read */
    U16 nRootDirEnt;    /* Number of Root directory entries */
    U16 sFAT;           /* nSectors in a FAT */
    U8  DevNum;         /* Device Number for this ldrv FF = NONE */
    U8  SecPerClstr;    /* For each logical drive */
    U8  nFATS;          /* number of FATs */
    U8  fFAT16;         /* True for FAT16 else FAT12 */
    };

static struct ldrvtype  Ldrv[nLDrvs];

/* This is the Hard Disk Device Status record.
   It is peculiar to the HD Drvr */

struct hddevtype{
  U32 erc;
  U32 blocks_done;
  U32 BlocksMax;
  U8  fNewMedia;
  U8  type_now;     /* current fdisk_table for drive selected */
  U8  resvd0[2];    /* padding for DWord align  */
  U32 nCyl;         /* total physical cylinders (we really don't care) */
  U32 nHead;        /* total heads on device    */
  U32 nSectors;     /* Sectors per track        */
  U32 nBPS;         /* Number of bytes per sect.  32 bytes out to here.*/
  U32 LastRecalErc0;
  U32 LastSeekErc0;
  U8  LastStatByte0;
  U8  LastErcByte0;
  U8  fIntOnReset;  /* Interrupt was received on HDC_RESET */
  U8  filler0;
  U32 LastRecalErc1;
  U32 LastSeekErc1;
  U8  LastStatByte1;
  U8  LastErcByte1;
  U8  ResetStatByte;    /* Status Byte immediately after RESET */
  U8  filler1;
```

```
  U32 resvd1[2];    /* out to 64 bytes */
  };

static struct hddevtype   HDDevStat;

/* This is the Floppy Device Status record.
   It is peculiar to the FD Drvr */

struct fdstattype{
  U32 erc;          /* Last Error from device */
  U32 blocks_done;
  U32 BlocksMax;
  U8 fNewMedia;
  U8 type_now;      /* current fdisk_table for drive selected */
  U8 resvd1[2];     /* padding for DWord align  */
  U32 nCyl;         /* total physical cylinders */
  U32 nHead;        /* total heads on device    */
  U32 nSectors;     /* Sectors per track        */
  U32 nBPS;         /* Number of bytes per sect */
  U8 params[16];    /* begin device specific fields */
  U8 STATUS[8];     /* status returned from FDC (for user status) */
  U32 resvd3;
  U32 resvd4;       /* 64 bytes total */
  };

static struct fdstattype  FDDevStat;

static long FSysStack[512]; /* 2048 byte stack for Fsys task */

static long FSysExch;

struct reqtype {            /* 64 byte request block structure */
    long ServiceExch;
    long RespExch;
    long RqOwnerJob;
    long ServiceRoute;
    char *pRqHndlRet;
    long dData0;
    long dData1;
    long dData2;
    int  ServiceCode;
    char npSend;
    char npRecv;
    char *pData1;
    long cbData1;
    char *pData2;
    long cbData2;
    long RQBRsvd1;
    long RQBRsvd2;
    long RQBRsvd3;
    };

static struct reqtype *pRQB;

static char *fsysname = "FILESYSM";

static unsigned long keycode;          /* for testing */
```

continues

Listing 27.1. continued

```
/*========================= BEGIN CODE ============================*/

/***********************************************
 Called from read_PE, this gets the starting
 cylinder, head and sector for the first boot
 sector on a physical drive and stores it in the
 phydrv array.  d is the drive, i is the index
 into the partition table we read in.
***********************************************/

static void GetBSInfo(U32 d, U32 i)
{
 PDrvs[d].BS1Head = partab[i].HeadStart;
 PDrvs[d].BS1Sect = partab[i].SecStart;
 PDrvs[d].BS1Cyl  = partab[i].CylStart;

 if (!i)
 {      /* primary partition info - use it for PDrv info */
     PDrvs[d].nHeads = partab[i].HeadEnd;
     PDrvs[d].nSecPerTrk = partab[i].nFirstSector & 0xff;
 }
}

/** InitFloppy *********************************
 This gets status from the floppy drive (device ld)
 and sets the physical & logical drive parameters
 for the type. It is called when the file system
 is first initialized and when there has been
 an error on the floppy.
***********************************************/

static U32 StatFloppy(U8 ld)
{
U32 erc, i;

/* Set gets status for the floppy type from the FDD and
   sets logical paramters for Ldrvs.
*/

Ldrv[0].DevNum= 10;      /* Device Numbers for floppies */
Ldrv[1].DevNum= 11;

erc = DeviceStat(ld+10, &FDDevStat, 64, &i);
if (!erc)
{
    PDrvs[ld].nHeads = FDDevStat.nHead;
    PDrvs[ld].nSecPerTrk = FDDevStat.nSectors;
    Ldrv[ld].LBA0 = 0;       /* Floppy Boot Sector - always 0 */
    Ldrv[ld].LBAMax= FDDevStat.BlocksMax-1; /* Max lba for logical drive 0 */

    Ldrv[ld].nHeads = FDDevStat.nHead;
    Ldrv[ld].nSecPerTrk = FDDevStat.nSectors;

    erc = 0;
}
else
   Ldrv[ld].DevNum = 0xff;
```

```
 return erc;
}

/***********************************************
 Reads the partition table entries from hard
 drives and sets up some of the the logical
 drive array variables for hard Disks.
 It also saves first cylinder, head and sector
 of the first partiton on each physical drive
 so we can get more info for the LDrv arrays
 from the boot sector of that partition.
***********************************************/

static U32 read_PE(void)
{
U32 erc, ercD12, ercD13, i, j;
U8 fFound1, fFound2;

fFound1 = 0;        /* Have we found first valid partition on drive */
fFound2 = 0;

/* Set defaults for 4 physical drives. This info will be set
   correctly when the partition table and boot sectors are read.
*/

for (i=2; i< nLDrvs; i++)
{   /* default to no logical hard drives */
   Ldrv[i].DevNum = 0xff;
}

i = 2;      /* first Logical Number for hard drives "C" */

for (j=2; j<4; j++)
{   /* Array index Numbers for 2 physical hard Disks */

  erc = DeviceOp(j+10, 1, 0, 1, abRawSector); /* add 10 for Disk device nums */
  if (j==2) ercD12 = erc;
  else ercD13 = erc;

  if (!erc)
  {
    CopyData(&abRawSector[0x01fe], &partsig, 2);

    /* It MUST have a partition table or we can't use it! */

    if (partsig != 0xAA55) return ErcNoParTable;

    CopyData(&abRawSector[0x01be], &partab[0].fBootable, 64);

/*
     Dump(&partab[0].fBootable, 64);
     ReadKbd(&keycode, 1);
*/

    if (partab[0].nSectorsTotal > 0)
    {
     Ldrv[i].LBA0 =partab[0].nFirstSector;  /* lba for Start of LDrv (bootSect) */
```

continues

Listing 27.1. continued

```
    Ldrv[i].LBAMax =partab[0].nSectorsTotal;   /* Max lba for logical drive */
    Ldrv[i].LBAMax =partab[0].nSectorsTotal;   /* Max lba for logical drive */
    if (partab[0].FATType > 3)
       Ldrv[i].fFAT16 = 1;
    Ldrv[i].DevNum = j+10;
    if ((j==2) && (!fFound1))
    { GetBSInfo(2, 0); fFound1=1; }
      if ((j==3) && (!fFound2))
    { GetBSInfo(3, 0); fFound2=1; }
      i++;                  /* if valid partition go to next LDrv */
    }

   if (partab[1].nSectorsTotal > 0)
   {
    Ldrv[i].LBA0   = partab[1].nFirstSector;
    Ldrv[i].LBAMax = partab[1].nSectorsTotal;
    if (partab[1].FATType > 3)
       Ldrv[i].fFAT16 = 1;
    Ldrv[i].DevNum = j+10;
    if ((j==2) && (!fFound1)) { GetBSInfo(2, 1); fFound1=1; }
    if ((j==3) && (!fFound2)) { GetBSInfo(3, 1); fFound2=1; }
    i++;                    /* if we had a valid partition go to next */
   }

   if (partab[2].nSectorsTotal > 0)
   {
    Ldrv[i].LBA0   = partab[2].nFirstSector;
    Ldrv[i].LBAMax = partab[2].nSectorsTotal;
    if (partab[2].FATType > 3)
       Ldrv[i].fFAT16 = 1;
    Ldrv[i].DevNum = j+10;
    if ((j==2) && (!fFound1)) { GetBSInfo(2, 2); fFound1=1; }
    if ((j==3) && (!fFound2)) { GetBSInfo(3, 2); fFound2=1; }
    i++;                    /* if we had a valid partition go to next */
   }

   if (partab[3].nSectorsTotal > 0)
   {
    Ldrv[i].LBA0   = partab[3].nFirstSector;
    Ldrv[i].LBAMax = partab[3].nSectorsTotal;
    if (partab[3].FATType > 3)
       Ldrv[i].fFAT16 = 1;
    Ldrv[i].DevNum = j+10;
    if ((j==2) && (!fFound1))
    {
       GetBSInfo(2, 3);
       fFound1=1;
    }
    if ((j==3) && (!fFound2))
    {
       GetBSInfo(3, 3);
       fFound2=1;
    }
    i++;                    /* if we had a valid partition go to next */
    }
   }
}
```

```
 if (ercD12) return ercD12;     /* Device 13 may not exist */
 else return 0;
}

/*****************************************************************
  Reads in the first boot sector from each physical drive to get
  drive geometry info not available in partition table.  This includes
  number of heads and sectors per track.  Then we call DeviceInit
  for each physical device to set its internal drive geometry.
  This must be done before we even try to read the other boot sectors
  if the disk has multiple partitions (otherwise it fails).
*****************************************************************/

static U32 SetDriveGeometry(U32 d)      /* d is the device number (12 or 13) */
{
U32 erc, i;

  if (d==12)
  {
    erc =  DeviceStat(12, &HDDevStat, 64, &i);
    if (!erc)
    {
      HDDevStat.nHead = PDrvs[2].nHeads;
      HDDevStat.nSectors = PDrvs[2].nSecPerTrk;
      erc = DeviceInit(12, &HDDevStat, 64); /* Set up drive geometry */
    }
  }

  if (d==13)
  {
    erc =  DeviceStat(13, &HDDevStat, 64, &i);
    if (!erc)
    {
      HDDevStat.nHead = PDrvs[3].nHeads;
      HDDevStat.nSectors = PDrvs[3].nSecPerTrk;
      erc = DeviceInit(13, &HDDevStat, 64); /* Set up drive geometry */
    }
  }

return erc;
}

/*****************************************************************
  Read boot sector from logical drive (i) and sets up logical and
  physical drive array variables for the FAT file system found on
  the logical drive (described in the boot sector).
*****************************************************************/

static U32 read_BS(U32 i)
{
U32 erc, j;

if (Ldrv[i].DevNum != 0xff)
{

    j = Ldrv[i].DevNum;         /* j is MMURTL Device number */
```

continues

Listing 27.1. continued

```
    erc = DeviceOp(j, 1, Ldrv[i].LBA0, 1, abRawSector);

    if ((erc==ErcNewMedia) && (i<2))
    {
        erc = DeviceOp(j, 1, Ldrv[i].LBA0, 1, abRawSector);
    }

    CopyData(abRawSector, &fsb.Jmp, 62);

    if (erc==0)
    {
       Ldrv[i].LBARoot     = fsb.ResSectors + Ldrv[i].LBA0 +
                             (fsb.FATs * fsb.SecPerFAT);
       Ldrv[i].nRootDirEnt = fsb.RootDirEnts;   /* n Root dir entries */
       Ldrv[i].SecPerClstr = fsb.SecPerClstr;
       Ldrv[i].nHeads      = fsb.Heads;
       Ldrv[i].nSecPerTrk  = fsb.SecPerTrack;
       Ldrv[i].sFAT        = fsb.SecPerFAT;     /* nSectors in a FAT */
       Ldrv[i].nFATS       = fsb.FATs;          /* number of FATs */
       Ldrv[i].LBAFAT      = Ldrv[i].LBA0 + fsb.ResSectors;
       Ldrv[i].LBAData     = Ldrv[i].LBARoot + (fsb.RootDirEnts / 16);
       if (fsb.FileSysType[4] == '2')
         Ldrv[i].fFAT16 = 0;

    } /* if erc */
} /* if valid logical device */
return 0;
}

/*******************************************************
  This gets the CMOS date & time and converts it into the
  format for the DOS FAT file system. This is two words
  with bits representing Year/Mo/day & Hr/Min/SecDIV2.
*******************************************************/
static void GetFATTime(U16 *pTimeRet, U16 *pDateRet)
{
U32 date, time;
U16 DDate, DTime, w;

    GetCMOSDate(&date);
    GetCMOSTime(&time);
    /* Do the date */
    DDate = (((date >> 12) & 0x0f) * 10) + ((date >> 8) & 0x0f); /* day */
    w = (((date >> 20) & 0x0f) * 10) + ((date>>16) & 0x0f) + 2;  /* month */
    DDate |= (w << 4);
    w = (((date >> 28) & 0x0f) * 10) + ((date >> 24)  & 0x0f);   /* year */
    DDate |= (w + 1900 - 1980) << 9;
    /* Do the time */
    DTime = (((((time >> 4) & 0x0f) * 10) + (time & 0x0f))/2);  /* secs/2 */
    w = (((time >> 12) & 0x0f) * 10) + ((time >> 8) & 0x0f);
    DTime |= (w << 5);                                          /* mins */
    w = (((time >> 20) & 0x0f) * 10) + ((time >> 16) & 0x0f);   /* hours */
    DTime |= (w << 11);
    *pTimeRet = DTime;
    *pDateRet = DDate;
}
```

```
/*****************************************************
  This updates a directory entry by reading in the
  sector it came from and placing the modifed entry
  into it then writing it back to disk.  The date is
  also updated at this time.
*****************************************************/
static U32 UpdateDirEnt(U32 iFCB)
{
U32 erc, i, j;
U8 Drive;
    Drive = paFCB[iFCB]->Ldrv;      /* What logical drive are we on? */
    i = paFCB[iFCB]->LBADirSect;    /* Sector on disk */
    j = paFCB[iFCB]->oSectDirEnt;   /* offset in sector */

    /* update time in dir entry */
    GetFATTime(&paFCB[iFCB].Time, &paFCB[iFCB].Date);

    /* Read sector into a buffer */
    erc = DeviceOp(Ldrv[Drive].DevNum, 1, i, 1, abDirSectBuf);

    if (!erc)
    {
        CopyData(&paFCB[iFCB], &abDirSectBuf[j], 32);
        erc = DeviceOp(Ldrv[Drive].DevNum, 2, i, 1, abDirSectBuf);
    }
  return erc;
}


/*****************************************************
  Checks the validity of a file handle and also
  returns the index to the FCB if the handle is OK.
  The function returns OK (0) if handle is good, else
  a proper error code is returned.
*****************************************************/
static U32 ValidateHandle(U32 dHandle, U32 *iFCBRet)
{
  /* get some checks out of the way first */

  if (dHandle < 4) return ErcBadFileHandle;
  if (dHandle >= nFUBs) return ErcBadFileHandle;
  if (!paFUB[dHandle].Job) return ErcBadFileHandle;

  /* Looks like a valid handle */
  *iFCBRet = paFUB[dHandle]->iFCB;
  return 0;
}

/*****************************************
  Returns absolute disk address for the
  cluster number you specify. This gives us
  the LBA of the first sector of data that
  the cluster number represents.
  The sector number is returned from the function.
Uses: Ldrv[CrntDrv].LBAData
      Ldrv[CrntDrv].SecPerClstr
*****************************************/
```

continues

Listing 27.1. continued

```
static U32 ClsToLBA(U16 Clstr, U8 Drive)
{
U32 LBA;

 Clstr-=2;      /* Minus 2 cause 0 and 1 are reserved clusters */
 LBA = Ldrv[Drive].SecPerClstr * Clstr;
 LBA += Ldrv[Drive].LBAData;
 return LBA;
}


/*****************************************************
  This writes out the specified FAT sector back into
  the FAT. It also checks to see if there is more
  than one copy of the fat and updates the second copy
  if it exists.
*****************************************************/
static U32 UpdateFAT(U32 iFAT)
{
U32 erc, i, k;
U8 Drive;

  erc = 0;
  if (Fat[iFAT].fModLock & FATMOD)
  { /* Modified?? */

    Drive = Fat[iFAT].Drive;        /* What logical drive are we on? */
    i = Fat[iFAT].LBASect;              /* Where to write it back */

    if (!iFAT)
    {           /* This is the floppy buffer [0] */
        /* set up to write up to 3 sectors from the buffer */

        if (i+2 < Ldrv[Drive].sFAT + Ldrv[Drive].LBAFAT)
            k = 3;
        else if (i+1 < Ldrv[Drive].sFAT + Ldrv[Drive].LBAFAT)
            k = 2;
        else
            k = 1;
    }
    else
        k=1;

    erc = DeviceOp(Ldrv[Drive].DevNum, 2, i, k, Fat[iFAT].pBuf);
    if (!erc)
    {
        Fat[iFAT].fModLock &= ~FATMOD;  /* Not modified anymore */
        if (Ldrv[Drive].nFATS > 1)
        {   /* 2 FATS! */
            /* if we have two FATS we must update the second fat
            also. This will be located directly aftrer the first
            FAT (by exactly LDrv.sFat sectors).
            */
            i+= Ldrv[Drive].sFAT;
            erc = DeviceOp(Ldrv[Drive].DevNum, 2, i, k, Fat[iFAT].pBuf);
        }
    }
```

```
  }
  return erc;
}

/*****************************************************
 Reads in the FAT sector that contains the Cluster we
 specified into a FAT buffer if it isn't already in
 one.  The index to the FAT buffer is returned.
 Returns Error if not in FAT.
 Uses: Ldrv[LDrive].LBAFAT
       Ldrv[LDrive].fFAT16
       Ldrv[LDrive].DevNum
 Each sector of the FAT contains 256 cluster entries
 for FAT16 types. To find it, we  Divide the cluster
 number by the count of entries (256), and add this
 to the beginning sector of the FAT.  It is SOOO
 important (for speed) to have the FAT sectors in
 memory, that we allocate the FAT buffers on a Least
 Recently Used (LRU) basis for hard disk drives.

 It's more complicated for a FAT12 types (floppies)
 because cluster entries span fat sectors (they have
 an odd number of nibbles). For this reason, we have
 one 3 sector fat buffer for fat12 devices (floppies).
 We fill it with up to 3 sectors. This is because the
 last entry may span the sectors and we must be able
 to read it.  There are 1024 cluster entries in
 a FAT12 3 sector buffer.
*****************************************************/

static U32 FindFatSect(U8 Drive, U16 Clstr, U32 *piFatRecRet, U8 fLock)
{
U32 i, j, k;
U32 first, oSector, erc, LRU, iLRU, iFound, Tick;
U16 MaxClstr;

  if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
  else
    MaxClstr = 0xff8;   /* FAT12 */

 if (Clstr >= MaxClstr)
    return(ErcEOF);

 if (Clstr < 2)
 {
    return (ErcBadFATClstr);
 }

 GetTimerTick(&Tick);

 erc = 0;       /* default to no error */

 /* Set oSector to offset of sector in FAT
    There are 256 cluster entries in 1 sector of a FAT16,
    and 1024 in a FAT12 (3 sectors)
 */
```

continues

Listing 27.1. continued

```
if (Ldrv[Drive].fFAT16)
{
   oSector = Clstr/256;
   first = Clstr-(Clstr%256);

   /* Set i to LBA of FAT sector we need by adding
      offset to beginning of FAT
   */

   i = oSector + Ldrv[Drive].LBAFAT;

   /* If FAT sector is out of range there's a BAD problem... */

   if (i >= Ldrv[Drive].sFAT + Ldrv[Drive].LBAFAT)
   {
       return (ErcBadFATClstr);
   }
   else
   {   /* Else we get it for them */

       /* Loop through the Fat bufs and see if it's in one already. */
       /* Save the index of the LRU in case it's not there. */
       /* Set iFound to index of FatBuf (if found). */
       /* Otherwise, Set up iLRU to indicate what the oldest buffer is  */

       iFound = 0xffffffff;
       LRU = 0xffffffff;   /* saves tick of oldest one so far */
       iLRU = 1;           /* default */
       for (j=1; j<nFATBufs; j++)
       {
           if (Fat[j].LastUsed > 0)
           {       /* Valid ? (ever been used) */
               if ((first == Fat[j].iClstrStart) &&
                   (Drive == Fat[j].Drive))
                   {
                       iFound = j;
                       if (fLock)
                           Fat[j].fModLock |= FATLOCK;
                       break;      /* Already IN! */
                   }
           }
           if (Fat[j].LastUsed < LRU)
           {
               LRU = Fat[j].LastUsed;
               iLRU = j;
           }
       }

       if (iFound != 0xffffffff)
       {                                   /* Its already in memory */
           Fat[j].LastUsed = Tick;         /* update LRU */
       }
       else
       {                                /* else put into oldest buffer */
           j = iLRU;

           /* Check to see if Fat[iLRU] is valid and has been
```

```
            modified. If it is, write it out before we read
            the next one into this buffer. This done by
            calling UpdateFAT(iFatRec).
         */
         if (Fat[j].fModLock & FATMOD)
             erc = UpdateFAT(j);

         if (!erc)
         {
             erc = DeviceOp(Ldrv[Drive].DevNum, 1, i, 1, Fat[j].pBuf);
             Fat[j].Drive = Drive;           /* Update Drive */
             Fat[j].LastUsed = Tick;         /* update LRU */
             Fat[j].iClstrStart = first;     /* update first cluster num */
             Fat[j].LBASect = i;             /* LBA this FAt sect came from */
         }
      }
   }
}

   /* This is for FAT12s */

else
{
   oSector = (Clstr/1024) * 3;  /* X3 cause we read 3 at a time */
   first = Clstr-(Clstr%1024);

   /* Set i to LBA of FAT sector we need by adding offset (oSector)
      to beginning of FAT */

   i = oSector + Ldrv[Drive].LBAFAT;
   j = 0;

   /* If FAT sector is out of range there's a BAD problem... */

   if (i >= Ldrv[Drive].sFAT)
       return (ErcBadFATClstr);
   else
   {   /* Else we get it for them */

       /* Check the single floppy fat buf and see if it's already there. */
       /* Set iFound to index of FatBuf (if found). */

       iFound = 0xffffffff;

       if (Fat[0].LastUsed > 0)
       {       /* Valid ? (nonzero means it's been used) */
           if ((first == Fat[0].iClstrStart) &&
               (Drive == Fat[0].Drive))
           {
               iFound = 0;
               if (fLock)
                   Fat[0].fModLock |= FATLOCK;
           }
       }

       if (iFound == 0xffffffff)
       {
```

continues

Listing 27.1. continued

```
            /* It's not the one we want or isn't there.
               Check to see if Fat[0] is valid and has been
               modified. If it is, write it out before we read
               the one we want into the buffer. This done by
               calling UpdateFAT(iFatRec).
            */
            if (Fat[0].fModLock & FATMOD)
                  erc = UpdateFAT(0);

            /* set up to read upto 3 sectors into buffer */

            if (i+2 < Ldrv[Drive].sFAT + Ldrv[Drive].LBAFAT)
                k = 3;
            else if (i+1 < Ldrv[Drive].sFAT + Ldrv[Drive].LBAFAT)
                k = 2;
            else
                k = 1;

            if (!erc)
            {
                erc = DeviceOp(Ldrv[Drive].DevNum, 1, i, k, Fat[0].pBuf);

                Fat[0].Drive = Drive;           /* Update Drive */
                Fat[0].LastUsed = Tick;         /* update LRU */
                Fat[0].iClstrStart = first;     /* update first cluster num */
                Fat[0].LBASect = i;     /* LBA this FAT sect came from */
            }
        }
    }
 }


 *piFatRecRet = j;  /* Buffer that holds the sector(s) */

 return (erc);      /* Disk error Bad news */
}

/*****************************************
  Returns the value found for this cluster
  entry in a fat sector buffer.  Values can be:
                              FAT16     FAT12
    Next entry in the chain (0002-FFF0 (002-FF0)
    Last entry in chain     (FFF8     )(FF8    )
    Available cluster       (0        )(0      )
    Bad Cluster             (FFF7     )(FF7    )
    (other values are reserved).
*****************************************/

static U32 GetClstrValue(U16 Clstr, U8 Drive, U8 fLock,
                         U16 *pValRet, U32 *iFatBufRet)
{
U32 erc, oClstr, iFat;
U16 ClstrVal, *pClstr;

    erc = FindFatSect(Drive, Clstr, &iFat, fLock);

    if (erc)
```

```
    {
        *pValRet= 0;
        return(erc);
    }

    pClstr = Fat[iFat].pBuf;
    oClstr = Clstr - Fat[iFat].iClstrStart;    /* offset into FatBuf */

    if (Ldrv[Drive].fFAT16)
    {   /* if drive is FAT16 type */
        pClstr += oClstr * 2;                   /* WORDS in */
        ClstrVal = *pClstr;
    }
        /* FAT12 entries are 1.5 bytes long (what a pain).
           This means we get the offset and see whether it
           is an odd or even byte, then take the proper nibble
           by ANDing or shifting.
        */
    else
    {                       /* a FAT12... */
        pClstr += oClstr + (oClstr/2);          /* 1.5 bytes in */
        ClstrVal = *pClstr;                 /* We have 16 bits */
        if (Clstr & 1)                          /* Odd, must shift */
            ClstrVal >>= 4;
        ClstrVal &= 0xfff;
    }
    *pValRet= ClstrVal;
    *iFatBufRet = iFat;

    return(erc);
}


/***********************************************
  Sets the value in Clstr to the value in
  NextClstr which will be one of the following
  values:                   FAT16     FAT12
    Next entry in the chain (0002-FFEF (002-FEF)
    Last entry in chain     (FFFF     )(FFF    )
    Available cluster       (0        )(0      )
    Bad Cluster             (FFF7     )(FF7    )
    (other vlaues are reserved).
  This marks the associated fat buffer as modified.
  This is the ONLY call that modifies a FAT buffer!
***********************************************/

static U32 SetClstrValue(U16 Clstr, U16 NewClstrVal, U8 Drive, U32 *iFatBufRet)
{
U32 erc, oClstr, iFat;
U16 ClstrVal, *pClstr, ClstrSave;

    erc = FindFatSect(Drive, Clstr, &iFat, 0);
    if (erc)
    {
        *iFatBufRet = 0;
        return(erc);
    }
```

continues

Listing 27.1. continued

```
    pClstr = Fat[iFat].pBuf;
    oClstr = Clstr - Fat[iFat].iClstrStart;    /* offset into FatBuf*/
    if (Ldrv[Drive].fFAT16)
    {   /* if drive is FAT16 type */
        pClstr += oClstr * 2;                   /* WORDS in */
        *pClstr = NewClstrVal;
    }
        /* FAT12 entries are 1.5 bytes long (remember??).
           SAVE THE CORRECT NIBBLE OF THE ADJACENT CLUSTER!!
        */
    else
    {                           /* a FAT12... */
        pClstr += oClstr + (oClstr/2);          /* 1.5 bytes in */
        ClstrSave = *pClstr;                        /* We have 16 bits */
        if (Clstr & 1)
        {                           /* Odd, must shift */
            NewClstrVal <<= 4;
            NewClstrVal &= 0xfff0;
            ClstrVal = (ClstrSave & 0x0F) ¦ NewClstrVal;
        }
        else
        {
            NewClstrVal &= 0x0fff;
            ClstrVal = (ClstrSave & 0xf000) ¦ NewClstrVal;
        }
        *pClstr = ClstrVal;
    }
    Fat[iFat].fModLock ¦= FATMOD;
    *iFatBufRet = iFat;
    return(erc);
}


/***********************************************
 Read the FAT and get the cluster number for the
 next cluster in the chain for the Clstr specifed.
 This returns 0 and an error if failed.
 0 is an illegal cluster number.
 Remember, the cluster you are on is actually the
 number of the next cluster in a linked list!
***********************************************/

static U32 NextFATClstr(U8 Drive, U16 Clstr, U16 *pNextClstrRet)
{
U32 erc, i;
U16 NextClstr;

    erc = GetClstrValue(Clstr, Drive, 0, &NextClstr, &i);

    if (erc)
    {
        *pNextClstrRet = 0;
        return(erc);
    }
    *pNextClstrRet = NextClstr;
    return(0);
}
```

```
/************************************************
 This allocates the next empty cluster on the disk
 to the tail of the clstr that is passed in.
 LastClstr is a valid cluster of a file or
 directory (and MUST be the last one).
 We error out if it isn't!
 This returns 0 and an error if it fails.
 Remember, the cluster you are on is actually the
 number of the next cluster in a linked list!
 This looks through the current and successive
 FAT sectors (if needed) to add to the file.
 A cluster is available to allocate if it is
 0.  This is strictly a first fit algorithm.
************************************************/

static U32 ExtendClstrChain(U8 Drive, U16 LastClstr, U16 *pNextClstrRet)
{
U32 erc, i, j, k;
U16 ClstrValue, MaxClstr, CrntClstr;
U8 fFound;

    if (Ldrv[Drive].fFAT16)
        MaxClstr = 0xfff8;
    else
        MaxClstr = 0xff8;   /* FAT12 */

    /* i is index to Fat with last sector of current chain */

    erc = GetClstrValue(LastClstr, Drive, 1, &ClstrValue, &i);
    if (erc)
    {
        *pNextClstrRet = 0;
        return(erc);
    }

    if (ClstrValue < MaxClstr)
    {       /* no need to extend it */
        *pNextClstrRet = ClstrValue;
        Fat[i].fModLock &= ~FATLOCK;        /* unlock it */
        return(0);
    }

    /* OK... now we have the Fat sector and the offset in the Fat
    buf of the last cluster allocated to this file.  Let's go
    further into the buffer and try to get an empty one.
    */

    CrntClstr = LastClstr;
    fFound = 0;
    while (!fFound)
    {
        ++CrntClstr;        /* next cluster */
        erc = GetClstrValue(CrntClstr, Drive, 0, &ClstrValue, &j);
        if (erc)
        {
            *pNextClstrRet = 0;
```

continues

Listing 27.1. continued

```
            Fat[i].fModLock &= ~FATLOCK;    /* unlock previous lastclstr */
             return(erc);
        }
        if (!ClstrValue)
        {
            fFound = 1;     /* found an empty one */
        }
    }

    if (fFound)
    {   /* CrntClstr is index to empty one */

        /* Set the LastCluster to point to the new cluster found */

        erc = SetClstrValue(LastClstr, CrntClstr, Drive, &k);
        if (erc)
        {
            *pNextClstrRet = 0;
            Fat[i].fModLock &= ~FATLOCK;    /* unlock previous lastclstr */
            return(erc);
        }
        Fat[k].fModLock &= ~FATLOCK;        /* unlock it */

        /* Set the newcluster to "end Cluster" chain value */

        erc = SetClstrValue(CrntClstr, 0xFFFF, Drive, &j);
    }
    *pNextClstrRet = CrntClstr;
    return(erc);
}

/***********************************************
 This truncates the file chain to the cluster
 specified (makes it the last cluster).
 This means we walk the rest of the chain setting
 all the entries to 0 (so they can be reallocated).
 This returns an error if failed.
***********************************************/

static U32 TruncClstrChain(U8 Drive, U16 Clstr)
{
U32 erc, i;
U16 MaxClstr, NextClstr, CrntClstr;

    if (Ldrv[Drive].fFAT16)
        MaxClstr = 0xfff8;
    else
        MaxClstr = 0xff8;   /* FAT12 */

    /* i will be index to FatRec with last sector of current chain */

    erc = GetClstrValue(Clstr, Drive, 0, &NextClstr, &i);
    if (erc)
        return(erc);

    if (NextClstr >= MaxClstr)
    {       /* no need to truncate it */
```

```
        return(0);                      /* It's already the end. */
    }

    /* OK... now we cut it off all the way down the chain.
    We start by placing MaxClstr in the last sector and
    then 0 in all entries to the end of the chain.
    */

    erc = GetClstrValue(Clstr, Drive, 0, &NextClstr, &i);
    if (erc)
        return(erc);
    erc = SetClstrValue(Clstr, 0xFFFF, Drive, &i);  /* new end of chain */
    if (erc)
        return(erc);

    while ((NextClstr) && (NextClstr < MaxClstr))
    {
        CrntClstr = NextClstr;
        erc = GetClstrValue(CrntClstr, Drive, 0, &NextClstr, &i);
        if (erc)
            return(erc);
        erc = SetClstrValue(CrntClstr, 0, Drive, &i);  /* Free it up */
        if (erc)
            return(erc);
    }

    /* DONE! */
    return(0);
}

/*******************************************************
  This finds the absolute cluster you want from the
  LFA in a particular file. The file handle must already
  be validated!  It also returns the relative LFA of
  the beginning of this cluster.
*******************************************************/

static U32 GetAbsoluteClstr(U32 dHandle, U32 dLFA,
                            U16 *pClstrRet, U32 *prLFARet)
{
U32 erc, iFCB, spc, bpc, rLFA;
U16 rClstrWant, rClstrNow, Clstr, MaxClstr;
U8 Drive;

  iFCB = paFUB[dHandle]->iFCB;
  Drive = paFCB[iFCB]->Ldrv;        /* What logical drive are we on? */
  spc = Ldrv[Drive].SecPerClstr;    /* sectors per cluster */
  bpc = spc * 512;                  /* bytes per cluster */

 if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
 else
    MaxClstr = 0xff8;   /* FAT12 */

/*
  Calculate relative by dividing cluster size in bytes by dLFA.
  If zero, we want the 1st cluster which is listed in the FCB.
```

continues

Listing 27.1. continued

```
  If it is greater than zero, we have to "walk the FAT cluster
  chain" until we reach the one we want, then read it in.

  The FUB fields LFAClstr and Clstr store the file LFA of the last
  cluster in this file that was read or written.  This means if the
  LFA is higher than the last read or written, we don't waste the
  time reading the whole chain. We start from where we are.

  The major difference is we may not be reading the first sector
  in the cluster. We figure this out from the dLFA as compared to
  LFAClstr.

*/

  rClstrWant = dLFA / bpc;                      /* Relative clstr they want */
  rClstrNow = paFUB[dHandle]->LFAClstr / bpc;   /* Rel 'Clstr' in FUB */

  if (rClstrWant < rClstrNow)
  {         /* Is it earlier in the file? */
    Clstr = paFCB[iFCB]->StartClstr;        /* Yes, start at the beginning */
    rClstrNow = 0;
    rLFA = 0;
  }
  else
  {
    Clstr = paFUB[dHandle]->Clstr;          /* No, start at current cluster */
    rLFA = paFUB[dHandle]->LFAClstr;        /* LFA of this cluster */
  }

  /* We need to run the cluster chain if rClstrNow < ClstrWant */

  while ((rClstrNow < rClstrWant) &&    /* haven't reach it yet */
         (Clstr < MaxClstr) &&          /* Not last cluster */
         (Clstr))
 {                  /* A valid cluster */
      erc = NextFATClstr(Drive, Clstr, &Clstr);
      if (erc)
        return(erc);
      ++rClstrNow;
      rLFA += bpc;
  }

  if (rClstrNow != rClstrWant)          /* Cluster chain appears broken... */
    return ErcBrokenFile;

  *pClstrRet = Clstr;
  *prLFARet = rLFA;
  return(0);
}


/*******************************************************
  SetFileSize sets the FileSize entry in the FCB for
  the handle specified.  This means we will allocate
  or deallocate clusters as necessary to satisfy this
  request. The file MUST be open in MODE MODIFY.
  This must be done before a file can be written
  to beyond current FileSize (Block and Stream types).
*******************************************************/
```

```
static U32 SetFileSizeM(U32 dHandle, U32 dSize)
{
U32 erc, i, iFCB, rLFA;
U32 CrntSize, nCrntClstrs, spc, bpc, nClstrsWant;
U16 Clstr;
U8 Drive;

  erc = ValidateHandle(dHandle, &iFCB);
  if (erc)
    return erc;
  if (!paFCB[iFCB]->Mode)
    return ErcReadOnly;

  Drive = paFCB[iFCB]->Ldrv;        /* What logical drive are we on? */
  spc = Ldrv[Drive].SecPerClstr;    /* sectors per cluster */
  bpc = spc * 512;                  /* bytes per cluster */

  /* Looks like it's valid to change the size */

  CrntSize = paFCB[iFCB]->FileSize;

  if (CrntSize == dSize)        /* No need to do anything! */
    return(0);

  nCrntClstrs = CrntSize/bpc;   /* nClusters currently  */
  if (CrntSize%bpc)
    nCrntClstrs++;

  if (!CrntSize)
    nCrntClstrs = 1;            /* ZERO length files have 1 Clstr! */

  nClstrsWant = dSize/bpc;      /* nClusters they we need  */
  if (dSize%bpc)
    nClstrsWant++;

  if (!dSize)
    nClstrsWant = 1;            /* ZERO length files have 1 Clstr! */


  if (nClstrsWant == nCrntClstrs)
    erc = 0;

  else if (nClstrsWant > nCrntClstrs)
  { /* Need to extend allocation */

    /* get the last cluster in the file */
    erc = GetAbsoluteClstr(dHandle, CrntSize, &Clstr, &rLFA);
    i = nCrntClstrs;
    while ((!erc) && (i < nClstrsWant))
    {
            erc = ExtendClstrChain(Drive, Clstr, &Clstr);
            i++;
    }
  }
  else if (nClstrsWant < nCrntClstrs)
  { /* Need to truncate it */
```

continues

Listing 27.1. continued

```
    /* Get to cluster where it should be truncated */

    erc = GetAbsoluteClstr(dHandle, dSize, &Clstr, &rLFA);
    if (!erc)
        erc = TruncClstrChain(Drive, Clstr);
    /* Now we must ensure that the cluster helper is NOT
    beyond EOF!
    */
    if (paFUB->LFAClstr > dSize)
    {
        paFUB->LFAClstr = 0;
        paFUB[dHandle]->Clstr = paFCB[iFCB]->StartClstr;
    }

  }
  if (!erc)
  {
    paFCB[iFCB]->FileSize = dSize;
    paFCB[iFCB]->fMod = 1;
  }

  return erc;
}



/*****************************************************************
 This searches a directory beginning at Clstr for pName and returns
 a pointer to the 32 byte dir entry which is in a temporary buffer.
 If not found, returns NIL. When searching directories, if the
 filename begins with a NULL the search need go no further!
*****************************************************************/

static U32 GetDirEnt(U8  *pName,
             U8  Drive,
             U16 Clstr,
             U32 *pLBARet,
             U32 *poEntRet,
             U8  **pEntRet)
{
unsigned long sector, i, j, k, erc;
U8 fFound, fEnd, *pEnt, *pStart;
U16 MaxClstr;

 j = Ldrv[Drive].SecPerClstr;       /* How many sectors per cluster */
 sector = ClsToLBA(Clstr, Drive);   /* absolute sector of first dir sector */
 if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
 else
    MaxClstr = 0xff8;   /* FAT12 */
 i = 0;
 fEnd=0;
 fFound=0;

  fFound= 0;
  while ((!fFound) && (!fEnd))
  {     /* while there are valid entries */
```

```
    if (i==j)
    {       /* reached last dir sector of this cluster */
        erc = NextFATClstr(Drive, Clstr, &Clstr);
        if (!erc)
        {
            if (Clstr >= MaxClstr)              /* last sector */
                return(ErcNoSuchFile);          /* not found */
            sector = ClsToLBA(Clstr, Drive);    /* LBA of next dir sector */
            i=0;
        }
        else
        {
            *pEntRet = 0;
            return(erc);
        }
    }

    erc = DeviceOp(Ldrv[Drive].DevNum, 1, sector++, 1, abDirSectBuf);
    if (erc)
        return(erc);

    ++i;        /* next sector in cluster */

    pEnt = &abDirSectBuf[0];
    pStart = pEnt;

    for (k=0; k<16; k++)
    {       /* 16 entries per sector */
        if (*pEnt==0)
        {           /* 0 in a DirEnt stops search */
            fEnd=1;
            break;
        }

        if (CompareNCS(pEnt, pName, 11) == -1)
        {
            fFound=1;
            *pLBARet = sector-1;        /* tell em what LBA of DirEnt */
            *poEntRet = pEnt-pStart;    /* Tell em offset in LBA */
            break;
        }
        pEnt+=32;   /* 32 byte per entry */
        }
    }
  if (fFound)
  {
    *pEntRet = pEnt;
    return(0);
  }
  else return (ErcNoSuchFile);
}


/*****************************************************
 This searches the ROOT directory for pName
 and returns a pointer to the 32 byte entry which
 is in a temporary buffer. If not found, returns NIL;
```

continues

Listing 27.1. continued

```
 When searching directories, if the filename begins
 with a NULL the search need go no futher!
*************************************************/

static U32 GetRootEnt(U8  *pName,
              U8  Drive,
              U32 *pLBARet,
              U32 *poEntRet,
              U8  **pEntRet)
{
unsigned long i, j, k, erc;
U8 fFound, fEnd, *pEnt, *pStart;

 i = Ldrv[Drive].LBARoot;
 j = Ldrv[Drive].nRootDirEnt;

 fFound = 0;
 fEnd = 0;
 while ((j) && (!fFound) && (!fEnd))
 {  /* while there are valid entries */

    erc = DeviceOp(Ldrv[Drive].DevNum, 1, i++, 1, abRawSector);

    if (erc)
        return(erc);

    pEnt = abRawSector;
    pStart = pEnt;
    for (k=0; k<16; k++)
    {
        if (*pEnt==0)
        {           /* 0 in a DirEnt stops search */
            fEnd=1;
            break;
        }
        if (CompareNCS(pEnt, pName, 11) == -1)
        {
            fFound=1;
            *pLBARet = i-1;             /* tell em what LBA of DirEnt */
            *poEntRet = pEnt-pStart;    /* Tell em offset in LBA */
            break;
        }
        --j;        /* one less dir ent */
        pEnt+=32;   /* 32 byte per entry */
    }
 }
 if (fFound)
 {
    *pEntRet = pEnt;
    return(0);
 }
 else
    return (ErcNoSuchFile);
}
```

```
/*****************************************
  This builds a full file specification from
  pName and places it in pDest based on the
  path from iJob. cbDestRet is set to size.
*****************************************/

static void BuildSpec(char *pName,
              long cbName,
              char *pDest,
              long *cbDestRet,
              long iJob)
{
long i;

 if ((cbName) && (pName) && (pName[1] == ':'))
 {      /* Do NOT use path */
     CopyData(pName, pDest, cbName);
     i = cbName;
 }
 else
 {                                  /* Use path as prefix */
    i = 0;
    GetPath(iJob, pDest, &i);
    if ((cbName) && (pName))
    {
        CopyData(pName, &pDest[i], cbName);
        i += cbName;
    }
 }
 *cbDestRet = i;
}



/*************************************************
 The parses out the path name into directories, filename,
 and extension  (Example):
 C:\TEMP\TEST.TXT  (with TEMP being a dir in the root).
 This also uses the current path for the job to build
 the filename.  If the name starts with \ or "DRIVE:"
 then the path from the JCB is NOT used. Otherwise
 the pName is concatenated to the job's path.
 SpecDepth is set to the level of the last valid
 11 character string (0-6).
*************************************************/

static U32 ParseName(U8 *pName, U32 cbName, U32 iJob)
{
unsigned long i, j, k, erc;
U8 c, *pPart;
char Spec[70];
U32 cbSpec;

 erc = 0;
 FDrive = 0;

 FillData(FileSpec, (7*11), ' ');       /* Fill parse table with spaces */
```

continues

Listing 27.1. continued

```
BuildSpec(pName, cbName, Spec, &cbSpec, iJob);

j = 0;      /* index into crnt part of spec */
k = 0;      /* index into crnt tree level   */
pPart = Spec;
for (i=0; i < cbSpec; i++)
{
   switch (c = *pPart++)
   {
       case 0x5c  :    /* '\' separates dir or fname */
           if (j>0)
           {  /* if it's not the first one */
               ++k;
               j=0;
           }
           break;
       case ':' :
           if ((j==1) && (k==0) && (FDrive==0))
           {
               FDrive = FileSpec[0][0] & 0xdf;  /* Make drive Upper*/
               FileSpec[0][0] = ' ';
               j=0;            /* back to beginning of part */
               k=0;
           }
           else erc = ErcBadFileSpec;
           break;
       case '.'  :             /* . can only appear once in dir or fname */
           if (j>8) erc = ErcBadFileSpec;
           else j=8;                       /* move to extension */
           break;
       case '>'  :                         /* not allowed in spec */
       case '<'  :
       case ','  :
       case '+'  :
       case '|'  :
       case ']'  :
       case '['  :
       case '+'  :
       case '='  :
       case '@'  :
       case '*'  :
       case '?'  :
            erc = ErcBadFileSpec;
           break;
       default   :                         /* make chars upper */
           if (j>10)
               erc = ErcBadFileSpec;
           else
           {
               if (((c >= 'A') && (c <= 'Z')) ||
                   ((c >= 'a') && (c <= 'z')))
                       c &= 0xdf;
               FileSpec[k][j] = c;
               ++j;
           }
           break;
       }
```

```
    if (erc) break;     /* bad news. Exit for loop */
    }
SpecDepth = k;
return erc;
}


/*** Get Directory Sector ****************************
 Gets a 512 byte directory sector in sequence number from
 the directory path given. SectNum = 0 to nMax for Dir.
 This also returns the LBA for sector itself for
 internal users of this call.
*****************************************************/

static U32 GetDirSectorM(char *pPath,
                long cbPath,
                char *pSectRet,
                long cbRetMax,
                long SectNum,
                long *LBARet,
                U16  *ClstrRet,
                long iJob)
{
U32 sector, i, j, k, erc, spc, level, iSect;
U16 MaxClstr, Clstr, rClstr;
U8  fFound, *pEnt, Drive;

  if (cbRetMax > 512)   /* WHOA Bub, 1 Sector at a time! */
    cbRetMax = 512;

  erc = ParseName(pPath, cbPath, iJob);

    /* The entire path has now been parsed out into an array of
       arrays, each 11 bytes long that contain each directory name
       in the path for the sector they want.
       The first is always the root (entry 0).
       The drive will be a letter in FDrive.
    */

  if ((FDrive > 0x40) && (FDrive < 0x52))       /* A to J */
    Drive = FDrive - 0x41;                      /* Make it 0-9 */
  else
    return(ErcNoSuchDrive);

  if (Drive < 2)
  {
    StatFloppy(Drive);
    erc= read_BS(Drive);
  }

  if (Ldrv[Drive].DevNum == 0xff)
    return(ErcNoSuchDrive);

  i = Ldrv[Drive].LBARoot;
  j = Ldrv[Drive].nRootDirEnt;
```

continues

Listing 27.1. continued

```
  if (FileSpec[0][0] == ' ')
  {     /* They want sector in root */
    if (SectNum > j/32)             /* Beyond Root entries! */
        return(ErcNoMatch);

    /* Else we can give them the sector NOW */
    erc = DeviceOp(Ldrv[Drive].DevNum, 1, i+SectNum, 1, abRawSector);
    if (!erc)
    {
        *LBARet = i+SectNum;
        CopyData(abRawSector, pSectRet, cbRetMax);
    }
    return(erc);
  }

    /* We have to run the root for a dir name... */

  fFound = 0;
  while ((j) && (!fFound))
  { /* while there are valid entries */
    erc = DeviceOp(Ldrv[Drive].DevNum, 1, i++, 1, abRawSector);
    if (erc)
        return(erc);
    pEnt = abRawSector;     /* Point to first entry */
    for (k=0; k<16; k++)
    {
        if (CompareNCS(pEnt, FileSpec[0], 11) == -1)
        {
            fFound=1;
            break;
        }
        --j;        /* one less dir ent */
        pEnt+=32;   /* 32 byte per entry */
    }
  }
  if (!fFound)
    return (ErcNoMatch);

  pDirEnt = pEnt;       /* Entry we just found in root was dir */

  if (!(pDirEnt->Attr & DIRECTORY))
  {
    return(ErcNoSuchDir);
  }

  if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
  else
    MaxClstr = 0xff8;   /* FAT12 */

  spc = Ldrv[Drive].SecPerClstr;    /* How many sectors per cluster */
  Clstr = pDirEnt->StartClstr;

  level = 1;    /* start at this directory+1, compare to FileSpec */

  while (!erc)
  { /* looking for Dir */
```

```
if (FileSpec[level][0] == ' ')
{ /* They want sector in this dir */

    if (!(pDirEnt->Attr & DIRECTORY))
    {
        return(ErcNoSuchDir);
    }
    rClstr = SectNum /spc;  /* calc relative cluster from start clstr */
    iSect  = SectNum % spc;  /* Add this to cluster start for sector */
    sector = ClsToLBA(Clstr, Drive);    /* sector of first dir sector */
    while ((rClstr--) && (!erc))
        erc = NextFATClstr(Drive, Clstr, &Clstr);
    if (erc)
        return(erc);
    sector = ClsToLBA(Clstr, Drive);  /* LBA of this clstr */
    sector += iSect;
    erc = DeviceOp(Ldrv[Drive].DevNum, 1, sector, 1, abRawSector);
    if (!erc)
    {
        CopyData(abRawSector, pSectRet, cbRetMax);
        *LBARet = sector;
        *ClstrRet = Clstr;
    }
    return(erc);
}
else
{  /* Else we must find this sub dir name */

    sector = ClsToLBA(Clstr, Drive);    /* sector of first dir sector */
    fFound=0;
    i = 0;

    while (!fFound)
    {       /* while there are valid entries */

        if (i==spc)
        {       /* reached last dir sector of this cluster */
            erc = NextFATClstr(Drive, Clstr, &Clstr);
            if (!erc)
            {
                if (Clstr >= MaxClstr)          /* last sector */
                    return(ErcNoSuchFile);      /* not found */
                sector = ClsToLBA(Clstr, Drive); /* LBA of next sector */
                i=0;
            }
            else
                return(erc);
        }

        erc = DeviceOp(Ldrv[Drive].DevNum, 1, sector++, 1, abRawSector);
        if (erc)
            return(erc);
        i++;    /* Next sector in this cluster */

        pEnt = &abRawSector[0];
        for (k=0; k<16; k++)
        {       /* 16 entries per sector */
```

continues

Listing 27.1. continued

```
                if (CompareNCS(pEnt, FileSpec[level], 11) == -1)
                {
                    fFound=1;
                    break;
                }
                pEnt+=32;   /* 32 byte per entry */
            }
        }
        pDirEnt = pEnt;              /* Entry we just found */
        Clstr = pDirEnt->StartClstr;  /* Clstr @ start of dir entry */
    }
    ++level;        /* next level of parsed filespec */
  }
return (erc);
}

/*****************************************************
 This is the BLOCK read for the MMURTL DOS file system.
 It reads whole sectors and returns them to pBytesRet.
 There are NO internal filesystem buffers for this call.
 Data is returned directly to your buffer from the Disk.
 This is the fastest method for reading a file.
 This is also used internally to fill a stream buffer.
*****************************************************/

static U32 ReadBlockM(U32 dHandle,
              U8  *pBytesRet,
              U32 nBytes,
              U32 dLFA,
              U32 *pdBytesRet,
              U8  fFill)          /* TRUE if filling a stream buffer */
{
U32 erc, j, LBA, iFCB, bpc, spc, nDone, rLFA, nLeft, nBlks;
U16 Clstr, MaxClstr, ClstrSav;
U8 Drive;

  erc = ValidateHandle(dHandle, &iFCB);     /* Sets iFCB if OK */
  if (erc) return erc;

  /* Certain FUB fields have different meanings in stream */

  if ((paFUB[dHandle].fStream) && (!fFill))
  {
    *pdBytesRet = 0;
    return ErcStreamFile;
  }

  /* set these up in advance */

  nBlks = nBytes/512;                   /* nBytes MUST be multiple of 512 */
  Drive = paFCB[iFCB]->Ldrv;            /* What logical drive are we on? */
  spc = Ldrv[Drive].SecPerClstr;        /* sectors per cluster */
  bpc = 512 * spc;                      /* Bytes per cluster */
  if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
  else
    MaxClstr = 0xff8;   /* FAT12 */
```

```
/* Call to find the absolute cluster on the the logical disk,
   and also the relative LFA of the cluster in question.
*/

erc = GetAbsoluteClstr(dHandle, dLFA, &Clstr, &rLFA);

LBA = ClsToLBA(Clstr, Drive);     /* Get LBA of the target cluster */

/* Now LBA equals beginning of cluster that dLFA resides in.
   We must see which sector in Clstr is the starting LBA.  To do this
   we MOD (dLFA in sectors) by (sectors per cluster) and
   add the leftover amount (should be 0 to nSPC-1) to LBA before we
   read.  For example: if dLFA was 2560 and spc was 4, this
   would be 5 % 4 = 1.  We would add 1 to the LBA.
   This is only done for the first read in the loop.

   We also set nleft which is how many sectors are left in the
   current cluster we are reading from.
*/

LBA += (dLFA/512) % spc;
nLeft = spc - ((dLFA/512) % spc);
nDone = 0;

while ((nBlks) && (!erc))
{ /* while buffer isn't full and no error */
  if (nBlks > nLeft)
      j = nLeft;
  else j = nBlks;

  paFUB[dHandle]->Clstr = Clstr;          /* Save Current cluster */
  paFUB[dHandle]->LFAClstr = rLFA;        /* Save LFA for Clstr in FUB */

  erc = DeviceOp(Ldrv[Drive].DevNum, 1, LBA, j, pBytesRet);
  if (erc)
      break;
  pBytesRet += j * 512;   /* further into their buffer */
  nBlks -= j;
  nLeft -= j;
  nDone += j;

  if ((nBlks) && (!nLeft))
  {       /* current cluster has none left */
      nLeft = spc;
      ClstrSav = Clstr;
      erc = NextFATClstr(Drive, Clstr, &Clstr);   /* next FAT cluster */
      if (erc)
      {
          *pdBytesRet = nDone*512;
          return(erc);
      }
      rLFA += bpc;                          /* Update rel LFA of new cluster*/
      if (Clstr >= MaxClstr)
          erc = ErcEOF; /* Last cluster */
      if (!Clstr)
          erc = ErcBrokenFile;     /* No good next cluster! */
```

continues

Listing 27.1. continued

```
      LBA = ClsToLBA(Clstr, Drive);        /* Get LBA of the target cluster*/
    }
  }
  *pdBytesRet = nDone*512;

  return erc;       /* WE'RE DONE, return the error (if any) */
}


/*** Write Block ************************************
 This is the BLOCK write for the MMURTL FAT file system.
 dLFA must be the LFA of a valid portion of the file and
 nBlks must not extend beyond the last cluster allocated
 to the file. in other words, you must call SetFileSize first.
 In a block write dLFA is always written on
 a sector boundry. We must make sure that the filesize
 will accomodate the sectors we want to write!
*****************************************************/

static U32 WriteBlockM(U32 dHandle, char *pData, U32 nBytes,
               U32 dLFA, U32 *pdnBytesRet)
{
U32 erc, i, j, LBA, iFCB, bpc, spc, nDone, rLFA, nLeft, nBlks;
U16 Clstr, MaxClstr;
U8 Drive;

  erc = ValidateHandle(dHandle, &iFCB);     /* Sets iFCB if OK */
  if (erc) return erc;

  nBlks = nBytes/512;
  dLFA = (dLFA/512)*512;        /* round LFA down to nearest sector */

  if (!paFCB[iFCB]->Mode)       /* Is it open in Modify?? */
    return(ErcReadOnly);

  i = (paFCB[iFCB]->FileSize/512);  /* Set i nBlks in file max */
  if (paFCB[iFCB]->FileSize%512)
    i++;

  j = (dLFA/512) + nBlks;   /* blocks to write past dLFA*/

  if (j > i)
    return(ErcBeyondEOF);

  /* It seems OK to write the blocks out, so now let's DO IT! */


  Drive = paFCB[iFCB]->Ldrv;            /* What logical drive are we on? */
  spc = Ldrv[Drive].SecPerClstr;        /* sectors per cluster */
  bpc = 512 * spc;                      /* Bytes per cluster */
  if (Ldrv[Drive].fFAT16)
    MaxClstr = 0xfff8;
  else
    MaxClstr = 0xff8;   /* FAT12 */

  erc = GetAbsoluteClstr(dHandle, dLFA, &Clstr, &rLFA);
  if (erc)
```

```
    return(erc);

  LBA = ClsToLBA(Clstr, Drive);     /* Get LBA of the target cluster */

  /* Now LBA equals beginning of cluster that dLFA resides in.
     We must see which sector in Clstr is the starting LBA.  To do this
     we MOD (dLFA in sectors) by (sectors per cluster) and
     add the leftover amount (should be 0 to nSPC-1) to LBA before we
     write.  For example: if dLFA was 2560 and spc was 4, this
     would be 5 % 4 = 1.  We would add 1 to the LBA.
     This is only done for the first write in the loop.
     We also set nleft which is how many sectors are left in the
     current cluster we are writing to.
  */

  LBA += (dLFA/512) % spc;
  nLeft = spc - (dLFA/512) % spc;
  nDone = 0;

  while ((nBlks) && (!erc))
  { /* while blocks are left to write */
    if (nBlks > nLeft)
        j = nLeft;
    else j = nBlks;

    paFUB[dHandle]->Clstr = Clstr;          /* Save Current cluster */
    paFUB[dHandle]->LFAClstr = rLFA;        /* Save LFA for Clstr in FUB */

    erc = DeviceOp(Ldrv[Drive].DevNum, 2, LBA, j, pData);
    if (erc)
        break;
    pData += (j * 512); /* Update address */
    nDone += j;         /* Total blocks done so far */
    nBlks -= j;
    nLeft -= j;

    if ((nBlks) && (!nLeft))
    {       /* done with current cluster */
        nLeft = spc;
        erc = NextFATClstr(Drive, Clstr, &Clstr);   /* next FAT cluster */
        if (erc)
            return(erc);
        rLFA += bpc;                        /* Update relative LFA of new
cluster*/
        if ((Clstr >= MaxClstr) && (nBlks))  /* Problem! */
            erc = ErcBeyondEOF;             /* Last cluster & they want more*/
        if (!Clstr) erc = ErcBrokenFile;    /* No good next cluster! */
        LBA = ClsToLBA(Clstr, Drive);       /* Get LBA of the target cluster*/
    }
  }
  *pdnBytesRet = nDone * 512;
  return erc;       /* WE'RE DONE, return the error (if any) */
}


/*******************************************************
 Fills the stream buffer for the Current LFA in the FUB.
 We simply figure out which relative LBA we want in the
```

continues

Listing 27.1. continued

```
 buffer and call ReadBlockM to fill it.  We then set
 LFABuf in the FUB to show the LFA of the first byte
 in the buffer.
*********************************************************/

static U32 FillStreamBuff(U32 dHandle, U8 fInitial)
{
U32 erc, i, LFA, cLFA, LFABuf, iFCB;
U32 sBuf;
U8  *pBuf;

  erc = 0;

  /* Set these up in advance */

  cLFA =  paFUB[dHandle]->CrntLFA;      /* Find out where we want to be */
  LFABuf = paFUB[dHandle]->LFABuf;      /* LFA of first byte in buffer now */
  pBuf = paFUB[dHandle]->pBuf;          /* Local ptr to buffer */
  sBuf = paFUB[dHandle]->sBuf;          /* size of buffer */
  iFCB = paFUB[dHandle]->iFCB;          /* FCB for this FUB */

    /*  If the file was just opened we fill with LFA 0 */

  if (fInitial)
  {
    erc = ReadBlockM(dHandle, pBuf, sBuf, 0, &i, TRUE);
  }

    /* Else If the LFA is already in the buffer we just exit OK */

  else if ((cLFA >= LFABuf) && (cLFA < (LFABuf + sBuf)))
  {
    erc = 0;
  }

    /* ELSE We must figure out what starting LFA we want and fill the buffer */

  else
  {
    LFA = (cLFA/512) * 512;     /* Round down to nearest sector */
    erc = ReadBlockM(dHandle, pBuf, sBuf, LFA, &i, 1);
    paFUB[dHandle]->LFABuf = LFA;
  }

  if (erc == ErcEOF)        /* We ignore this when filling the buffer */
    erc = 0;
  return erc;       /* WE'RE DONE, return the error (if any, except EOF) */
}

/*********************************************************
 This is the STREAM read for the MMURTL FAT file system.
 Used in conjunction with Get & Set FileLFA, you can
 move to any portion of the file to read one or more bytes.
 Data is buffered in Page sized chunks by the file
 system. This is the easiest method for reading a file.
*********************************************************/
```

```
static U32 ReadBytesM(U32 dHandle, U8 *pBytesRet, U32 nBytes, U32 *pdReadRet)
{
U32 erc, iFCB, sBuf, cLFA, fSize;
U32 nBytesDone,     /* Total read so far */
    lfaEOB,         /* LFA end of Buffer */
    nBytesOut;      /* Number of bytes to copy (this buffer full) */
U8  *pOut, *pBuf;   /* Next data to send caller from buffer. Local pbuf */

  erc = ValidateHandle(dHandle, &iFCB);     /* Sets iFCB if OK */
  if (erc) return erc;

  /* Certain FUB fields have different meanings in stream type file */

  if (!paFUB[dHandle]->fStream)
    return ErcBlockFile;

  cLFA = paFUB[dHandle]->CrntLFA;           /* local cLFA */
  fSize = paFCB[iFCB]->FileSize;            /* local size */

  /* check and see if we are at EOF */

  if (cLFA >= fSize)
  {                     /* early out */
    *pdReadRet = 0;
    return ErcEOF;
  }

  /* We are not at EOF so we need to fill stream buff
     and then calculate how many bytes we can give them
     from this buffer full of data.
  */

  pBuf = paFUB[dHandle]->pBuf;          /* Local ptr to buffer/size */
  sBuf = paFUB[dHandle]->sBuf;
  nBytesDone = 0;

  while ((nBytesDone < nBytes) &&
         (cLFA < fSize) &&
         (!erc))
  {
    erc = FillStreamBuff(dHandle, 0);               /* Fill the buff */

    /* Find out the LFA at the end of the buffer since we
       just filled it.  It may be less than the end since
       we may be near EOF.
    */

    lfaEOB = paFUB[dHandle]->LFABuf + sBuf -1;  /* LFA at End of Buffer */
    if (lfaEOB > fSize)                         /* Beyond EOF? */
        lfaEOB = fSize - 1;

    /* Calculate pointer to the next chunk out from stream buffer */

    pOut = pBuf + (cLFA - paFUB[dHandle]->LFABuf);

    /* Calc how many bytes we can get out of buffer (belongs to file) */
```

continues

Listing 27.1. continued

```
    nBytesOut = lfaEOB - cLFA + 1;
    if (nBytesOut > nBytes-nBytesDone)
        nBytesOut = nBytes-nBytesDone;

    /* Send bytes to pBytesRet */

    CopyData(pOut, pBytesRet, nBytesOut);

    pBytesRet += nBytesOut;          /* Update pBytesRet */
    nBytesDone += nBytesOut;         /* Update nBytesDone */
    cLFA += nBytesOut;               /* update CrntLFA */
    paFUB[dHandle]->CrntLFA = cLFA;
  }

  *pdReadRet = nBytesDone;           /* Tell em how many bytes they got */
  if (!erc)
    if (cLFA == fSize)
        erc = 1;

  return erc;
}

/*********************************************************
 Flushes the stream buffer if it has been modified.
 We read the current LFA of the buffer and write it
 to disk. Then reset the flag in the FCB.
 This uses WriteBlockM.  We ensure we do not call
 WriteBlockM with more sectors than are allocated
 to the file cause the buffer may extend past the
 actual allocated amount of clusters!
*********************************************************/

static U32 FlushStreamBuff(U32 dHandle)
{
U32 erc, i, j, LFABuf, iFCB, size;
U32 sBuf;
U8  *pBuf;

  erc = 0;

  if (paFUB[dHandle]->fModified)
  {

      LFABuf = paFUB[dHandle]->LFABuf;      /* LFA of first byte in buf now */
      pBuf = paFUB[dHandle]->pBuf;          /* Local ptr to buffer */
      sBuf = paFUB[dHandle]->sBuf;          /* size of buffer */
      iFCB = paFUB[dHandle]->iFCB;          /* to check filesize */
      size = paFCB[iFCB]->FileSize;

      i = (sBuf/512);             /* Total blocks in buff */
      j = (size-LFABuf)/512;      /* Blocks in buf belonging to file */
      if ((size-LFABuf)%512)      /* Odd bytes, add one more block */
        j++;
      if (j < i)
        i = j;
```

```
      erc = WriteBlockM(dHandle, pBuf, i*512, LFABuf, &i);
      paFUB[dHandle]->fModified = 0;
    }
    return erc;
}


/*** Write Bytes (Stream) ****************************
 This is the STREAM write for the MMURTL FAT file system.
 This uses FillStreamBuff to set up the buffer, and
 FlushStreamBuff to write modified stream buffers.
 This also calls SetFileSizeM to extend the
 filelength when necessary (writing at EOF).
*****************************************************/

static U32 WriteBytesM(U32 dHandle, char *pData, U32 nBytes, U32 *nBytesRet)
{
U32 erc, iFCB, sBuf, cLFA, fSize, OrigSize;
U32 nBytesDone,     /* Total written so far */
    lfaEOB,         /* LFA end of Buffer */
    nBytesOut;      /* Number of bytes to copy (this buffer full) */
U8  *pOut, *pBuf;   /* Next data to send caller from buffer. Local pbuf */

  erc = ValidateHandle(dHandle, &iFCB);     /* Sets iFCB if OK */
  if (erc) return erc;

  /* Certain FUB fields have different meanings in stream type file */

  if (!paFUB[dHandle]->fStream)
    return(ErcBlockFile);

  if (!paFCB[iFCB]->Mode)       /* Is it open in Modify?? */
    return(ErcReadOnly);

  cLFA = paFUB[dHandle]->CrntLFA;           /* local cLFA */
  fSize = paFCB[iFCB]->FileSize;            /* local size */

  /* check and see if we are at EOF or will go past it
  when we write */

  if (cLFA + nBytes > fSize)
  {     /* Must set file size first */
        OrigSize = fSize;
        erc = SetFileSizeM(dHandle, cLFA + nBytes);
        if (erc)
            return(erc);
        fSize = paFCB[iFCB]->FileSize;          /* local size */
  }

  /* Now we loop writing the bytes to the stream buffer
     filling it with each new section of the file.
     As this occurs we must call fillStreamBuff with
     each new section of the file if not already in the
     buff to ensure proper continuity of the file in case
     we are overwriting existing sections of the file.
  */
```

continues

Listing 27.1. continued

```
  pBuf = paFUB[dHandle]->pBuf;          /* Local ptr to buffer/size */
  sBuf = paFUB[dHandle]->sBuf;
  nBytesDone = 0;

  while ((nBytesDone < nBytes) &&
        (cLFA < fSize) &&
        (!erc))
  {
    /* First call FlushStreamBuff which will write the current
    buffer and reset the fModified flag (if needed).
    */
    erc = FlushStreamBuff(dHandle);
    if (erc)
        return(erc);

    if (cLFA < OrigSize)
        erc = FillStreamBuff(dHandle, 0);          /* Fill the buff */
    else
        paFUB[dHandle]->LFABuf= (cLFA/512) * 512;   /* Just set Buff LFA */

    /* Find out the LFA at the end of the buffer since we
       just filled it.  It may be less than the end since
       we may be near EOF.
    */

    lfaEOB = paFUB[dHandle]->LFABuf + sBuf -1;  /* LFA at End of Buffer */
    if (lfaEOB > fSize)                         /* Beyond EOF? */
        lfaEOB = fSize - 1;

    /* Calc pointer to where next chunk goes in stream buffer */

    pOut = pBuf + (cLFA - paFUB[dHandle]->LFABuf);

    /* Calc how many bytes we can write to buffer */

    nBytesOut = lfaEOB - cLFA + 1;
    if (nBytesOut > nBytes-nBytesDone)
        nBytesOut = nBytes-nBytesDone;

    /* Get bytes from pData into stream buffer */

    CopyData(pData, pOut, nBytesOut);
    paFUB[dHandle]->fModified = 1;

    pData += nBytesOut;             /* Update pData pointer */
    nBytesDone += nBytesOut;        /* Update nBytesDone */
    cLFA += nBytesOut;              /* update CrntLFA */
    paFUB[dHandle]->CrntLFA = cLFA;
  }

  *nBytesRet = nBytesDone;          /* Tell em how many bytes we wrote */

  return erc;

}
```

```
/*****************************************************
  GetFileSize returns the FileSize entry from the FCB
  for the handle specified.  This means the file must be
  OPEN.  This call would NOT be used for a directory
  listing function as you would have to open each file
  to get the information.  Use ReadDirSector instead.
*****************************************************/

static U32 GetFileSizeM(U32 dHandle, U32 *pdSizeRet)
{
U32 erc, iFCB;
  erc = ValidateHandle(dHandle, &iFCB);
  if (erc) return erc;
  *pdSizeRet = paFCB[iFCB]->FileSize;
  return 0;
}


/*****************************************************
  SetFileLFA sets a Stream mode file pointer to dLFA.
  If attempted on a block mode file an error is returned.
  -1 (0xffffffff) is equal to EOF.  The Stream Buffer
  is filled with the sectors that contains the LFA.
*****************************************************/

static U32 SetFileLFAM(U32 dHandle, S32 dLFA)
{
U32 erc, iFCB;

  erc = ValidateHandle(dHandle, &iFCB);
  if (erc) return erc;

  if (!paFUB[dHandle]->fStream)
    return ErcBlockFile;

  if (paFCB[iFCB]->Mode)    /* Modify mode - Flush will flush if needed */
       erc = FlushStreamBuff(dHandle);

  /* -1 = Set file ptr to EOF */

  if (dLFA == -1)
    dLFA = paFCB[iFCB]->FileSize;

  if (dLFA > paFCB[iFCB]->FileSize)
    erc = ErcBeyondEOF;

  if (!erc)
  {
      paFUB[dHandle]->CrntLFA = dLFA;   /* Find out where we are in the file */
      erc = FillStreamBuff(dHandle, 0);
  }


  return erc;
}
```

continues

Listing 27.1. continued

```
/*******************************************************
  GetFileLFA gets a Stream mode file pointer for caller
  returning it to pdLFARet.
  If attempted on a block mode file an error is returned.
  EOF is returned as the FileSize.
*******************************************************/

static U32 GetFileLFAM(U32 dHandle, U32 *pdLFARet)
{
U32 erc, iFCB;
  erc = ValidateHandle(dHandle, &iFCB);
  if (erc) return erc;
  if (!paFUB[dHandle]->fStream)
    return ErcBlockFile;
  *pdLFARet = paFUB[dHandle]->CrntLFA;

  return erc;
}


/*******************************************************
 This calls parse to validate the filename and separate
 it into directories and a filename.  It then walks
 up the tree until it either finds and opens the file,
 or errors out.  The handle that is returned is
 4 higher than the index to the FUB.  This is because
 0, 1, 2 & 3 are reserved for NUL, KBD, VID, and LPT
 which are  only accessible from the blocking File calls.
*******************************************************/

static U32 OpenFileM(U8 *pName,
              U32 cbName,
              U8 Mode,
              U8 fStream,
              U32 *pdHandleRet,
              U32 iJob)         /* make use of iJob later!!! */
{
U32 erc, level, i, iFCB, iFUB, LBADirEnt, EntOffset;
U16 Clstr;
U8 fFound, *pMem, Drive;

  if (Mode > 1) return ErcBadOpenMode;

  level = 0;    /* start at the root, compare to SpecDepth */

  erc = ParseName(pName, cbName, iJob);

    /* The entire path has now been parsed out into an array of
       arrays, each 11 bytes long that contain each directory name
       up to and including the filename.  The first is always
       the root (entry 0). The drive will be a letter in FDrive.
    */

  if ((FDrive > 0x40) && (FDrive < 0x52))   /* A to J */
    Drive = FDrive - 0x41;                          /* Make it 0-9 */
  else erc = ErcNoSuchDrive;
```

```
if (Ldrv[Drive].DevNum == 0xff)
  erc = ErcNoSuchDrive;

if (Drive < 2)
{
  StatFloppy(Drive);
  erc= read_BS(Drive);
}

if (!erc)
{                                   /* Get Root dir entry */
  erc = GetRootEnt(FileSpec[level],
                   Drive,
                   &LBADirEnt,
                   &EntOffset,
                   &pDirEnt);

  if (erc == ErcNoSuchFile)
  {
      if (level == SpecDepth) erc = ErcNoSuchFile;
      else erc = ErcNoSuchDir;
  }
  if (erc)
      return(erc);

  if (!erc)
      Clstr = pDirEnt->StartClstr;  /* Clstr = beginning of file or dir */

  while ((level < SpecDepth) && (!erc))
  {   /* looking for Dir, not file yet */

      ++level;                      /* next level of parsed filespec */

      erc = GetDirEnt(FileSpec[level],
                      Drive,
                      Clstr,
                      &LBADirEnt,
                      &EntOffset,
                      &pDirEnt);
      if (erc == ErcNoSuchFile)
      {
          if (level == SpecDepth) erc = ErcNoSuchFile;
          else erc = ErcNoSuchDir;
      }
      else if (erc)
          return(erc);
      else
          Clstr = pDirEnt->StartClstr;  /* Clstr @ start of dir entry */
  }

  /* if we got here with no error we've got a file or a DIR.
     If it's DIR then it's an error.
     pDirEnt points to its directory entry, and Clstr
     is the starting cluster of the file
  */

  if (!erc)
```

continues

Listing 27.1. continued

```
{
    /* If Attributes say it's not a file then ERROR */

    if (pDirEnt->Attr & (VOLNAME ¦ DIRECTORY))
        return ErcNotAFile;

    /* If ModeModify and File is readOnly then ERROR */

    if ((Mode) && (pDirEnt->Attr & READONLY))
        return ErcReadOnly;

/* We check to see if it's already open by looking through the
   valid FCBs to see if we have a Drive, StartClstr& name match.
   If so, we must see if the modes are compatible.
   A valid FCB is one where nUsers > 0.
*/

fFound = 0;
i=0;
while ((i<nFCBs) && (!fFound))
{

    if ((paFCB[i]->nUsers) &&
        (paFCB[i]->Ldrv == Drive) &&
        (paFCB[i]->StartClstr == pDirEnt->StartClstr) &&
        (CompareNCS(&paFCB[i],
                    FileSpec[SpecDepth], 11) == 0xffffffff))
         fFound = 1;
    else
        ++i;
}

if (fFound)
{           /* it's open already.  i is index into FCBs  */
    if (paFCB[i]->Mode)     /* it's open in Modify already */
        return ErcFileInUse;
    else
    {
        iFCB = i;           /* Index to this FCB */
        pFCB = &paFCB[i];   /* make pFCB point to FCB found */
        pFCB->nUsers++;     /* One more user */
    }
}
else
{                   /* It not already open. Find empty FCB */
    i = 0;
    while ((i<nFCBs) && (paFCB[i]->nUsers)) ++i;    /* Find new FCB */

    if (i==nFCBs) return ErcNoFreeFCB;          /* Couldn't */

    /* i now indexes into FCBs for a free FCB.  New we copy the
       directory entry for the file into the FCB and set up
       the other FCB values. */

    iFCB = i;                       /* used to add an FUB */
    pFCB = &paFCB[i];               /* make pFCB point to FCB found */
    CopyData(pDirEnt, pFCB, 32);    /* Copy Dir Ent into FCB */
```

```
    pFCB->Ldrv = Drive;             /* Set Drive */
    pFCB->nUsers++;                 /* Now in use */
    pFCB->Mode = Mode;              /* Open mode */
    pFCB->LBADirSect =  LBADirEnt;  /* So we know where it came from */
    pFCB->oSectDirEnt = EntOffset;  /* "  "  */
}

/* Now we have an FCB (either existing or we just built it).
   Now add an FUB and fill in the info for the user
   so we can return a handle to it.  The Job fields is 0
   for free FUBs.
*/

i = 4;
while ((i<nFUBs) && (paFUB[i]->Job)) ++i;   /* Find new FUB */
if (i==nFUBs)
{
    pFCB->nUsers--;                 /* Make FCB correct */
    return ErcNoFreeFUB;            /* Couldn't */

}

/* If we got here, i is an index to a free FUB. */

iFUB = i;
paFUB[iFUB]->Job = iJob;        /* Job Owner */
paFUB[iFUB]->iFCB = iFCB;       /* Set index to FCB for this file */
paFUB[iFUB]->CrntLFA = 0;       /* Current Logical File Address */
paFUB[iFUB]->fModified = 0;     /* Stream buf was modified */
paFUB[iFUB]->fStream = fStream; /* NonZero for STREAM mode */
paFUB[iFUB]->Clstr = pDirEnt->StartClstr;   /* Start Cluster */
paFUB[iFUB]->LFAClstr = 0;      /* Rel LFA to 0 */
paFUB[iFUB]->LFABuf = 0;        /* First LFA in Buffer */
paFUB[iFUB]->sBuf = 0;          /* Default to No Buf */

if (fStream)
{       /* allocate/fill buffer and set rest of FUB */

    erc = AllocOSPage(1, &pMem);        /* Stream Buf is 4K */

    if (erc)
    {                           /* No MEM left... Bummer */
        pFCB->nUsers--;                 /* Return FCB to pool */
        paFUB[iFUB]->Job = 0;           /* Return FUB to pool */
        return erc;                     /* Return Erc to user... */
    }

    paFUB[iFUB]->pBuf = pMem;       /* Ptr to buffer if stream mode */
    paFUB[iFUB]->sBuf = 4096;       /* Size of buffer */

    erc = FillStreamBuff(iFUB, 1);  /* fInitial to TRUE */

    if (erc)
    {
        pFCB->nUsers--;             /* Return FCB to pool */
        paFUB[iFUB]->Job = 0;       /* Return FUB to pool */
        DeAllocPage(pMem, 1);       /* Free memory for buffer */
```

continues

Listing 27.1. continued

```
            return erc;                 /* Return Erc to user... */
        }
    }

    *pdHandleRet = iFUB;        /* File handle */
    }
  }

return erc;

}


/*******************************************************
  CLOSE FILE for the MMURTL DOS file system. This finds
  the FUB, checks to see if the buffer should be flushed
  and deallocated (only for stream mode), invalidates
  the FUB, and the FCB (if this is the last or only user).
*******************************************************/

static U32 CloseFileM (U32 dHandle)
{
U32 erc, iFCB, i;

    erc = ValidateHandle(dHandle, &iFCB);
    if (erc) return erc;

    if (paFCB[iFCB]->Mode)
    {  /* Modify mode */
        if (paFUB[dHandle]->fStream)
        {
            erc = FlushStreamBuff(dHandle);
        }
        UpdateDirEnt(iFCB);         /* ignore error */
    }

    if (paFUB[dHandle]->fStream)
        DeAllocPage(paFUB[dHandle]->pBuf, 1);   /* Free buffer */

    /* This means the FS is screwed up. This shouldn't happen... */
    if (!paFCB[iFCB]->nUsers)
        erc = ErcBadFCB;
    else
        paFCB[iFCB]->nUsers--;

      /* Now we should be able to close it and free the FUB.
         If the FCB.nUsers flips to 0 it will be free too
      */

    paFUB[dHandle]->Job = 0;

    /* This will write all modified fat sectors */

    for (i=0; i<nFATBufs; i++)
            UpdateFAT(i);

  return erc;
}
```

```
/*** Create File ************************************
 This is Create File for the MMURTL FAT file system.
 This is also used internally to create directories.
****************************************************/

static U32 CreateFileM(char *pName,
                long cbName,
                long attrib,
                long iJob)
{
unsigned long dHandle, i, j, k, erc, LBA, spc;
char Path[70];
long cbPath;
char filename[12];
U16 CrntClstr, ClstrValue, iStart, DirClstr;
U8 fFound, Drive, fDir;

    /* First we try to open it to see if it exists. If we get back
        ErcOK or ErcFileInUse the name is already
        in use as a file and we give them ErcDupName.
        We return other errors as we find them.
    */
    if (attrib & DIRECTORY)
    {
        fDir = 1;
        erc = 0;
    }
    else
        fDir = 0;

    erc = OpenFileM(pName, cbName, 0, 0, &dHandle, iJob);

    switch (erc)
    {
    case ErcOK:
        CloseFileM(dHandle);
        erc = ErcDupName;
        break;
    case ErcFileInUse:
        erc = ErcDupName;
        break;
    case ErcNotAFile:   /* It a directory... */
        break;
    case ErcNoSuchFile:
    {   /* OK, this means we can try to create it! */

        erc = 0;

        BuildSpec(pName, cbName, Path, &cbPath, iJob);

        erc = ParseName(Path, cbPath, iJob);
        if (erc)
            return(erc);

        /* FDrive was set up on Parse */
        Drive = FDrive - 0x41;          /* Make it 0-9 */
```

continues

Listing 27.1. continued

```
/* First we setup the filename from what was parsed out
during the Parse call. Then eliminate it from Path.
*/

CopyData(FileSpec[SpecDepth], filename, 11); /* filename */

filename[11] = 0;

/* Hack the filename from Path so we can search the
directory path properly */

while ((cbPath) && (Path[cbPath-1] != 0x5C))
{
    cbPath--;
}

/* Each directory sector has 16 32 byte entries.
We will now walk thru each sector until we find one
that is a deleted or empty entry.
A deleted entry has E5h as its first character in the name.
An unused entry has 00h as its first character in the name.
*/

fFound = 0;
i = 0;                              /* i = sectornum */
while ((!fFound) && (!erc))
{
    erc = GetDirSectorM(Path, cbPath, abTmpSector,
                        512, i++, &LBA, &DirClstr, iJob);
    if (!erc)
    {
        k = 0;
        pDirEnt = abTmpSector;
        while (k<16)
        {
            if ((pDirEnt->Name[0] == 0xE5) ||
                (!pDirEnt->Name[0]))
                {
                    fFound = 1;
                    break;
                }
            pDirEnt += 32;
            k++;
        }
    }
}
/* When we get here, we have either found an entry or
we have run out of sectors!  If we run out of sectors
and this is the root, we error out (RootFull), otherwise
we extend the directory.
*/

if ((erc == ErcNoMatch) && (!SpecDepth))
    return (ErcRootFull);

else if (erc == ErcEOF)
{       /* reach end of dir! */
```

```
    /* We must now extend the cluster chain for this
       directory entry. DirClstr holds the last
       valid sector of the directory.
    */

  FillData(abTmpSector, 512, 0);
  spc = Ldrv[Drive].SecPerClstr;        /* sectors per cluster */
  erc = ExtendClstrChain(Drive, DirClstr, &DirClstr);
  if (erc)
    return(erc);
  LBA = ClsToLBA(DirClstr, Drive);
  j = LBA;
  i = spc;
  erc = 0;
  while ((i--) && (!erc))
  {
    erc = DeviceOp(Ldrv[Drive].DevNum, 2, j++,
                   1, abTmpSector);
  }
  pDirEnt = abTmpSector;  /* first entry in new sector */
  fFound = 1;

    /* We now have a new cluster on the end of the
    directory and it is all zeros!. The first sector
    is pointed to by LBA and abTmpSector is still zeros
    just as an new dir sector should be with pDirEnt
    pointing to the first entry.
    */
}

if ((!erc) && (fFound))
{       /* Let's DO IT! */

    /* pDirEnt points to the entry we will use and
       abTmpSector still has the entire sector in it,
       so we find an empty clstr on the disk, allocate
       to this file, fill in the rest of the dir
       entry and we are almost done.
    */

    /* Find a fat buf already in memory for this drive */
    /* One WILL be here! */

    k = 0;
    CrntClstr = 0;
    while ((k<nFATBufs) && (!CrntClstr))
    {
        if ((Drive == Fat[k].Drive) && (Fat[k].LastUsed))
            CrntClstr = Fat[k].iClstrStart; /* valid cluster */
        k++;
    }

    if (!CrntClstr)
        CrntClstr = 2;      /* Can't find it so start at beginning */

    iStart = CrntClstr;     /* where we started looking for empties */
```

continues

Listing 27.1. continued

```
fFound = 0;
while (!fFound)
{
    ++CrntClstr;        /* next cluster */
    if (CrntClstr == iStart)
        return(ErcDiskFull);

    erc = GetClstrValue(CrntClstr, Drive, 0, &ClstrValue, &j);

    if ((!erc) && (!ClstrValue))
            fFound = 1;     /* found an empty one */

    else if (erc == ErcBadFATClstr)
    { /* off the end */
            /* we started AFTER beginning of disk so
              we will go back and look for empties
              from beginning to where we started.
            */

        if (iStart > 2)
            CrntClstr = 2;
        else
            return(ErcDiskFull);
    }
    else if (erc)
            return(erc);
}

/* If we got here, we found an empty cluster */

CopyData(filename, pDirEnt, 11);
if (!fDir)
    pDirEnt->Attr =
        attrib & (READONLY ¦ HIDDEN ¦ SYSTEM ¦ ARCHIVE);
else
    pDirEnt->Attr = attrib;
GetFATTime(&pDirEnt->Time, &pDirEnt->Date);
pDirEnt->StartClstr = CrntClstr;
pDirEnt->FileSize = 0;
erc = SetClstrValue(CrntClstr, 0xFFFF, Drive, &i);
/* Now we write the dir sector back to disk */
if (!erc)
    erc = DeviceOp(Ldrv[Drive].DevNum, 2, LBA,
                   1, abTmpSector);

/* If we were creating a directory, we must add
the two default directory entries . and ..
This is done by filling out abTmpSector as
the first sector of an empty directory and writing
it out to the allocated cluster.
We then zero it out and write it to the rest
of the sectors in the new cluster.
*/

if (fDir)
{
  FillData(abTmpSector, 512, 0);
```

```
            pDirEnt = abTmpSector;  /* first entry in new sector */

              /* do the current dir entry (.) */

            CopyData(".          ", pDirEnt, 11);
            pDirEnt->Attr = DIRECTORY;
            GetFATTime(&pDirEnt->Time, &pDirEnt->Date);
            pDirEnt->StartClstr = CrntClstr;
            pDirEnt->FileSize = 0;

              /* do the previous current dir entry (.) */

            pDirEnt += 32;
            CopyData("..         ", pDirEnt, 11);
            pDirEnt->Attr = DIRECTORY;
            GetFATTime(&pDirEnt->Time, &pDirEnt->Date);
            pDirEnt->StartClstr = DirClstr;
            pDirEnt->FileSize = 0;

            spc = Ldrv[Drive].SecPerClstr;        /* sectors per cluster */
            LBA = ClsToLBA(CrntClstr, Drive);
            j = LBA;

            /* Write this sector out to disk */
            erc = DeviceOp(Ldrv[Drive].DevNum, 2, j++,
                           1, abTmpSector);

            FillData(abTmpSector, 512, 0); /* zero the rest */

            erc = 0;
            i = spc-1;    /* less one cause we wrote the first */
            while ((i--) && (!erc))
            {
              erc = DeviceOp(Ldrv[Drive].DevNum, 2, j++,
                             1, abTmpSector);
            }

          }
          /* This will write all modified fat sectors */

          for (i=0; i<nFATBufs; i++)
              UpdateFAT(i);

          return(erc);

        }
        else
            return(erc);
    }
    default: ;

    } /* switch */
    return (erc);
}

/*** Delete File ***************************************
 This is Delete File for the MMURTL FAT file system.
```

continues

Listing 27.1. continued

```
 The file must be opened in mode modify which gives
 the caller exclusive access. The file is closed
 even if the Delete fails.
*****************************************************/

static U32 DeleteFileM(long *dHandle)
{
U32 erc, iFCB, i;
U16 iStart;
U8 Drive;

    erc = ValidateHandle(dHandle, &iFCB);
    if (erc) return erc;

    Drive = paFCB[iFCB]->Ldrv;          /* What logical drive are we on? */

    if (!paFCB[iFCB]->Mode)
    {  /* Modify mode? */
        CloseFileM(dHandle);
        return ErcReadOnly;
    }
    if (paFUB[dHandle]->fStream)
        DeAllocPage(paFUB[dHandle]->pBuf, 1);   /* Free buffer */

    iStart = paFCB[iFCB]->StartClstr;
    if (iStart)
    {
        erc = TruncClstrChain(Drive, iStart);
        if (!erc)
            erc = SetClstrValue(iStart, 0, Drive, &i);
    }

    paFCB[iFCB]->Name[0] = 0xE5;
    UpdateDirEnt(iFCB);         /* ignore error */

    /* This means the FS is screwed up. This shouldn't happen... */
    if (!paFCB[iFCB]->nUsers)
        erc = ErcBadFCB;
    else
        paFCB[iFCB]->nUsers--;

      /* Now we should be able to close it and free the the FUB.
         If the FCB.nUsers flips to 0 it will be free too
      */

    paFUB[dHandle]->Job = 0;

    /* This writes all modified fat sectors */

    for (i=0; i<nFATBufs; i++)
            UpdateFAT(i);

    return erc;
}
```

```
/*** Rename File ************************************
 This is Rename File for the MMURTL FAT file system.
****************************************************/

static U32 RenameFileM(char *pCrntName, long dcbCrntName,
              char *pNewName, long dcbNewName, U32 iJob)
{
U32 dHandle, erc, erc1, iFCB;

    erc = OpenFileM(pCrntName, dcbCrntName, 1, 0, &dHandle, iJob);
    if (!erc)
    {
        FDrive1 = FDrive;
        CopyData(FileSpec, FileSpec1, 77);
        SpecDepth1 = SpecDepth;
        erc = ParseName(pNewName, dcbNewName, iJob);
        if (!erc)
            if ((FDrive1 != FDrive) || (SpecDepth1 != SpecDepth))
                erc = ErcRenameDrv;         /* No Rename across drives */
        if (!erc)
            if (SpecDepth)      /* Compare upper tree */
                if (CompareNCS(FileSpec, FileSpec1, SpecDepth * 11) != -1)
                    erc = ErcRenameDir;     /* No Rename across dirs */
        if (!erc)
        { /* OK to rename */
            iFCB = paFUB[dHandle]->iFCB;        /* FCB for this FUB */
            CopyData(FileSpec[SpecDepth], &paFCB[iFCB], 11);
            erc = UpdateDirEnt(iFCB);
        }
        erc1 = CloseFileM(dHandle);
        if (!erc)
            erc = erc1;
    }
    return (erc);
}

/*** Create Dir *************************************
 This is Create Directory for the MMURTL FAT file system.
****************************************************/

static U32  CreateDirM(char *pPath, long cbPath, long iJob)
{
long erc;

  erc = CreateFileM(pPath, cbPath, DIRECTORY, iJob);
  return(erc);

}
/*** Delete Directory *******************************
 This is Delete Directory for the MMURTL FAT file system.
****************************************************/

static U32  DeleteDirM(char *pPath, long cbPath, long fAllFiles, long iJob)
{
    pPath = 0;
    cbPath = 0;
    fAllFiles = 0;
```

continues

Listing 27.1. continued

```
    iJob = 0;
}


/*******************************************************
 This is the File system task. All file system requests
 end up here to be serviced.
*******************************************************/

static void FSysTask(void)
{
U32 FMsg[2], merc, erc, i;
U16 i16;

while (1)
{
  erc = WaitMsg(FSysExch, FMsg);
  if (!erc)
  {

    pRQB = FMsg[0];     /* first DD in Msg is pointer to RQBlock */

    switch (pRQB->ServiceCode)
    {
        case 0 :        /* JobAbort - CLOSE ALL FILES FOR THIS JOB! */
            i = 4;
            while (i<nFUBs)
            {
                if (paFUB[i]->Job == pRQB->dData0)
                {
                    CloseFileM(i);
                }
                ++i;    /* next FUB */
            }

            erc = 0;
            break;
        case 1 :        /* OpenFile */
            erc = OpenFileM(pRQB->pData1,       /* pFilename */
                            pRQB->cbData1,      /* dcbFilename */
                            pRQB->dData0,       /* Mode */
                            pRQB->dData1,       /* Type */
                            pRQB->pData2,       /* pdHandleRet */
                            pRQB->RqOwnerJob);  /* iJob Owner */
            break;
        case 2 :        /* CloseFile */
            erc = CloseFileM(pRQB->dData0);     /* Handle */
            break;
        case 3 :        /* ReadBlock */
            erc = ReadBlockM(pRQB->dData0,       /* Handle */
                             pRQB->pData1,       /* pDataRet */
                             pRQB->cbData1,      /* nBytes */
                             pRQB->dData1,       /* dLFA */
                             pRQB->pData2,       /* pdnBytesRet */
                             0);                 /* NOT internal */
            break;
```

```
case 4 :        /* WriteBlock */
    erc = WriteBlockM(pRQB->dData0,      /* Handle */
                      pRQB->pData1,      /* pData */
                      pRQB->cbData1,     /* nBytes */
                      pRQB->dData1,      /* dLFA */
                      pRQB->pData2);     /* pdBytesRet */
    break;
case 5 :        /* ReadBytes */
    erc = ReadBytesM(pRQB->dData0,       /* Handle */
                     pRQB->pData1,       /* pDataRet */
                     pRQB->cbData1,      /* nBytes */
                     pRQB->pData2);      /* pdnBytesRet */
    break;
case 6 :        /* WriteBytes */
    erc = WriteBytesM(pRQB->dData0,      /* Handle */
                      pRQB->pData1,      /* pData */
                      pRQB->cbData1,     /* nBytes */
                      pRQB->pData2);     /* pdnBytesRet */
    break;
case 7 :        /* GetFileLFA */
    erc = GetFileLFAM(pRQB->dData0,      /* Handle */
                      pRQB->pData1);     /* pdLFARet */
    break;
case 8 :        /* SetFileLFA */
    erc = SetFileLFAM(pRQB->dData0,      /* Handle */
                     pRQB->dData1);     /* dNewLFA */
    break;
case 9 :        /* GetFileSize */
    erc = GetFileSizeM(pRQB->dData0,     /* Handle */
                      pRQB->pData1);    /* pdSizeRet */
    break;
case 10 :       /* SetFileSize */
    erc = SetFileSizeM(pRQB->dData0,     /* Handle */
                       pRQB->dData1);    /*  dSize */
    break;
case 11 :       /* CreateFile */
    erc = CreateFileM(pRQB->pData1,      /* pFilename  */
                      pRQB->cbData1,     /* cbFilename */
                      pRQB->dData0,      /* Attributes */
                      pRQB->RqOwnerJob); /* iJob Owner */
    break;
case 12 :       /* RenameFile */
    erc = RenameFileM(pRQB->pData1,      /* pCrntName  */
                      pRQB->cbData1,     /* cbCrntName */
                      pRQB->pData2,      /* pNewName */
                      pRQB->cbData2,     /* dcbNewName */
                      pRQB->RqOwnerJob); /* JobNum */
    break;
case 13 :       /* DeleteFile */
    erc = DeleteFileM(pRQB->dData0);     /* Handle  */
    break;
case 14 :       /* CreateDirectory */
    erc = CreateDirM(pRQB->pData1,      /* pPath */
                     pRQB->cbData1,     /* cbPath */
                     pRQB->RqOwnerJob); /* JobNum */
    break;
```

continues

Listing 27.1. continued

```
        case 15 :       /* DeleteDirectory */
            erc = DeleteDirM(pRQB->pData1,      /* pPath */
                             pRQB->cbData1,     /* cbPath */
                             pRQB->dData0,      /* fAllFiles */
                             pRQB->RqOwnerJob); /* JobNum */
            break;
        case 16 :       /* GetDirSector */
            erc = GetDirSectorM(pRQB->pData1,       /* pPath    */
                                pRQB->cbData1,      /* cbPath   */
                                pRQB->pData2,       /* pSectRet */
                                pRQB->cbData2,      /* cbRetMax */
                                pRQB->dData0,       /* SectNum */
                                &i,                 /* for LBARet */
                                &i16,               /* for DirClstr */
                                pRQB->RqOwnerJob); /* JobNum   */
            break;
        default :
            erc = ErcBadSvcCode;
            break;
    }

    merc = Respond(FMsg[0], erc);

  }
}       /* forever */
}

/***************** PUBLIC BLOCKING CALLS FOR FILESYSM *************
 These calls query the TSS Exhange and use it to make Requests
 to the file system service on the behalf of the caller.
 These calls are fully reentrant! No static data!
*****************************************************************/

U32 far _OpenFile(char *pFilename,
                  long dcbFilename,
                  long Mode,
                  long Type,
                  long *pdHandleRet)
{
long erc, exch, rqhndl, i, msg[2];
    if (dcbFilename == 3)
    {
        if (CompareNCS(pFilename, "NUL" , 3) == -1)
        {
            *pdHandleRet = 0;
            return(0);
        }
        else if (CompareNCS(pFilename, "KBD" , 3) == -1)
        {
            *pdHandleRet = 1;
            return(0);
        }
        else if (CompareNCS(pFilename, "VID" , 3) == -1)
        {
            *pdHandleRet = 2;
            return(0);
        }
```

```
        else if (CompareNCS(pFilename, "LPT" , 3) == -1)
        {
            erc = DeviceOp(3, 10, 0,    0,  &i);   /* 10=Open */
            if (!erc)
                *pdHandleRet = 3;
            return(erc);
        }
    }

    GetTSSExch(&exch);      /* No error will come back! */
    erc = Request(fsysname, 1, exch, &rqhndl,
                  1,                              /* 1 Send ptr */
                  pFilename, dcbFilename,
                  pdHandleRet, 4,
                  Mode, Type, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _CloseFile(unsigned long dHandle)
{
long erc, exch, rqhndl, i, msg[2];

    if (dHandle < 3)
        return(0);
    else if (dHandle == 3)
    {
        erc = DeviceOp(3,  11,   0,   0,  &i);    /* 11 = Close */
        return(erc);
    }

    GetTSSExch(&exch);
    erc = Request(fsysname, 2, exch, &rqhndl,
                  0,                              /* 0 Send ptr */
                  0, 0,
                  0, 0,
                  dHandle, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _ReadBlock(long dHandle,
                   char *pDataRet,
                   long nBlks,
                   long dLFA,
                   long *pdnBlksRet)
{
long erc, exch, rqhndl, msg[2], i;
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 3, exch, &rqhndl,
                  1,                              /* 1 Send ptr */
```

continues

Listing 27.1. continued

```
                pDataRet, nBlks,
                pdnBlksRet, 4,
                dHandle, dLFA, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc)
        return(erc);
    if(msg[1])
    {
        DeviceStat(10, &FDDevStat, 64, &i);
    }
    return(msg[1]);
}

/***********************************/
U32 far _WriteBlock(long dHandle,
                    char *pData,
                    long nBlks,
                    long dLFA,
                    long *pdnBlksRet)

{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 4, exch, &rqhndl,
                    1,                          /* 1 Send ptr */
                    pData, nBlks,
                    pdnBlksRet, 4,
                    dHandle, dLFA, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _ReadBytes(long dHandle,
                   char *pDataRet,
                   long nBytes,
                   long *pdnBytesRet)
{
long erc, exch, rqhndl, msg[2], i;
    if (dHandle == 0)
    {
        *pdnBytesRet = 0;
        return(0);
    }
    else if (dHandle == 1)
    {
        i = 0;
        while (i < nBytes)
        {
            ReadKbd(*pDataRet++, 1);
            i++;
        }
        *pdnBytesRet = i;
        return(0);
```

```
    }
    else if ((dHandle == 2) || (dHandle == 3))
    {
        *pdnBytesRet = 0;
        return(ErcWriteOnly);
    }

    GetTSSExch(&exch);
    erc = Request(fsysname, 5, exch, &rqhndl,
                  1,                              /* 1 Send ptr */
                  pDataRet, nBytes,
                  pdnBytesRet, 4,
                  dHandle, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _WriteBytes(long dHandle,
                    char *pData,
                    long nBytes,
                    long *pdnBytesRet)

{
long erc, exch, rqhndl, VidAttr, msg[2];

    if (dHandle == 0)
    {
        *pdnBytesRet = nBytes;
        return(0);
    }
    else if (dHandle == 1)
    {
        *pdnBytesRet = 0;
        return(ErcReadOnly);
    }
    else if (dHandle == 2)
    {
        GetNormVid(&VidAttr);
        TTYOut(pData, nBytes, VidAttr);
        *pdnBytesRet = nBytes;
        return(0);
    }
    else if (dHandle == 3)
    {
        erc = DeviceOp(3, 2, 0,  nBytes, pData);   /* 2 = CmdWriteRec */
        return(erc);
    }

    GetTSSExch(&exch);
    erc = Request(fsysname, 6, exch, &rqhndl,
                  1,                              /* 1 Send ptr */
                  pData, nBytes,
                  pdnBytesRet, 4,
                  dHandle, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
```

continues

Listing 27.1. continued

```
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far  _GetFileLFA(long dHandle,
                     long *pdLFARet)

{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcEOF);

    GetTSSExch(&exch);
    erc = Request(fsysname, 7, exch, &rqhndl,
                  0,                              /* 0 Send ptrs */
                  pdLFARet, 4,
                  0, 0,
                  dHandle, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _SetFileLFA(long dHandle,
                    long dNewLFA)
{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 8, exch, &rqhndl,
                  0,                              /* 0 Send ptrs */
                  0, 0,
                  0, 0,
                  dHandle, dNewLFA, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _GetFileSize(long dHandle,
                     long *pdSizeRet)
{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 9, exch, &rqhndl,
                  0,                              /* 0 Send ptrs */
                  pdSizeRet, 4,
                  0, 0,
                  dHandle, 0, 0);
```

```
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _SetFileSize(long dHandle,
                     long dSize)
{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 10, exch, &rqhndl,
                  0,                          /* 0 Send ptrs */
                  0, 0,
                  0, 0,
                  dHandle, dSize, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _CreateFile(char *pFilename,
                    long cbFilename,
                    long Attribute)
{
long erc, exch, rqhndl, msg[2];
    GetTSSExch(&exch);
    erc = Request(fsysname, 11, exch, &rqhndl,
                  1,                          /* 1 Send ptrs */
                  pFilename, cbFilename,
                  0, 0,
                  Attribute, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _RenameFile(char *pCrntName,
                    long cbCrntName,
                    char *pNewName,
                    long cbNewName)
{
long erc, exch, rqhndl, msg[2];
    GetTSSExch(&exch);
    erc = Request(fsysname, 12, exch, &rqhndl,
                  2,                          /* 2 Send ptrs */
                  pCrntName, cbCrntName,
                  pNewName, cbNewName,
                  0, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}
```

continues

Listing 27.1. continued

```
/***********************************/
U32 far _DeleteFile(long dHandle)
{
long erc, exch, rqhndl, msg[2];
    if (dHandle < 4)
        return(ErcNotSupported);
    GetTSSExch(&exch);
    erc = Request(fsysname, 13, exch, &rqhndl,
                  0,                              /* 0 Send ptrs */
                  0, 0,
                  0, 0,
                  dHandle, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _CreateDir(char *pPath,
                   long cbPath)
{
long erc, exch, rqhndl, msg[2];
    GetTSSExch(&exch);
    erc = Request(fsysname, 14, exch, &rqhndl,
                  1,                              /* 1 Send ptrs */
                  pPath, cbPath,
                  0, 0,
                  0, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _DeleteDir(char *pPath,
                   long cbPath,
                   long fAllFiles)
{
long erc, exch, rqhndl, msg[2];
    GetTSSExch(&exch);
    erc = Request(fsysname, 15, exch, &rqhndl,
                  1,                              /* 1 Send ptrs */
                  pPath, cbPath,
                  0, 0,
                  fAllFiles, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/***********************************/
U32 far _GetDirSector(char *pPathSpec,
                      long cbPathSpec,
                      char *pEntRet,
                      long cbEntRet,
                      long SectNum)
{
```

```
long erc, exch, rqhndl, msg[2];
    GetTSSExch(&exch);
    erc = Request(fsysname, 16, exch, &rqhndl,
                  1,                              /* 1 Send ptrs */
                  pPathSpec, cbPathSpec,
                  pEntRet, cbEntRet,
                  SectNum, 0, 0);
    if (!erc) erc = WaitMsg(exch, msg);
    if (erc) return(erc);
    return(msg[1]);
}

/*****************************************
 Initialization Routine for the File system.
 This is called by the Monitor where any errors
 will be reported.
*****************************************/

U32 InitFS(void)
{
U32 erc, i, j;
U8 *pMem;

  /* Allocate FAT buffers and initialize FAT related structures.
     We will allocate 24Kb worth of buffers (6 Pages, 16 buffers).
   */

  Fat[0].pBuf = FatBufA;  /* floppy fat buffers */

  erc = AllocOSPage(2, &pMem);

  if (!erc)              /* hard disk fat buffers from allocated mem */
    for (i=1; i<nFATBufs; i++)
    {
      Fat[i].pBuf = pMem;           /* Make pBuf point to each buffer */
      pMem += 512;                  /* Next buffer in allocated memory */
    }

  /* Allocate and initialize FCBs and FUBs. This is enough FCBs and
     FUBS for 128 openfiles.  (Good enough to start...)
  */

  if (!erc)
      erc = AllocOSPage(2, &paFCB); /* a pointer to allocated FCBs. */
  if (!erc)
      FillData(paFCB, 8192, 0);
  if (!erc)
      erc = AllocOSPage(1, &paFUB); /* a pointer to allocated FUBs. */
  if (!erc)
      FillData(paFUB, 4096, 0);

  if (!erc) erc = read_PE();    /* reads partition tables */

  if (!erc) erc = SetDriveGeometry(12);

  if (!erc)
  {
```

continues

Listing 27.1. continued

```
    erc = SetDriveGeometry(13);
    if (erc == 663) erc = 0;        /* may be invalid drive */
  }

  StatFloppy(0);
  StatFloppy(1);

    /* read all logical drive boot sectors */

  if (!erc)
  {
    for (i=0; i< nLDrvs; i++)
    {
      if (Ldrv[i].DevNum != 0xff)
      {
            read_BS(i);
      }
    }
  }

  for (i=0; i<nLDrvs; i++)
    if (Ldrv[i].DevNum != 0xff)
    {
    j=12;
      if (Ldrv[i].fFAT16)
          j=16;
      xprintf("%c: Heads %d, Sec/Trk %d, Sec/Clstr %d, Dev %d, FAT%d \r\n",
            i+0x41,
            Ldrv[i].nHeads,
            Ldrv[i].nSecPerTrk,
            Ldrv[i].SecPerClstr,
            Ldrv[i].DevNum,
            j);
    }

  if (!erc)
      erc = AllocExch(&FSysExch);

    /* Start the filesystem task at a decently high priority (5).
    This should be higher than the Monitor status task and even the
    Keyboard. */

  if (!erc)
    erc = SpawnTask(&FSysTask, 5, 0, &FSysStack[511], 1);

  if (!erc)
   erc = RegisterSvc(fsysname, FSysExch);

return erc;

}
```

V

Additional Software

28

DASM: A 32-Bit Intel-Based Assembler

DASM is an Intel-based, 32-bit assembler designed for the development of the MMURTL operating system. It is also used to develop software to run on MMURTL.



The version included with this book runs in MS-DOS. The source code is included. To develop an operating system, you must run the assembler in another environment. Running in MS-DOS will make DASM easy for you to use as a development tool for your own system. Of course, better assemblers exist, but none are as inexpensive, and few come with source code. How you design your memory-management and loading techniques will determine how much you must modify DASM to suit your needs.

Unlike most other Intel-based assemblers, DASM combines the functions of an assembler and a linker. It produces an executable file called a RUN file (along with DLLs and device drivers). A RUN file is analogous to the MS-DOS executable file (.EXE). It can be modified quite easily to output object modules compatible with 32-bit linkers on other systems, if that is what you need.

DASM Concepts

Most assemblers and compilers produce what is called *object code.* This is an intermediate form of machine code, and binary data stored in a file is called an *object module.* Object modules are usually combined to produce executable code that will be ready to load and run by an operating system.

Using object code modules has these advantages:

- You can place modules of object code in a library and search the library for what you need (unresolved externals).
- You can break your project (the program you are working on) into smaller pieces and compile them separately as you change them. When you want to produce an executable file, you "link" all of these object modules together with a program called a linker.
- Local variables, labels, and procedures may be hidden from other modules.

Separate, relocatable modules are also required when you are programming with a "segmented" memory model.

DASM eliminates the need for object code. This means you don't need a separate linker. DASM, however, still provides separate compilation of source code modules and library search functions—advantages normally associated with a linker and object code.

For example, in systems that use a separate linker, with three C source files, you do this:

1. Compile Module 1 into object module.
2. Compile Module 2 into object module.

3. Compile Module 3 into object module.
4. Place object modules in library (if desired).
5. Link object module, making an executable file.

With MMURTL and three C source files, using DASM you do this:

1. Compile Module 1 into assembly language.
2. Compile Module 2 into assembly language.
3. Compile Module 3 into assembly language.
4. Assemble (make an executable file).

The difference between these two systems is in what the compiler produces. In other systems, the compilers and assemblers produce object modules. With MMURTL, the CM32 compiler produces assembly language, and the assembler produces the executable file (the RUN file).

The previously mentioned advantages for systems producing object code are not lost in the MMURTL development system. You can still edit and compile separate modules to speed the development cycle, provide organization, and assist in code reuse. External library code can still be included in your project, and local (nonpublic or static) variables, labels, and functions are still hidden in each module. The one obvious difference is that all of the library code is in assembler source format.

To eliminate the need for a linker, each of the *public* variables and code labels in your source libraries are listed in a text file that is searched by the assembler. A small utility is included that automatically searches and indexes them into the required text format for you. This is your librarian.

Because there is no linking, the development cycle is further reduced.

Using DASM

DASM combines the functions of an assembler and a linker. Because of this combination, you don't specify an assembler source file for DASM on the command line. Instead, you provide the name of an assembly language template file (.ATF). The template file is actually an assembler source file that defines the structure of your program. It provides complete control of how your program is assembled and linked. The template file may contain standard INCLUDE files for MMURTL entry code and exit code required by compilers as well as .ASM file names. The template file also contains statements to set stack size, statements to set virtual memory offsets, and more.

The following is a template file for the Hello World program.

Listing 28.1. Simple application template file.

```
;STANDARD APPLICATION TEMPLATE FILE

;------- SYSTEM Entry/Exit module/commands go here

.DATA                       ;Start data segment
.VIRTUAL 0h
.CODE
.VIRTUAL 0h

;------- USER modules Begin here

.INCLUDE Hello.ASM          ;Your assembly module
.INCLUDE \CM32\CM32.AS    ;Standard Entry/Exit code

;------- USER Library Search files begin here

.SEARCH \CM32\LIB\CM32.PUB
.SEARCH \MMURTL\LIB\OS.PUB

.END
```

The DOT commands (commands preceded by a period), INCLUDE files, and library code search file lists are all DASM needs to build your complete application and turn it into a run file. To build the application, compile Hello.c, which produces the Hello.ASM file that contains code and data assembly language sections. Then at the command prompt you execute DASM providing the name of the template file:

```
>DASM Hello.ATF <Enter>
```

Hello.ATF is the assembly language template file shown previously. A run file named Hello.RUN is produced if no errors occurred during the assembly and link processes.

The ATF file in Listing 28.2 is for a seemingly more complicated program including more modules.

Listing 28.2. A more complicated template file.

```
;STANDARD APPLICATION TEMPLATE FILE

;- Commands that affect the entire program go here

.DATA                       ;Start data segment
.STACK 1000                 ;This will default to 1 Page (4K)

;- USER modules Begin here

.INCLUDE Editor.ASM         ;Main Editor Module
.INCLUDE Display.ASM
```

```
.INCLUDE Search.ASM
.INCLUDE LoadSave.ASM
.INCLUDE BufMgmt.ASM

;- USER Library Search files begin here

.SEARCH \CM32\LIB\CM32.PUB
.SEARCH \MMURTL\LIB\OS.PUB

.END
```

As you can see, this ATF file is not much more complicated than Hello.ATF; it just has a few more modules.

Command Line Options

The format of the DASM command line is as follows:

```
DASM TemplateFile [RunFile] /L /S /E /D /V
```

Table 28.1 presents the command-line options (switches) available with DASM.

Table 28.1. The DASM command-line options.

Flag	*Description*
`/L`	Complete list file generated
`/S`	Include SYMBOLS (only in complete list file)
`/E`	List file for errors/warnings only
`/D`	Process as dynamic link library (.DLL built)
`/V`	Process as device driver (.DRV built)

These options are *not* case-sensitive and are explained in the following sections.

List File

The `/L` option produces a complete list file of all actions the assembler performs on your source code. The list file is automatically named SourceName.LIS. It has the same prefix as your source file with the .LIS extension. The list file format is as follows:

```
Line   Address  Action/Code/Data   Source Code
00001  0000000  <- DSeg begins     .DATA
```

The line number is the line for the current module you are assembling. The address is the offset in the segment you are currently in (`CSeg` or `DSeg`). The Action/Code/Data column shows what action DASM took based on the source code for that line. The list file may show instructions, data storage, or command options. Your source code is shown in the far right column.

NOTE

Generating a complete list file takes considerable time and disk space. You should use the /L option only when necessary.

Errors Only

The /E option also produces a list containing only the error statements. If no list file is specified because you didn't use the /L or /E options, all of the errors are displayed on the screen.

Approximately 70 errors are generated by DASM. They are explained in text form but also contain a reference number; you can use the number to refer to the section at the end of this chapter for more detailed information about errors.

Dynamic Link Libraries

The /D option causes DASM to produce a MMURTL-compatible dynamic link library (.DLL) instead of a .RUN file. A DLL in MMURTL may not contain any data. If the DSeg offset is not 0 at the end of the compile, an error is produced. DLLs in MMURTL may contain only code and stack-based variables. DLLs must be fully re-entrant. See Chapter 10, "Systems Programming."

Device Drivers

The /V option causes DASM to produce a MMURTL-compatible device driver (.DRV) instead of a .RUN file. A device driver in MMURTL may contain code and data. Device drivers in MMURTL do not necessarily have to be fully re-entrant. See Chapter 10, "Systems Programming," for more information on device drivers.

DASM Source Files

DASM isn't a fully parameterized macro assembler. It does, however, support simple macro substitution through the use of the EQU statement. DASM operates with DOT commands.

NOTE

DOT commands are reserved command words that begin with a period and are the first non-whitespace data on a line.

Local (Nonpublic) Declarations

DASM allows nesting up to four levels of INCLUDE files (five if you include the level 0 template file). When each of the INCLUDE files listed in the .ATF file are opened, they cause DASM to clear all *nonpublic* labels and variables from the symbol table. The next level and subsequent levels do not do this. You can still write .ASM files that use INCLUDE files for shared local declarations. In the first example (Hello.c and Hello.ASM), if there were an INCLUDE statement in Hello.ASM, it would *not* clear local names from the symbol table. However, if Hello.ASM were broken into two files (Hello1.ASM and Hello2.ASM) and both were listed in the .ATF file, when DASM closed Hello1.ASM and then opened Hello2.ASM, all of the local labels and variables listed in Hello1.ASM would be "forgotten."

Memory Model

DASM is designed specifically for the MMURTL operating system. MMURTL is a paged-memory OS. If you are familiar with the Intel processor segmentation model, you know that code and data may reside in multiple segments. This was a requirement when programs were restricted to a 64K segment. You may also be familiar with the infamous small memory model: a program was made of a single code segment, and the data and stack shared another segment. This is effectively MMURTL's memory model for all programs. Code and data segments in a MMURTL application are not limited to 64K. In theory, the segments can be up to one gigabyte. A program's stack resides in the data segment just as in the DOS small memory model. The code, data, and stack of your program are logically separated in memory on 4096 byte pages. What all this means to the programmer is *ease of use*. DASM does all the work by lining up the code and data from different ASM source modules. Address offsets are completely resolved at assemble time, and DLL offsets are resolved at load time.

DASM Program Structure

A stand-alone DASM program is made of code and data segment parts that may be broken up through one or more source files. Two DOT commands determine the current segment. DASM module structure in Listing 28.3 illustrates this:

Listing 28.3. Alternating data and code in a template file.

```
.STACK 4096
        ;Minimum stack size in bytes

.DATA
        ;Start data segment
        ;Variables, data, data EQUs, etc. are defined here
```

continues

Listing 28.3. continued

```
.CODE
.START
        ;Entry point code defined here
.DATA
        ;Continue data segment
        ;More data defined here
.CODE
        ;Continue code segment
        ;More code here
.END
```

DOT Commands

A DOT command is a reserved command word preceded by a period (.). This command must be the only active text on the line (trailing comments are allowed). Some DOT commands have a single parameter following the command word. Only a few DOT commands exist, because the memory model in MMURTL is so simple to use. The following DOT commands are recognized by DASM and are described in detail in the following paragraphs:

```
.DATA
.CODE
.START
.INCLUDE filename
.ALIGN n
.VIRTUAL n
.END
.SEARCH filename
.STACK n
.VERSION string
.DATE
.DEBUG n
```

Each MMURTL program has only two segments. MMURTL uses segmented memory only for the purpose of protection, which is transparent to the programmer. As far as the programmer is concerned, there is only *code*, *data*, and a *stack*.

The following are detailed descriptions of the DOT commands and their parameters (if any):

- `.DATA` This command indicates the start or continuation of the data segment. The data segment is where you define your data storage variables. No processor instructions are allowed in the data segment.
- `.CODE` This command indicates the start or continuation of the code segment where processor instructions and read-only storage may be defined. DASM is designed to recognize the complete 80386 instruction set. Instructions specific to the 80486 or Pentium processors should be coded in-line with DB statements if required.

- `.START` This command is placed just before the instruction you want as your program entry point (where it begins execution when the program is loaded). Only one `.START` command is allowed in your program, no matter how many separate modules you have linked together. It must be in a code segment; otherwise, an error will occur. If you are using one of the provided format files for use with the CM32 compiler, `.START` is already included in the \CM32\LIB\SUPPORT.ASM file. All you have to do is make sure you have defined `main()` in one of your modules.
- `.STACK (n)` The *n* parameter is the number of bytes in the initial program stack for the main program thread (first JOB task). If no stack command is found, the MMURTL loader automatically allocates a one-page stack (4096 bytes). The `.STACK` statements are additive. This enables you to add to the stack in a heavily recursive library module, if necessary. MMURTL allows true multithreading. Memory must be allocated (or used from your existing data segment) as a stack for each additional task that you create. The number of initial pages of stack is limited to 256 (1MB). Stack requirements are discussed in much greater detail in Chapter 9, "Application Programming."
- `.VIRTUAL xxxx` The parameter *xxxx* is a value that tells the assembler where to assume the segment is loaded. This command is used in the data segment or code segment to set the default initial segment offset value the assembler works with. In virtual (paged) memory systems (such as MMURTL), there are advantages to specifying where the assembler assumes the segment will be loaded or relocated. This command should appear only once in each segment, and it must appear before any code, data, or labels are defined. Application programs don't need to use the `.VIRTUAL` command. The virtual offsets are 0 by default for the code segment and the data segment. Consider the following example:

```
.DATA
.VIRTUAL 40000000h
```

  These lines tell DASM that the data segment will begin at linear address 40000000h (1GB). Subsequent continuations of the segment in any of your source modules should *not* have a `.VIRTUAL` command in them. This does not mean that the module will actually load in memory at that linear address; the `.VIRTUAL` command simply tells DASM to assume it does. You should *not* assume this yourself.
- `.ALIGN boundary` where *boundary* is `WORD`, `DWORD`, or `PARA`. This command enables you to align the next storage declarations in the data segment on a memory address multiple of the specified size. This is used rarely, but it may be useful in situations such as optimizing file buffers for direct disk reads and writes. `WORD` means an even address. `DWORD` means an address that is modulo 4. `PARA` means an address that is modulo 16 (paragraph aligned).
- `.INCLUDE filename` This statement opens *filename,* and begins processing data statements and instructions. It also clears all local variables and labels from the symbol table if it is a "first level" include statement (listed in the .ATF file). Local variables are

those not defined as PUBLIC or EXTRN, which is different from many assemblers, but provides the same functionality of separately linked modules to allow hidden local variables and code labels. In C, functions and variables defined as static will not be listed as PUBLIC in the assembler file.

- .SEARCH *filename.pub* This searches a public text index file for unresolved variables, code label names, and function label names. PUB files are specially formatted text files that list public code and variables in assembly language source files. The linker includes the assembly source files listed in the PUB file if they contain the public name of an unresolved external. The .PUB files may be read several times if there is included a new module that defined externals not already in the DASM's symbol table.
- .VERSION *string* This allows the definition of a string that will be included in the RUN file header so the program version can be *easily* obtained without running the program or without searching with a binary editor or viewer. The string can be up to 70 characters in length, begins after the first whitespace following the .VERSION command, and ends when the end of line is detected. All characters beyond 70 are truncated.
- .DATE This tells the assembler to place the current date and time in the RUN file header in text format.
- .DEBUG *n* DASM allows several debugging modes, and *n* is the number of the mode you want activated. This command causes additional code (and possibly data) to be added to your RUN file for debugging purposes, or causes a separate symbol table to be generated that a debugger can open, read, and use to find publics, line numbers, and so on.
- .END This tells the assembler you are done with all source modules. The .END command is used only at the end of the format file (or module, if this is a single-file assembly language program), not at the end of each segment piece or assembler source module. A segment type ends by default when a new segment type begins. The .END command indicates there is no more source text to process.

DASM Versus DOS Assemblers

DASM programs are generally very similar to DOS assembler programs. A major difference is the lack of SEGMENT commands and segment alignment options, for they aren't required with MMURTL. Remember, with MMURTL, your program sees a 2GB, 32-bit, linear address space. ASSUME commands are also absent from DASM. They aren't needed with "flat" application memory. DASM assumes that DS, ES, SS, FS, and GS, all refer to your data segment. CS always refers to your code segment. If you need to read data from your code segment, segment register prefixes work fine, although I can't imagine where you would need to do so. Code segment pages in MMURTL are read-only. Consider the following example:

```
CS:[EBX+MyCodeLabel+5]
```

Another difference is the lack of the PROC (Procedure) attribute. The only operational purpose PROC served was to tell the assembler what type of return instruction to use. Code labels in MMURTL serve as the PROC identifier, and you specify whether or not you want them public. Public labels can be referenced by code in other modules and will be resolved at assemble time. You also specify a RETN (near) or RETF (far) instruction manually.

NOTE

Only MMURTL operating system software uses the RETF instruction because *all* calls to your code will be 32-bit near calls (relative or indirect). In fact, the only *far* calls from your program will be to the operating system or its device driver interfaces, which are all predefined through processor call gates.

As with some DOS assemblers, references to forward code labels and forward data are allowed, as long as DASM can figure out the size of the reference. You will receive an error if it can't. References to data or code outside the current module *must* have an EXTRN declaration in advance. The assembler must be aware of its type to generate the correct instruction.

DASM Addressing Modes

DASM recognizes all valid 32-bit addressing modes for the 386/486 processors, with 16-bit addressing modes *not* allowed or recognized. DASM is a little fussier than most assemblers when it comes to address or memory operands. As you will see from the following examples, however, use of memory operands is fairly easy to understand, and close to the DOS assemblers. It follows closely the definitions in the *Intel 386/486 Programmer's Reference* manuals.

The following rules apply:

1. All address operands must be contained in square brackets [effective address], unless it is a simple address variable with no registers or scaling involved.
2. The *disp* (displacement) value shown in the following examples may be a variable name, a number, an expression involving constants, variables, or labels.
3. If the address operand is a single variable name, DASM assumes you are moving that variable's contents (the size it was defined) to or from the memory address or register. This works the same as DOS assemblers. The size of the variable is overridden if you use a register as the source or destination and it's not the same size as the variable, or if you force the size of the move with the modifiers BYTE PTR, WORD PTR, DWORD PTR, and FWORD PTR.
4. Table 28.1 shows the address constructs for memory operands in instructions. These are 32-bit addressing modes only.

Table 28.1. Specifying memory operands.

Format	*[Brackets Required?]*
`[disp32]`	optional
`[Reg32]`	required
`[Reg32+disp32]`	required
`[Reg32+Reg32]`	required
`[Reg32*Scale+disp32]`	required
`[Reg32+Reg32*Scale+disp32]`	required

The following paragraphs describe items in Table 28.1 and provide examples.

The `disp32` construct is a displacement value from the beginning of the segment. It is usually entered in your program as a simple variable name such as `MOV EAX, MyVar`. It may also be a number derived in a simple additive expression using a simple variable, such as `[MyVar+5]`. In the latter case, you should enclose the expression in square brackets to indicate that it is an address and not an immediate value, for immediate values may also be constructed with expressions containing variable names and/or code labels. An example would be the following:

```
Label2-Label1
```

A value that is the difference between the addresses of the two labels would not be considered an address after the calculation is performed. It becomes a constant in the code. No address "fixup" will be applied by the loader. This type of statement might be used to find out how large is a particular section of code.

`Reg32` is any 32-bit register (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP).

`Scale` will be number 2, 4, or 8 and indicates a special scaled addressing mode on the 386 processor. The `*2`, `*4`, or `*8` *must* follow the `Reg32` that you want to be scaled (multiplied) by that value. An example of a scaled instruction is this:

```
MOV AX, [MyArray+ECX+EBX*2]
```

If `MyArray` were a two-dimensional array of `WORDS` (16-bit values), ECX could contain the offset into the first dimension, while EBX had the index to the item you wanted in the second dimension. The `*2` scale would effectively multiply the index by the size of a `WORD` to ensure you accessed the correct second-dimension entry. This means high-level languages don't need to do their own multiplication for access to `WORD`, `DWORD` ,or `QWORD` (64-bit) entries for one- or two-dimensional arrays.

The actual syntax for memory operands is the same as the constructs shown in Table 28.1 above. Square brackets are required for all memory operands except simple variable names. Here are some examples:

```
.DATA
MyVar      DD 10           ;a DWORD variable
MyBArray   DB 100 DUP(0)   ;a single dimension byte array
MyWArray   DW 1600 DUP(0)  ;This is a 40 x 40 array of words

.CODE
    MOV MyVar, 10    ;Immediate value into DWORD MyVar
                     ;
```

The next example moves 20h into the tenth WORD of the fifth element of MyWArray. (for example, MyWArray[5][10] = 20h):

```
    MOV ECX, 5       ;Set up for the example
    MOV EBX, 10      ; "  "
    MOV [MyArray+ECX+EBX*2], 20h
```

The next example does the same thing as the preceding example. It shows you that addressing elements are accepted in any order, *but* all must be inside the square brackets:

```
    MOV [EBX*2+ECX+MyArray], 20h    ;same as above
                                    ;
            ;Now we move 0 into an byte of MyBArray
            ;that is indexed by some value in EDX
            ;
    MOV [MyBArray+EDX], 0h
.END
```

In the preceding examples, DASM knew what size the immediate value was because a variable of known size was used in the address. Consider this invalid instruction:

```
    MOV [ECX], 20h       ;Bad instruction
```

DASM has no way to know which ECX points to, a BYTE, WORD, or DWORD! In these cases, you must use address modifiers to tell DASM the size of the destination memory variable. DASM will sign extend the value as required. Here's an example:

```
MOV WORD PTR [ECX], 20h
```

This line says ECX is pointing to a WORD. The immediate value would be sign-extended (to 16 bits) and moved to the address contained in ECX.

Storage Declarations

Inside the data or code segments of your program you can issue storage instructions for data to be accessed by your program. This is done with the DB, DW, and DD instructions. Modifiers can be used to duplicate storage values for arrays. Data in your code segment is assumed to be read-only.

Labels (variable names) may precede any storage statement, as in this example:

```
    Count    DB 0
```

DB (Define Byte)

The DB instruction defines one or more bytes of storage. Immediate numeric values or strings may be defined with DB. Examples are as follows:

```
EOL DB    0Ah          ;creates one byte of labeled
                       ;storage with 0Ah as the value

    DB  5, 4, 6, 8     ;4 bytes with the values shown

    DB  100 DUP(0)     ;One hundred bytes filled with 0

    DB 'The Quick Brown Fox', 0Ah
                       ;Creates ASCII bytes with the text
                       ;followed by a 0Ah byte
```

DW (Define Word)

The DW instruction defines one or more WORDS of storage (two bytes). The syntax is the same as for DB except that string values are not allowed.

DD (Define Double Word)

The DD instruction defines one or more DWORDS of storage (four bytes). The syntax is the same for bytes except that quoted strings are not allowed.

DF (Define Far Word)

The DF instruction is used only to define a single 48-bit quantity (far pointer). It defines a six-byte quantity broken into two parts. The first part is the *offset* (four bytes), and the second is the selector (two bytes). The offset portion must be separated from the selector portion by a colon (:). An example follows:

```
pFarProc DF 00000000h : 0000h
```

The DF instruction is used for two things. First, it defines OS entry points in library module definitions. This allows you to *hardcode* the OS entry points. In other operating systems, this would not be a good idea, but because MMURTL uses 386/486 call gates, the OS code can move while its entry points will always stay the same. Second, the DF statement can also be used to define variables for use with the SIGT, LIDT, and LGDT, and other instructions that require a 48-bit (6-byte) storage size.

Application programmers will not generally use the DF statement except to access OS calls indirectly.

DASM Peculiar Syntax

DASM has some syntax differences from other assemblers you may be familiar with. The following sections cover these differences.

PUBLIC Declarations

By default, variables and code labels are not visible to other modules that are included as a program. To make them visible (to resolve external use), you must precede the label with the reserved word PUBLIC. Here are some examples:

```
.DATA

PUBLIC MyVar DB 10h      ;Will be visible to ALL modules
                         ;specified in the .ATF file

.CODE

PUBLIC MyProc:           ;Will be visible to ALL modules
    PUSH EBP             ;specified in the .ATF file
    MOV EBP, SP
    ...

.END
```

Note that this is a little different from most DOS assemblers. They usually require the PUBLIC declaration on a separate line.

EXTRN Declarations

External references from your module must be presented to DASM before their use. If you call external procedures directly, you must declare them somewhere inside your code segment before they are referenced. External variables must be declared in your data segment prior to use. There are three different types of externals in DASM: *data, near* code, and *far* code.

EXTRN data declarations will always be used when you access public variables from separate ASM source modules. Due to re-entrancy requirements, dynamic link libraries will not contain data.

EXTRN NEAR code declarations will always be used when you access code labels as CALL or JMP targets from separate ASM source modules or dynamic link libraries. The targets must be declared PUBLIC in the separate modules.

EXTRN FAR code declarations will be used only to access operating system calls (procedures). Each external label must be declared on a single line with the keyword EXTRN, the name, and the type.

The following are examples of external declarations:

```
.DATA
       ;External variables in another module you created

EXTRN MyVar1 DD
EXTRN bOne   DB
EXTRN bTwo   DB
EXTRN wOne   DW
EXTRN array  DB      ;With external arrays you need
                     ;only specify the size of the
                     ;elements of the array
```

```
.CODE
    ;Far procedures with absolute addresses
    ;resolved at link time are declared in the CODE segment

EXTRN FAR AllocExch
EXTRN FAR DeAllocExch
EXTRN FAR SendMsg
EXTRN FAR Sleep

    ;Near code direct calls to one of your module or a
    ;routine in an object module from a library

EXTRN NEAR MyProc1, MyProc2

.END
```

Note this is also a little different from most DOS assemblers. They usually require the `EXTRN` declaration on a separate line.

Labels and Address

In the following sections, I'll cover certain instructions and how they are used. An introduction to the instruction comes first, followed by a discussion of the instruction and finally a coding example.

Segment and Instruction Prefixes

With the 386 processor, memory references refer to the data segment by default, which is the value contained in the DS register. There are only two exceptions:

1. Instructions using the EBP and ESP registers (frame pointers and stack pointers) which use the SS segment register, and,
2. String instructions using the EDI register which uses the ES segment register.

The 386 enables you to specify a different segment for each of your memory references. By default, MMURTL makes DS = ES = SS = FS = GS, and all are set to your data segment. Chances are slim that you will need segment prefixes, unless you want to access data contained in your code segment or you want make use of the two additional segment registers (FS and GS) on the 386. MMURTL uses the FS and GS registers internally, and quite frankly, I can't imagine why you would need them with flat memory space, but they are supported in DASM for OS use. To use the segment prefix in a memory reference, simply place the register name with a *colon* (:) after it before the open square bracket of the memory reference; the segment prefix will be properly coded in the instruction. Examples are as follows:

```
MOV EAX, CS:[MyProc+10]

MOV AL, CS:[AppCodeLabel+EBX]
```

Instruction prefixes such as `LOCK` and `REPNE`, which may precede some instructions, are coded on the same line as the instruction they affect, which is typical of most 386/486 assemblers.

The REP (repeat) series of prefixes deals specifically with the string instructions such as REP MOVSB.

CALL and JUMP Addresses

The CALL and JMP instructions (including JMP on condition) may make forward references into the local module's code segment. This section describes all the address types for calls and jumps, and explains what DASM assumes about them. Coding examples are provided for clarity.

```
CALL LabelName    - Call to Near Pointer
JMP LabelName     - Jump to Near Pointer
Jc LabelName      - Jump on Condition to Near Pointer
```

These examples indicate a call or jump to a label anywhere in your program regardless of the module. All calls to code labels are considered NEAR and local to the current module unless previously defined EXTRN. This means they are assumed to be within a 32-bit offset from the call or jump instruction itself. If the called label were in a different module and not previously defined as external, then DASM assumes it is a forward reference and codes it as such, filling in the address/offset when it finds it. If the end of the module is reached and the label has not been found in the code segment, then an error is generated. If the label is defined EXTRN, then the label in the other module should have been defined as PUBLIC (for example, PUBLIC MyLabel:). A 32-bit offset relative to the next instruction is encoded as part of the instruction itself.

The following are examples:

```
    .CODE
    EXTRN MyLabel    ;Needed if in a different module

    CALL MyLabel     ;NEAR PTR is default if left out
    JMP  MyLabel
    JZ   MyLabel
    CALL MyLabel
    JMP MyLabel
    JNZ MyLabel
;-----------------------------------------------
```

The next is a *short* jump:

```
JMP SHORT     - Jump Short (+/- 127 bytes)

Jc SHORT      - Jump on Condition (+/- 127 bytes)
```

The two previous examples are a special case for jump instructions; they tell the assembler that the target label for the jump is less than 128 bytes away from the jump instruction. This form is three or four bytes smaller, and executes faster. If the label was already defined (preceding the jump instruction), DASM uses the short form of the instruction if possible without the SHORT modifier being present. The SHORT modifier is used if you jump to a label defined after the jump, but known to be close enough to be a short jump. If you use the SHORT modifier and the label turns out to be farther than 127 bytes, then an error is generated. An 8-bit offset relative to the next instruction is encoded as part of the instruction itself. The instructions are these:

```
    JMP SHORT MyLabel
    JZ SHORT MyLabel
;-------------------------------------------
```

Far code declarations are completely different from most other assemblers. The only far code you have is the operating system itself.

`CALL FAR PTR` - Call to a Far Pointer

`JMP FAR PTR` - Jump to a Far Pointer

The `FAR` modifier indicates a call or jump to a far address. With the MMURTL operating system, this would be a call to the operating system itself. With MMURTL, there is no need for a `FAR` jump, as all labels in a program are near (within a 32-bit offset). The `FAR` jump is included in DASM for operating system use only.

NOTE

The 48-bit address is encoded as part of the instruction.

Because the only time you need a `FAR` call is to reach the operating system, and because you will not actually be linking with a library to resolve these addresses, DASM encodes this as an immediate `FWORD` value in the instruction. You must define the values for each of the OS calls you make in this fashion, with the EQU statement as follows:

```
WaitMsg EQU 00000000h:40h
SendMsg EQU 00000000h:48h
```



NOTE

The indirect addresses for MMURTL public calls are listed in the file MPUBLICS.ASM in the same directory as the operating source code on the CD-ROM.

To use this type of call in DASM, a `FAR` declaration should have been made previously in the code segment of this module. The following example shows the external declaration. If the label was not previously defined as `FAR`, an error is generated.

```
.CODE

WaitMsg EQU 00000000h:40h
SendMsg EQU 00000000h:48h

    CALL FAR PTR WaitMsg
    CALL FAR PTR SendMsg
;-------------------------------------------
```

The following code shows calling indirectly:

`CALL DWORD PTR` —Call to a Near Indirect address

`JMP DWORD PTR` —Jump to a Near Indirect address

This code indicates an indirect call or jump to a near address. A `DWORD` variable in `DSeg` contains the address you are calling or jumping to. The following examples show the external declaration that would be in your data segment if required. If the label was not previously defined as either a local `DD` or an external `DD`, then an error is generated.

NOTE

The offset in the program's data segment of the variable containing the address is encoded as part of the instruction. Externals are fully resolved at assemble time.

```
.DATA                          ;In data segment
    EXTRN DD pMyProc           ;If not in this module
    pMyLabel DD OFFSET MyLabel ;Needed if it IS local

.CODE                          ;In code segment
    CALL DWORD PTR MyLabel
    JMP DWORD PTR MyLabel
```

You may also call far addresses indirectly. In fact, indirectly calling far addresses is the preferred way to call MMURTL procedures.

`CALL FWORD PTR` —Call a Far address indirectly

`JMP FWORD PTR` —Jump to Far address indirectly

This code indicates an indirect call or jump to a far address. This means that an `FWORD` variable contains the address you are calling (4-byte offset, 2-byte segment). The following example shows an external declaration and two ways to define a local declaration in your data segment. If the label was not previously defined as a local `DF`, a `DD/DW`, or an external `DF`, then an error is generated. In MMURTL, `CALL FWORD PTR` is yet another way to reach the OS calls. A register pointing to, or the memory address of the `FWORD` variable containing, the target 48-bit address is encoded as part of the instruction.

```
.DATA                  ;Somewhere in data segment
    EXTRN DF pFarProc  ;Needed if not in this module

    pFarProc DF 00000000:0000

    pFarProc DD Offset MyLabel
             DW CodeSeg        ;Code Seg is a constant

.CODE                          ;Somewhere in code segment
    CALL FWORD PTR pFarProc

    JMP FWORD PTR pFarProc     ;Not used in MMURTL Apps
```

386 Instruction Set

DASM supports all 32-bit 80386 instructions. The 386 instruction set is a subset of the 80486 and Pentium processors. Table 28.2 is an alphabetical listing of the instructions that DASM supports.

To find the actual binary encoding for the instruction and timing information, you should refer to the documentation for the processor you are working with. Examples of almost all of these instructions can be found in the accompanying source code.

Table 28.2. Supported instructions.

Instruction	*Description*
AAA	ASCII adjust after addition
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiply
AAS	ASCII adjust AL after subtraction
ADC	Add w/ carry
ADD	Add
AND	AND logically
ARPL	Adjust RPL field of selector
BOUND	Check array index against bounds
BSF	Bit scan forward
BSR	Bit scan reverse
BT	Bit test
BTC	Bit test & clear
BTR	Bit test & reverse
BTS	Bit test & set
CALL	Call a routine
CBW	Convert Byte to word
CWDE	Convert word to dword (Extend)
CLC	Clear carry
CLD	Clear direction
CLI	Clear interrupts
CLTS	Clear task switch flag in CR0
CMC	Compliment carry flag
CMP	Compare

Instruction	*Description*
CMPSB	Compare strings of bytes
CMPSW	Compare strings of words
CMPSD	Compare strings of dwords
CWD	Convert word to dword
CDQ	Convert dword to Quad
DAA	Decimal adjust accumulator
DAS	Decimal adjust AL after subtract
DEC	Decrement
DIV	Divide
ENTER	Make stack frame for procedure
HLT	Halt processor
IDIV	Integer divide
IMUL	Integer multiply
IN	Reads form port into AL, AX, EAX
INC	Increment
INSB	Read byte(s) from a port
INSW	Read word(s) from a port
INSD	Read dword(s) from a port
INT	Call to Interrupt
INTO	Call to Interrupt 4 if Overflow
IRET	Return from interrupt 16 bit
IRETD	Return From Interrupt 32 bit
JA	Jump if above
JNBE	Jump not below or equal
JAE	Jump above or equal
JNB	Jump not below (Same as JAE)
JNC	Jump no carry
JB	Jump below
JBE	Jump below or equal
JNA	Jump not above
JC	Jump if carry

continues

Table 28.2. continued

Instruction	*Description*
JNAE	Jump not above or equal
JCXZ	Jump if CX=0 (Short only)
JECXZ	Jump if equal or CX=0 (Short only)
JE	Jump if equal (Same as JZ)
JZ	Jump if zero
JG	Jump if greater
JGE	Jump if greater or equal
JNL	Jump not less than
JL	Jump if less
JNGE	Jump not greater or equal
JLE	Jump less or equal
JNG	Jump if not greater
JNE	Jump if not equal (Same as JNZ)
JNZ	Jump if not zero
JNLE	Jump not less or equal
JNO	Jump not odd
JNP	Jump not parity
JPO	Jump parity odd
JNS	Jump not signed
JO	Jump parity odd
JP	Jump if parity
JPE	Jump parity even
JS	Jump if signed
JMP	Jump unconditionally
LAHF	(no params)
LAR	Load access rights (rRGW, rRMW)
LEA	Load effective address (rRGW, mem)
LEAVE	(no params)
LGDT	Load Global Descriptor Table register
LIDT	Load Interrupt Descriptor Table register
LDS	Load Far Ptr into DS:General Register

Instruction	*Description*
LSS	Load Far Ptr into SS:General Register
LES	Load Far Ptr into ES:General Register
LFS	Load Far Ptr into FS:General Register
LGS	Load Far Ptr into GS:General Register
LLDT	Load Local Descriptor Table Register
LMSW	Load Machine Status Word (obsolete)
LOCK	Lock bus (prefix)
LODSB	Load string byte
LODSW	Load string word
LODSD	Load string dword
LOOP	Loop to label if CX <> 0
LOOPE	Loop to label if CX <> 0 and Zero bit set
LOOPZ	Same as LOOPE
LOOPNE	Loop to Label if CX <> 0 and Zero bit set
LOOPNZ	Same as LOOPE
LSL	Load Segment Limit (rRGW, rRMW)
LTR1	Load Task Register (rm16)
MOV	Move value into register or memory
MOV	Segment into 16-bit register
MOV	16-bit register into segment
MOV	Move 32-bit register into control register
MOV	Move control register into 32-bit register
MOV	Move 32-bit register into debug register
MOV	Move debug register into 32-bit register
MOV	Move 32-bit register into test register
MOV	Move test register into 32-bit register
MOVSB	Move Byte(s)
MOVSW	Move word(s)
MOVSD	Move dword(s)
MOVSX	Move Byte or word to dword and Sign extend
MOVZX	Move Byte or word to dword and Zero extend

continues

Table 28.2. continued

Instruction	*Description*
MUL	Multiply
NEG	Negate
NOP	No Operation (Same as XCHG EAX,EAX)
NOT	Logical NOT
OR	Logical OR
OUT	Output byte/word/dword to Port
OUTSB	Out String Bytes
OUTSW	Out String words
OUTSD	Out String dwords
POP	Pop a value from the stack (32-bit only)
POPAD	Pop all registers from stack
POPFD	Pop 32-bit flags from stack
PUSH	Push a value onto the stack
PUSHAD	Push all registers
PUSHFD	Push 32-bit flag register onto stack
RCL	Logical roll w/carry left
RCR	Logical roll w/carry right
ROL	Logical roll left
ROR	Logical roll right
REP	Repeat prefix (used with string instructions)
REPE	Repeat if equal
REPNE	Repeat if not equal
RETN	Return from NEAR call
RETF	Return from FAR call
SAL	Logical shift accumulator left
SAR	Logical shift accumulator right
SHL	Logical shift left
SHR	Logical shift right
SBB	Subtract w/borrow
SCASB	String compare and scan byte
SCASW	String compare and scan word

Instruction	*Description*
SCASD	String compare and scan dword
SETA	Set if above
SETAE	Set if above or equal
SETB	Set if below
SETBE	Set if below or equal
SETC	Set if carry
SETE	Set if equal
SETG	Set if greater than
SETGE	Set if greater than or equal
SETL	Set if less than
SETLE	Set if less than or equal
SETNA	Set if not above
SETNAE	Set if not above or equal
SETNB	Set if not below
SETNBE	Set if not below or equal
SETNC	Set if no carry
SETNE	Set if not equal
SETNG	Set if not greater than
SETNGE	Set if greater than or equal
SETNL	Set if not less than
SETNLE	Set if not less than or equal
SETNO	Set if not odd
SETNP	Set if not parity
SETNS	Set if not signed
SETNZ	Set if not zero
SETO	Set if odd
SETP	Set if parity
SETPE	Set if parity even
SETPO	Set if parity odd
SETS	Set if signed
SETZ	Set if zero

continues

Table 28.2. continued

Instruction	*Description*
SGDT	Store global descriptor table
SIDT	Store interrupt descriptor table
SHLD	Shift left dword
SHRD	Shift right dword
SLDT	Store local descriptor table (Not used in MMURTL)
SMSW	Store machine status word (Obsolete)
STC	Set carry flag
STI	Set interrupt flag
STD	Set direction flag
STOSB	Store string of bytes
STOSW	Store string of words
STOSD	Store string of dwords
STR	Store task register
SUB	Subtract
TEST	Logical test (AND with results in flags)
VERR	Verify
VERW	Verify Word
WAIT	Wait for NCP
XCHG	Exchange reg/reg or reg/mem
XLAT	Translate
XLATB	Translate BL
XOR	Logical XOR

Executable File Format

The MMURTL RUN file is one of three types of executable files supported by the MMURTL OS. The three types are these:

RUN	A MMURTL standard executable
DLL	A MMURTL dynamic link library
DDR	A MMURTL device driver executable

The files for all three types of loadable or executable files are essentially the same. The files are composed of *tagged* fields. The fields are in TLV format (Tag, Length, Value).

The format for the file is not fixed (meaning that it has no header with fixed-length fields) and is therefore expandable for future requirements. Also, programs can read the file and look for only the information they need and understand. Some fields are mandatory, whereas others are optional.

The tags are single, unsigned bytes with values 80H and above. This allows 128 individual pieces of information in the RUN file; which is far more than you should ever need for an executable file in MMURTL (remember, simplicity was the goal from the beginning).

The length is a 4-byte (32-bit) unsigned number. It is always included after each tag and contains the length of the data that follows it.

For example, the first tag in all RUN files is the *begin tag* (80h). It is followed by a 4-byte length with a value of 1 (00000001h). A single byte follows the length as indicated by the value of 1. The byte describes the file type (RUN, DLL, or DDR).

If the file type value were 1 and you did a HEX dump of the RUN file, the first seven bytes would look like this:

```
80h 01h 00h 00h 00h 00h 01h
```

The *tagged* fields must be included in numeric order in the file. This is *mandatory* and is for the benefit of the tag reader (the operating system loader or applications that read these files). It is also used for a consistency check. If the loader detects tags that are out of order, it will not load the file. Table 28.3 describes each of the tags.

Table 28.3. Tag usage.

Tag	*Description*	*Use*
80h	FILE ID	Mandatory
82h	VERSION STRING	Optional
83h	DATE/TIME STRING	Optional
84h	COMMENT	Optional
90h	INITIAL SEGMENT SIZES	Mandatory
92h	ASSUMED DATA OFFSET	Optional
94h	ASSUMED CODE OFFSET	Optional
96h	STARTING OFFSET	Mandatory
A0h	DLL IDENTIFIER	As Required

continues

Table 28.3. continued

Tag	*Description*	*Use*
B0h	CODE SEGMENT	Mandatory
B2h	DATA SEGMENT	Mandatory
C0h	CSEG DATA ADDRESS FIXUP	As Required
C1h	CSEG CODE ADDRESS FIXUP	As Required
C2h	DSEG DATA ADDRESS FIXUP	As Required
C3h	DSEG CODE ADDRESS FIXUP	As Required
C5h	DLL ADDRESS FIXUP	As Required
C8h	DLL PUBLIC	As Required
FFh	FILE END/CHECKSUM	Mandatory

Each of the tag types, length field use and the values for each tag are discussed in detail in the following sections.

TAG 80h FILE ID (Mandatory)

LEN 1
VAL 1 indicates a RUN file (standard executable)
2 indicates a DLL (dynamic link library)
3 indicates a DDR (device driver)

TAG 82h VERSION STRING (Optional)

LEN Length of version string
VAL Version string

TAG 83h DATE/TIME STRING (Optional)

LEN Length of string
VAL Date/Time string

TAG 84h COMMENT (Optional)

LEN Length of string
VAL Comment string

TAG 90h INITIAL SEGMENT SIZES (Mandatory)

LEN 12

VAL Three unsigned dwords that contain the sizes of the initial stack, code segment, and data segment in that order. The values for Code and Data should match the totals found in tags B0 and B2 (code and data) segments.

TAG 92h ASSUMED DATA OFFSET (Optional)

LEN 4

VAL An unsigned, 32-bit number containing the value the assembler assumed for the DSEG offset. If VAL is not included, the loader assumes 0.

TAG 94h ASSUMED CODE OFFSET (Optional)

LEN 4

VAL An unsigned, 32-bit number containing the value the assembler assumed for the CSEG offset. If VAL is not included, the loader assumes 0.

TAG 96h STARTING OFFSET (Mandatory)

LEN 4

VAL An unsigned, 32-bit number containing the offset in the code segment of the first instruction to execute after loading.

TAG A0h DLL IDENTIFIER (As Required)

LEN Length of DLL File Name

VAL Full file specification of a DLL file, which may have to be loaded if not already resident. This must be included if you call public DLL procedures from this DLL in your program.

TAG B0h CODE SEGMENT (Mandatory)

LEN Length of code segment

VAL Binary executable code of the length expressed in LEN. There may be one or more of these in a RUN file. If more than one is used, the order of the tags must be in the instruction sequence as required in the code segment.

TAG B2h DATA SEGMENT (Mandatory)

LEN Length of data segment

VAL Binary data of the length expressed in LEN. For DLLs, the LEN must be 0, and no data will follow. There may be one or more of these in a RUN file. If more than one is used, the order of the tags *must* be in the data sequence as required in the data segment.

TAG C0h CSEG DATA ADDRESS FIXUP (As Required)

LEN Multiple of 4

VAL One or more offset addresses (each address is 32 bits) in the code segment of a 32-bit value that refers to an address in the data segment which may change when the data segment is loaded (relocated). One or more of these tags may be in a single executable file. One tag may contain all the fixups, or multiple tags may be used.

For example, `MOV EAX, MyVar` encodes the address of `MyVar` in the data segment as part of the instruction. This address is an offset from zero or the value specified by the `VIRTUAL` command. The data segment may not be located at that address, so the MMURTL loader has to know how to change the value in the instruction.

TAG C1h CSEG CODE ADDRESS FIXUP (As Required)

LEN Multiple of 4

VAL Offset of one or more addresses in the code segment of a 32-bit value that refers to an address in the code segment that may change when the code segment is loaded (relocated). As with TAG C0, there may be one or more of these tags.

The following is an example of what would cause this:

```
MOV EAX, OFFSET MyCodeLabel
```

This encodes the address of `MyCodelabel` as part of the instruction. This address is an offset from the beginning of the code segment as understood by DASM. The code segment may not be located at that address, so the MMURTL loader has to know how to change the value in the instruction.

TAG C2h DSEG DATA ADDRESS FIXUP (As Required)

LEN Multiple of 4

VAL One or more offset addresses in the data segment of a 32-bit value that refers to an address in the data segment that may change when the data segment is loaded (relocated). As with TAG C0, there may be one or more of these tags. Here is an example:

```
pMyData DD OFFSET MyVar
```

This places the offset of a variable into another variable in the data segment. Once again, if the data segment is not located at the address DASM assumed, it must be changed after loading.

TAG C3h DSEG CODE ADDRESS FIXUP (As Required)

LEN 4

VAL Offset address in data segment of a 32-bit value that refers to an address in the code segment that may change when the code segment is loaded (relocated). As with TAG C0, there may be one or more of these tags. Consider the following example:

```
pMyCode DD OFFSET MyCodelabel
```

This places the offset of a code label (a procedure or function) into a variable in the data segment. Once again, if the code segment is not located at the address DASM assumed, it must be changed after loading.

TAG C5h DLL ADDRESS FIXUP (As Required)

LEN 4 + length of DLL public name

VAL Offset address in code segment of a 32-bit value that refers to a DLL `PUBLIC` procedure, followed by the DLL public name. DLL code references are *near* 32-bit *calls*. The value in the code segment will be zero until the loader resolves it. All DLL public names must be unique in MMURTL. See Chapter 10, "Systems Programming", for details on DLLs.

TAG C8h DLL PUBLIC (As Required)

LEN 4 + length of DLL public name

VAL Offset address in code segment of the entry point for a DLL public procedure, followed by the DLL public name. This tag type should only appear in DLL files. All DLL public names must be unique in MMURTL. See Chapter 10, "Systems Programming," for details on DLLs.

TAG FFh FILE END/CHECKSUM (Mandatory)

LEN 4

VAL 32-bit simple check sum of the entire file prior to this tag. This means you simply total the value of each byte in a 32-bit variable while ignoring overflow. This value should match your total. If not, you can assume the file is corrupt.

Error Codes From DASM

The following errors can be returned from DASM. Each is assigned a number, followed by text that may be displayed, and a description of what may have caused it.

1.	`Invalid expression, ')' expected`	You have unbalanced parenthesis in an expression.
2.	`Invalid expression, value expected`	You are attempting a math operation on a nonnumeric.
3.	`Value expected after unary '-'`	A numeric value is expected after a single minus sign.
4.	`Too many digits for numeric radix`	Thirty-two digits are allowed for base 2, nine for base 10, and eight for base 16.
5.	`Invalid character in a number`	You have probably left off the "h" to indicate a hex number and a letter was found in it.
6.	`Unterminated string`	You have omitted the trailing quotes in a data storage statement.

7. `Unrecognized character`	More than likely, a letter was found in a number that doesn't fit its radix.
8. `Invalid alignment specified`	Only `WORD` and `DWORD` are allowed in the `.ALIGN` statement.
9. `Start command only allowed in CSEG`	`.START` was found in DSEG.
10. `Virtual command must be first in segment`	It must also be in the first occurrence of that segment. Check your ATF file.
11. `Invalid virtual value`	0h to 7FFFFFFFh is the limit.
12. `Starting address not found`	You must have a `.START` statement somewhere in your code or in included library files.
13. Reserved	
14. `Invalid command`	DOT What??

The following errors all deal with instruction format. You should look at the section of this chapter dealing with memory references to help understand the specific error.

15. `Invalid operand`
16. `Invalid segment register use`
17. `Invalid scale value 'Reg*?'`
18. `Scale value expected (*2,*4,*8)`
19. `Too many address scale values`
20. `Invalid register for memory operand`
21. `Invalid memory operand`
22. `Offset must be from data segment`
23. `Nested brackets`
24. `Unbalanced brackets`
25. `Invalid operand size attribute`

26–31. Reserved

32. `Unknown token in operand array`
33. `Too many operands or extra character`
34. Reserved
35. `Invalid expression or numeric value`
36. `Operand expected before comma`
37. Reserved

38. `Invalid character or reserved word in operand`
39. `Relative jump out of range`
40. `Operand size NOT specified or implied`
41. `Instructions not allowed in data segment`
42. `Instruction expected after prefix`
43. `Mismatched sizes in operands.` — This indicates you have specified an operand size, and it didn't match the source or destination operand size, for example, `MOV EAX, BYTE PTR Variable`.
44. `Wrong operand type for instruction`
45. `Incorrect format for memory operand`

The following errors all deal with data storage:

46. `Strings only valid for DB storage` — DW "xxx" is not allowed.
47. `Expected '(' after 'DUP'` — Duplicated storage values must be parenthesized.
48. `Storage expected between commas` — DB 23,34,??,
49. `':' not expected` — A colon is not allowed between data segment labels and storage definitions.
50. `DWord storage required for OFFSET` — All segment offsets are 32-bit in MMURTL.
51. `Invalid storage value`

52–53. Reserved

54. `':' expected after last label` — A semicolon is required after a label in the code segment.
55. `Macro not allowed in lexical level 0`
56. `EQU or Storage expected`

57–62. Reserved

63. `Instruction expected before register name`
64. `Public symbol already defined`
65. `Local symbol already defined`

66. `Number not expected`	
67. `New symbol must follow ➥PUBLIC keyword`	
68. `Label, Command, Instruction, ➥or Storage expected`	Can't understand what's on the line!
69. `Inconsistent redeclaration`	This label or variable is defined differently elsewhere. Usually the message is caused by an extern not being the same as the public when it's found.

Your Own Assembler?

If you're not using the Intel processors, this assembler won't do you much good, I'm afraid. On the other hand, you don't need to write one from scratch anyway. Cross-development is always an option for you. Cross-development is developing programs with tools and compilers under one operating system, targeted to run under a different operating system. This is really what you are doing with the DOS versions of the CM32 compiler and DASM.

If you want to build a complete environment as I did, you will have to think about an assembler, a compiler, and all the utilities to go with them.

One final note about the DASM assembler: The documentation was almost as difficult to write as the assembler. I recommend that you keep that in mind.

29

CM32: A 32-Bit C Compiler

CM stands for C-Minus. CM32 was written specifically to use while I was building the MMURTL computer operating system. Every change and fix that was made to the compiler, and some of the nonstandard extensions, are there specifically to support the operating system. I will not apologize for its incomplete state, because my real goal was to build an operating system. CM32 was a necessary detour.



As the name implies, this version of the C language is missing some pieces. CM32 started life as an early version of Dave Dunfield's Micro-C. I liked Mr. Dunfield's style, and I learned a lot from his code. The compiler is not very portable now, but portability wasn't my goal. He has given permission to include the source code on the CD-ROM. Permission is required from the authors as listed in the source code in order to use the compiler for any commercial purposes.

Mr. Dunfield also let me include his introductory document on the C language (CIntro.doc in the \Dunfield directory on the CD-ROM). If you are not familiar with the C programming language, CIntro.doc is a very good introduction. It is geared to a subset of the C language, but CM32 is also a subset, so it should be valuable if you need to learn or brush-up on C. All of Mr. Dunfield's demo products are also included in the \Dunfield directory on CD-ROM. If you, or someone you know, does embedded systems work on 8- and 16-bit processors, you should take a look at these products. There are complete working tools included, along with a catalog of all his other goodies.

The extentions I made to Micro-C all follow the guidelines presented in *The C Programming Language, Second Edition (ANSI C)*, by Brian W. Kernighan and Dennis M. Ritchie. You don't need this book to use CM32, but it was my guide for modifications to the compiler. After all, Kernighan and Ritchie (K&R) invented the language. There is a small amount of irony here, however, as CM32 doesn't support the original K&R style of function declarations. ANSI function prototypes are required.

To write an operating system with a complete development environment, you have to start somewhere. I started with the Microsoft 5.1 Assembler, eventually gravitated to the Borland Turbo Assembler, still using MASM conventions, and eventually ended up using DASM, our own assembler. All the MMURTL kernel and hardware handling code is done in hand-coded assembler. Once the kernel was done, and I started on the device drivers, I knew I would need a 32-bit compiler to cut down development time on the rest of the software. Several companies make excellent 32-bit compilers, but I also wanted to eventually port the compiler to the MMURTL environment. Quite frankly, the cost of the source code would be exorbitant, so I took a user-supported 16-bit compiler that generated assembler and worked from there. Talk about re-inventing the wheel. This set the MMURTL development effort back about four months, but what the heck. I certainly learned a whole lot.

CM32 produces assembly language that is compatible with DASM, the assembler in the CD-ROM.

Memory Usage and Addressing

The code and library source are set up for the MMURTL OS. MMURTL's memory model allows 2GB segment addressing with 32-bit flat pointers. For compatibility with MS-DOS programming tools and source code, I went with the following conventions in CM32:

```
char    = 8 Bits  (Byte)
short   = 16 Bits (word)
int     = 16 Bits (word)
long    = 32 Bits (dword)
```

I did this because of the wealth of source code available for MS-DOS. There is also a wealth of code available for UNIX, and in most UNIX compilers, an int is 32 bits. But far too much of it is so heavily UNIX specific, that rewriting it would be almost as fast as porting some of it.

Important Internal Differences

As you may know, C began on a machine with a flat memory architecture. The Intel 32-bit processors are built on a segmented architecture even though you can choose to ignore the segmentation. As described in the previous chapters, MMURTL pretty much ignores the segmentation capabilities, with one exception. The code and data segments are different even though they share the same linear address space in the operating system. This is why CM32 has support for 48-bit far calls, 16-bit selector, and 32-bit offsets.

Something that has given the Intel world a fit is C stack handling, especially Pascal and PLM-86 users. The Intel 32-bit processors have certain instructions to make stack operations easier, as well as making function-handling almost a pleasure. One such instruction is the `RET XX`, where *XX* is the number of bytes to remove from the stack prior to returning from the call. The UNIX/C convention has always been to have the caller remove his parameters (args) from the stack after the return. This allows for variable length argument lists. I didn't need this in MMURTL, as all the parameter lists to operating system calls are fixed length. This is a waste of good time-saving instructions. The MMURTL OS uses the `RET XX`, and removes the bytes before returning from all calls, so that's how I wrote the compiler.

Variable-Length Argument Lists

CM32 does allow you to prototype a function with an ellipsis (...), in which case CM32 will build the calling function to remove the arguments. The included stdarg.h, along with the supporting library code, handles variable length arguments. Please follow the conventions for variable arguments documented in *The C Programming Language, Second Edition.* Details can be found in the in stdarg.h file with the included library source code.

Some early C compilers were a little goofy when it came to retrieving parameters from the stack. They pop, pop, pop, until they get the parameter they want, or pop CX forever to remove them on return. Intel gave us a very clean way to get at stack parameters, and I use it. It's the frame pointer—EBP. Other C compilers also save certain registers before the call, and restore them after. CM32 saves none.

One last stack-related point, and probably the most notorious, is the order in which parameters are pushed on the stack. The C convention has been from right to left. I opted for left to right. This is called the PLM, or Pascal calling convention. Remember, if you use C in the recommended portable fashion, none of this will bother or concern you, we just thought you'd like to know. Of course, if you intend to add additional library support for CM32, you will have to know and understand it all.

CM32 Language Specifics

The CM32 compiler is a subset of the full ANSI-C language. It has several extentions, as well as short-comings. These are documented in the following sections.

Far Calls

One important extension CM32 has made to the C language is for the far function calls. All C compilers that work on Intel platforms usually have some type of far extensions.

CM32 supports 48-bit *far* calls, and `RETF xx` instructions for local functions designated as far, that will be called from outside your current segment. See the following section, Far External Functions, for specifics on how to use the far storage class modifier with functions.

Because MMURTL uses a separate virtual memory space for each program, we don't need far data pointers and don't support them. The only reason far calls are supported is to allow access to the MMURTL OS through 386/486 call gates.

Interrupt Functions

The interrupt class is used to designate the function as an interrupt service routine. It changes the standard entry and exit code for a function, producing the following:

```
PUSHAD
    ;Your code
POPAD
IRETD
```

Needless to say, don't call an interrupt function from your program. The results would be rather nasty. The `PUSHAD` and `POPAD` instructions save all registers and flags and restore them.

Supported C Language Features

The following C statements are supported by CM-32:

```
if/else
while
do/while
for
break
continue
return
goto
switch/case/default
{}
;
```

The following operators are supported:

```
+  +=  ++
-  -=  --
*  *=
/  /=
%  %=
^   ^=
~
&  &=
|  |=
&&  ||
<<  <<=  >>  >>=
>   >=  <  <=
=  ==
!  !=
?: (tertiary operators)
,  (comma)
()
[] (array indexing)
. ->  (structure field access operators)
sizeof()
*  for Indirection up to 7 levels
&  for "ADDRESS OF" including structures
```

The following data types and modifiers are supported:

```
struct (all members are packed)
char   (signed and unsigned)   8  bits
short  (signed and unsigned)   16 bits
int    (signed and unsigned)   16 bits
long   (signed and unsigned)   32 bits
arrays  (single and multidimensional)
pointers (to all types including pointers)
void
```

The following storage classes, or modifiers to classes, are supported:

```
signed   (the default as per ANSI)
unsigned
extern
```

```
static
const
register
far       (for functions only)
interrupt (for functions only)
```

The following character constants are supported. Decimal, octal, and hex constants are also supported as character constants in expressions (127, 0177, 0x7f, `\n`, or `Some Text\t`, for example):

```
\n      Newline
\r      Carriage return
\t      Tab
\b      Backspace
\f      Formfeed
\v      Vertical Tab
\177    Octal constants
\x7f    Hexadecimal constants
\0      Null
```

The following preprocessor commands are supported:

```
#define       (fully parameterized)
#undef
#include      (nested to 5 levels)
#ifdef
#ifndef
#else
#endif
#asm/#endasm  (for inline assembly language)
#pragma       (properly ignored)
```

In-line assembly code is supported with the `#asm` operator as defined above. It is simply placed line-for-line into the generated assembly language output file.

Library Header File Access

The standard include files which are listed inside angle brackets (<stdio.h> for example) should be located in the \CM32\INCLUDE directory on your current disk.

Quoted includes ("MyHeader.h" for example) should specify the full path if the file is not located in the current directory.

Limitations as Compared to ANSI

The following items are not supported:

Real numbers

unions

enumerated types

bit fields

type casts

`#if` (with separate defined)

`#elif`

Structure Limitations

Structures are supported including tags. Structure arrays are also supported, but the following limitations apply:

Structures cannot be passed as arguments to functions.

They cannot be returned from functions.

They cannot be assigned to each other.

Structures cannot be nested.

The use of pointers to structures gets around all of these problems except the lack of nested structures.

The length of a structure is exactly the sum of each member's size.

The register class modifier doesn't do a lot, but it's recognized for compatibility. Automatic variables, variables local to functions, *cannot* be initialized.

Far External Functions

The far type modifier can be quite confusing to someone that has always worked on a machine with a flat memory architecture. If you are one of those people, the best analogy of *far* is that you are actually working on several machines, each with their own large flat memory space. Any time you want to access a function to some other machine's memory, you need the far modifier.

For example, if you are calling a function that is not in your code segment in your machine's memory space if you will, you will have to tell the compiler that it is a far function.

Far functions are those that you must call in another segment, or those in your segment that must be called from outside your segment. For example, the function `GetMoney`, a fictitious but useful system call that resides outside your current code segment, would have to be defined as being far in its function prototype in order for the compiler to generate a *far* call in assembler. Example follows of a prototype for a far call outside your segment:

```
extern long far GetMoney(void);

/* a useful but ficticious function */
```

Far functions *cannot* return far pointers. This is an intentional limitation of CM32 compiler. In fact, the only reason CM32 supports far calls is because MMURTL calls are made through *call gates* which require a unique selector to identify each call.

This means that if you defined `GetMoney` as:

```
extern far long GetMoney(void);
```

the far modifier still applies to the function and not the long returned variable.

Type Conversion and Pointers

As mentioned above, CM32 does not support type casts. Consequently, type checking is relaxed with pointer assignments. Also, type conversions are automatic, and follow these rules, which are ANSI compatible:

8 bit var = 16 bit var—The low-order eight bits of the 16-bit variable will be placed in the 8-bit variable.

16 bit var = 32 bit var—The low order 16 bits of the 32-bit variable will be placed in the 16-bit variable.

32 bit var = 16 bit var and 16 bit var = 8 bit var—The value is properly sign or zero extended before assignment.

unsigned var = signed var—Signed variable is sign or zero extended if required, and placed into the unsigned variable. A char with the value of -1 assigned to an unsigned long will result in 0xFFFFFFFF as you would expect. Through the conversion, -1 equals -1.

All pointers are 32 bit and may be assigned, manipulated, and compared as unsigned long integers.

Initialization of Static Variables

CM32 supports bracketed initialization with braces ({}), as specified by ANSI. I have seen some compilers that simply can't seem to get it right. The following examples show my implementation so there should be no confusion. If I'm doing it wrong, at least you'll how I do it.

Examples of initialization ANSI style as defined in Kernighan and Ritchie's book.

The following example defines a 3×5 array of characters:

```
char c[][5] = { {"abc"}, {"def"}, {"ghi"} };
```

Stored in memory, it looks like this (15 bytes):

'a' 'b' 'c' 0 0
'd' 'e' 'f' 0 0
'g' 'h' 'i' 0 0

The next example defines a 3×1 array of integers.

```
int x[] = {1,3,5};
```

Stored in memory, it looks like this (three integers) :

1 3 5

Both of the following examples define a 4×3 array of integers. Note the inside braces are optional, but serve a purpose if you can't define all the elements in each sub-array:

```
int y1[4][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};
```

```
int y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
    };
```

Both look like this stored in memory (12 integers) :

1 3 5

2 4 6

3 5 7

0 0 0 (Last index zeroed automatically)

The following example is also a 4×3 array, but only the first value of each of the first indexes is initialized. This is where the inside braces are required:

```
int y2[4][3] = { {1}, {2}, {3}, {4} };
```

Stored in memory, it looks like this (three integers):

1 0 0

2 0 0

3 0 0

4 0 0

Both of the following examples are 3×7 character arrays. They can be defined either way. The first dimension of the array is determined by how many initializers there are:

```
char rgReserved[][7]  = {
        {"THE"},
        {"BIG"},
        {"DOGS"},
        };

char rgTest[][7]  = {
        "THE",
        "BIG",
        "DOGS",
        };
```

In memory, each of the strings is zero, filled out to the length specified in the second dimension:

'T' 'H' 'E' 0 0 0 0
'B' 'I' 'G' 0 0 0 0
'D' 'O' 'G' 'S' 0 0 0

The following is a 3×3 array of pointers. They point to literal strings that are stored in another location in memory:

```
char *paTest[3][3]  = {
       {"ALIGN", "TEST",},
       {"BYTE"},
       {"CODE"},
       };
```

In memory, this declaration stores an array of pointers, and it stores the strings that they point to. The undefined pointers in the array will be 0 (Null). This is how it looks in memory, where `paTest[x]` are points to the null-terminated strings stored in memory:

p1 p2 0
p3 0 0
p4 0 0

The next example stores a string. Note that with single character arrays, braces ({ }) are not needed. The characters are stored in memory terminated with a null. The dimension (size) of the string is determined by how many characters there are in the string. One null is added to the length.

```
char testc[] = "This is a test! ";
```

The following is also stored in memory as an array of characters, except the string is zero padded out to the length specified in the index, which is 17. It is null terminated and the null is included in its length.

```
char testc1[17] = "This is a test! ";
```

Other Implementation-Dependent Information

The shift right function zero fills on all operations.

Structures are packed. There are no fills for word, or double word, alignment. This means you can use a memory copy function to make copies of structures. You may not assign structures, and they may not be passed as parameters.

Using CM32

To compile a C program with CM32 you enter the name of the source file first, and optionally, the name of the destination ASM file next, then any command line options (switches), for example:

```
CM32 MyFile.c MyFile.asm /L /S
```

The compiler will then process the file and produce the assembly language file you specified. If you left out the destination assembly file name, it will use the source file name with .ASM as the output file.

Errors will be sent to `stdout`, which is the screen, unless you use the `/L` switch which directs all errors to a list file named SourceFile.LST.

Command Line Options

CM32 has the following command line options available. They are not case sensitive (`/s` = `/S` for example):

`/E` (Embedded source mode) This will embed the C source code in the assembly language output. The data declarations from your source will all be packed after the ASM data declarations, while each line of code precedes the assembly language statements that it translates into. Using this option helps to see how the compiler handles things, which is good for additional hand optimization.

`/G` (Generate separate files) This tells the compiler to generate a separate code and data assembly language file as the output. This is useful if you are including the compiler output into other assembly language files. Output of all segment definition and all assume statements is suppressed with this option. The files are named .DAS and .CAS (data and code, respectively).

`/L` (List file) This tells the compiler to direct all errors to a file. The filename is the same as your source file, except the file extension is .LST.

`/N` (No Optimization) This tells the compiler to skip the optimizer phase of the compilation. I recommend you always optimize. This optimization is for both speed and size and to be honest, the generated code stinks without it. It is just barely acceptable with it.

`/S` (Suppress herald) This suppresses the compiler name and version herald that is displayed when the compiler is first executed. If you use the `/L` option, the only text displayed on the standard output device, the screen, will be fatal compiler errors.

`/W` (Warnings ON) This causes the compiler to output warnings for certain practices such as variables that are never referenced, and type conversions that look odd.

CM32 searches the \CM32\INCLUDE directory on your current disk path for standard `#include <xxx.h>` files.

Library Functions

CM32 is limited in the library functions that are provided. The implementation is closer to a free-standing version than a hosted version. You can add library functions if you like. The supported functions and macros are listed under each header file in Listing 29.1.

Listing 29.1. Supported library functions.

```
<stdio.h>

extern FILE *fopen(char *name, char *mode);
extern long *fclose(FILE *stream);
extern long fgetc(FILE *stream);
extern char *fgets(char *s, long n, FILE *stream);
extern long fputc(long c, FILE *stream);
extern long fputs(const char *s, FILE *stream);
extern long printf(char *fmt, ...);
extern long sprintf(char *s, char *fmt, ...);
extern long fprintf(FILE *stream, char *fmt, ...);
extern long fread(void pData, size_t
   objsize, site_t nobj, FILE *stream);
extern long fwrite(void pData, size_t
   objsize, site_t nobj, FILE *stream);

<stdlib.h>

extern exit(long signal);

<ctype.h>

extern long iscntrl(long c);
extern long isspace(long c);
extern long isdigit(long c);
extern long isupper(long c);
extern long islower(long c);
extern long ispunct(long c);
extern long isalpha(long c);
extern long isxdigit(long c);
extern long isalnum(long c);
extern long isgraph(long c);
extern long toupper(long c);
extern long tolower(long c);

<string.h>

extern char *strcpy(char *s, const char *ct);
extern char *strncpy(char *s, const char *ct, long n);
extern char *strcat(char *s, const char *ct);
extern char *strncat(char *s, const char *ct, long n);
extern long strcmp(const char *cs, const char *ct);
extern long strncmp(const char *cs, const char *ct, long n);
```

```
extern long strlen(char *cs);
extern char *strchr(const char *cs, int *c, long n);

<stdarg.h>   (The following are macros in stdarg.h )

va_list unsigned long *
va_start(ap,arg)
va_arg(ap,type)
va_end(ap)

<stdlib.h>

extern void exit(long error);
extern void stdentry(void);
```

Library Code Examples

Two samples of the library code are shown below to demonstrate the interface with the MMURTL operating system.

Fileio.c

Fileio.c implements many of the file calls in stdio.h. It is a good example of the proper use of the MMURTL file system service. See Listing 29.2.

Listing 29.2. Standard file I/O for MMURTL.

```
/*  fileio.c

#include <string.h>
#include <stdio.h>

#include "C:\OSSource\MFiles.h"
#define ErcDupName  226     /* Name exists as a file or dir already */

static long error_num = 0;

/************** fopen ***************
 opens a file in stream mode for reading,
 or reading and writing (modify mode).
************************************/

FILE *fopen(char *name, char *mode)
{
long handle, lmode, fcreate, fdiscard, erc, fappend;

    erc = 0;
    handle = 0;
    fappend = 0;
    fcreate = 0;
    fdiscard = 0;
```

continues

Listing 29.2. continued

```
    if (*mode=='r')
        lmode = 0;
    else if (*mode=='w')
    {
        lmode = 1;
        fcreate = 1;
        fdiscard = 1;
    }
    else if (*mode=='a') {
        fappend = 1;
        fcreate = 1;
        lmode = 1;
    }
    else
        return (0);

    /* see if they want to update also */

    if ((mode[1] == '+') || (mode[2] == '+'))
        lmode = 1;

    if (fcreate)        /* try to create with archive bit set */
        erc =  CreateFile(name, strlen(name), 0x20);

    if (erc == ErcDupName)
        erc = 0;

    if (!erc)
        erc =  OpenFile(name, strlen(name), lmode, 1, &handle);
    if (erc)
    {
        error_num = erc;
        handle = 0;
    }
    if ((!erc) && (fappend))
    {
        erc = SetFileLFA(handle, 0xffffffff);       /* EOF */
    }
    if ((!erc) && (fdiscard))
    {
        erc = SetFileSize(handle, 0);       /*  */
    }

    return (handle);
}

/*************** rename ***************/

int rename(char *oldname, char *newname)
{
long  erc;

    erc =  RenameFile(oldname, strlen(oldname), newname, strlen(newname));
    if (erc) {
        error_num = erc;
        return(-1);
    }
```

```
    return(0);
}

/*************** remove ***************/

int remove(char *name)
{
long handle, erc;

    erc =  OpenFile(name, strlen(name), 1, 0, &handle);
    if (erc)
    {
        error_num = erc;
        return(-1);
    }
    erc =  DeleteFile(handle);
    if (erc)
    {
        error_num = erc;
        CloseFile(handle);
        return(-1);
    }
    return(0);
}

/*************** fclose ***************/

long fclose(FILE *stream)
{
long erc, i;
erc = CloseFile(stream);
if (erc) {
    error_num = erc;
    i = EOF;
    }
else
    i = 0;
return stream;
}

/********** fgetc ***************************************
*  Read a char and return from function. Return EOF (-1)
*  If EOF. This returns char as an int.
************************************************************/

long fgetc(FILE *stream)
{
long erc, i, chl;
unsigned char ch;

    erc = ReadBytes (stream, &ch, 1, &i);
    if (erc)
        return EOF;
    else
        chl = ch;
    return (chl);
}
```

continues

Listing 29.2. continued

```
/******** fgets ***************************************
  Read chars into string (s) until we get a LF or n-1=0.
  Null terminate the string.
*****************************************************/

char *fgets(char *s, long n, FILE *stream)
{
long  ch;
char  *ss, c;
    ss = s;
    while (n > 1)
    {
        ch = fgetc(stream);
        if (ch == EOF)
        {
            *s = 0;      /* null terminate */
            return(0);
        }
        else
        {
            c = ch;
            *s = c;
            n--;
            if (c == 0x0A)
                n = 0;
            s++;
        }
    }
    *s = 0;
    return (ss);
}


/*************** fputc ***************/

/* writes the char (int) in c to the stream.
   No translation is done to the char.
*/

long fputc(long c, FILE *stream)
{
long erc, i, ch;
    erc = WriteBytes (stream, &c, 1, &i);
    if (erc)
        ch = EOF;
    else
        ch = c;
    return (ch);
}

/******************** fputs *************************
 Writes the string pointed to by *s to file *stream
 No EOL translation is done. The terminating NULL is
 not placed in the file.
*****************************************************/
```

```
long fputs(const char *s, FILE *stream)
{
long erc, i, ch;

    erc = 0;
    while ((*s) && (!erc))
    {
        erc = WriteBytes (stream, s++, 1, &i);
    }
    if (erc)
        ch = EOF;
    else
        ch = 0;
    return (ch);
}

/************** ftell ****************************/
/* returns the file pointer (Logical File Address)  */
/* of stream.                                       */

long ftell(FILE *stream)
{
long erc, i;
    erc = GetFileLFA(stream, &i);
    if (erc)
        i = EOF;
    return (i);
}

/************** rewind ***************************/
/* Set file LFA to 0.                               */

void rewind(FILE *stream)
{
long erc;
    erc = SetFileLFA(stream, 0);
}

/************** fseek ****************************/
/* Set file LFA to desired position.               */

void fseek(FILE *stream, long offset, long origin)
{
long erc, crnt;

    if (origin==SEEK_CUR)
        erc = GetFileLFA(stream, &crnt);
    else if (origin==SEEK_END) {
        offset = 0;
        crnt = -1;  /* MMURTL Seeks to EOF */
    }
    else if (origin==SEEK_SET)
        crnt = 0;
    else
        return 1;
```

continues

Listing 29.2. continued

```
    erc = SetFileLFA(stream, offset+crnt);
    if (erc)
        return (1);
    else
        return (0);
}

/********** fread ***************************************
*  reads nobjects of nsize from stream and returns them
*  to ptr.
*******************************************************/

long fread(char *ptr, long size, long nobj, FILE *stream)
{
long erc, i, j;

    erc = ReadBytes (stream, ptr, size*nobj, &i);
    if (erc < 2)
        j = i/size;     /* number of objects of size */
    else
        j = 0;          /* nothing! */
    return (j);
}

/********** fwrite **************************************
*  writes nobjects of nsize to stream from ptr.
*******************************************************/

long fwrite(char *ptr, long size, long nobj, FILE *stream)
{
long erc, i, j;

    erc = WriteBytes (stream, ptr, size*nobj, &i);
    if (!erc)
        j = i/size;     /* number of objects of size */
    else {
        error_num = erc;
        j = 0;          /* nothing! */
    }
    return (j);
}

/****************** End of FileIO.c *************/
```

string.c

String.c implements the supported string functions. Listing 29.3 provides a good example of how to use an assembly language interface with MMURTL.

Listing 29.3. Standard string functions for MMURTL.

```
/**************************************
* The most commonly used string functions
* are contained in this libarary source file.
* strcmp  strncmp  strcpy  strncpy, strlen,
* strncat, strchr
*/

long strcmp(char *str1, char *str2)
{
;
#asm
    MOV ESI,[EBP+12]
    MOV EDI,[EBP+8]
strcmp0:
    MOV AL,[ESI]
    CMP AL,BYTE PTR [EDI]
    JG strcmp1
2    JL strcmp2
    CMP AL, 0
    JE strcmp3
    INC DI
    INC SI
    JMP SHORT strcmp0
strcmp1:
    MOV EAX, 1
    JMP SHORT strcmp4
strcmp2:
    MOV EAX, -1
    JMP SHORT strcmp4
strcmp3:
    XOR EAX,EAX
strcmp4:

#endasm
}

/* same as strcmp except only compares up
   to set length */

long strncmp(char *str1, char *str2, long n)
{
;
#asm
    MOV ESI,[EBP+16]    ;str1
    MOV EDI,[EBP+12]    ;str2
    MOV ECX,[EBP+8]     ;n
strncmp0:
    CMP ECX, 0          ;Equal so far?
    JE strncmp3         ;Yes, we're done
    DEC ECX             ;One less
    MOV AL,[ESI]        ;
    CMP AL,BYTE PTR [EDI]
    JG strncmp1
    JL strncmp2
    CMP AL, 0           ;End of String?
```

continues

Listing 29.3. continued

```
    JE strncmp3         ;Equal up to here!
    INC EDI             ;Next chars
    INC ESI
    JMP SHORT strncmp0  ;Back again
strncmp1:
    MOV EAX, 1
    JMP SHORT strncmp4
strncmp2:
    MOV EAX, -1
    JMP SHORT strncmp4
strncmp3:
    XOR EAX,EAX
strncmp4:

#endasm
}

/* strcpy - String Copy ct to s */
/* including null, return s     */

char *strcpy(char *s, char *ct)
{
;
#asm
    MOV EDI,[EBP+12]        ;destination s
    MOV ESI,[EBP+8]         ;source ct
    CLD
strcpy0:
    MOVSB
    CMP BYTE PTR [ESI], 0
    JNZ strcpy0
    MOV EAX,[EBP+12]        ;return s
#endasm
}

/* strncpy - String Copy ct to s */
/* upto max of n chars, pad with */
/* nulls if s < ct, return s     */

char *strncpy(char *s, char *ct, long n)
{
;
#asm
    MOV EDI,[EBP+16]        ;destination *s
    MOV ESI,[EBP+12]        ;source *ct
    MOV ECX,[EBP+8]         ;max mov  n
    CLD
strncpy0:
    CMP ECX, 0              ;End yet??
    JE strncpy2             ;Yes
    DEC ECX                 ;One less
    MOVSB                   ;Move it
    CMP BYTE PTR [ESI], 0   ;End of Source?
    JNZ strncpy0            ;No, go back
strncpy1:
    CMP ECX,0
```

```
    JE strncpy2
    INC EDI
    MOV BYTE PTR [EDI], 0
    DEC ECX
    JMP SHORT strncpy1
strncpy2:
    MOV EAX,[EBP+16]

#endasm
}

long strlen(char *cs)
{
;
#asm
    XOR EAX, EAX
    MOV ESI,[EBP+8]
_strlen0:
    CMP BYTE PTR [ESI],0
    JE _strlen1
    INC ESI
    INC EAX
    JMP SHORT _strlen0
_strlen1:
#endasm
}

char *strncat(char *s, char *ct, long n)
{
;
#asm
    MOV EDI,[EBP+16]        ;destination *s
    MOV ESI,[EBP+12]        ;source *ct
    MOV ECX,[EBP+8]         ;max mov  n
    CLD
strncat00:                  ;get to end of s
    CMP BYTE PTR [EDI], 0
    JE strncat0
    INC EDI
    JMP SHORT strncat00
strncat0:
    CMP ECX, 0              ;End yet??
    JE strncat2             ;Yes
    DEC ECX                 ;One less
    MOVSB                   ;Move it
    CMP BYTE PTR [ESI], 0   ;End of Source?
    JNZ strncat0            ;No, go back
strncat1:
    CMP ECX,0
    JE strncat2
    INC EDI
    MOV BYTE PTR [EDI], 0
    DEC ECX
    JMP SHORT strncat1
strncat2:
    MOV EAX,[EBP+16]        ;Return s
```

continues

Listing 29.3. continued

```
#endasm
}

char *strcat(char *s, char *ct)
{
;
#asm
    MOV EDI,[EBP+12]    ;destination *s
    MOV ESI,[EBP+8]     ;source *ct
    CLD
strcat0:                    ;get to end of s
    CMP BYTE PTR [EDI], 0
    JE strcat1
    INC EDI
    JMP SHORT strcat0
strcat1:
    MOVSB                   ;Move it
    CMP BYTE PTR [ESI], 0   ;End of Source?
    JNZ strcat1             ;No, go back for more
    MOVSB                   ;Yes, Null terminte
    MOV EAX,[EBP+12]        ;Return s
#endasm
}

/***************** strchr ******************/
/* Find first occurence of 'chr' in 'string' */

char *strchr(char *string, char chr)
{
    do
        if(*string == chr)
            return string;
    while(*string++);
    return 0;
}
```

I

Index

Symbols

32-bit software, 130-131
386 microprocessors, 6
DASM assembler instruction set, 682-688
486 microprocessors, 6

A

abstraction (hardware), 82
accessing devices (file system), 206
active job (monitor), 160
AddAliasRun function, 388-390
AddCallGate( ) call, 227-228
AddGDTCallGate( ) call, 394-395
AddIDTGate( ) call, 228-229, 395-396
AddOSPT function, 392-394
addresses
CM32, 699
DASM assembler, 673-675
files (logical), 204
memory, aliasing, 78
AddRun function, 387-388
AddUserPT function, 390-392
Alarm( ) call, 229
Alarm( ) function, 151, 406
timer, 411-412
aliases (memory addresses), 78
AliasMem( ) call, 229-230, 397-400
allocating memory, 63
linear, 76
linked list memory allocation, 65-66
listing 5.3 returning memory to free pool, 67-68
physical, 77
AllocDMAMem( ) function, 158
AllocDMAPage( ) call, 230
AllocDMAPage( ) function, 76
AllocExch( ) call, 230
AllocExch( ) function, 45, 135, 370-371
AllocOSPage( ) call, 230-231, 396-397
AllocOSPage( ) function, 76
AllocPage( ) call, 231
AllocPage( ) function, 76, 134
alphanumeric key values (keyboard), 193
ANSI (CM32), 702-703
API (Application Programming Interface), 8, 116-119
documenting, 116
procedural interfaces, 116-120
calling, 117-118
error handling, 119
hardware, 118-119
portability, 119
systems programming, 120-125
device drivers, 120-125
internal calls, 120
applications
chaining, 140
CLI (Command-Line Interpreter), 266-288
external commands, 270-271
hot keys, 271
internal commands, 266-270
program listing, 271-288
debugger, 185
DumbTerm, 314-317
editor, 288-314
exiting, 140
Job Control Block, 136-137
messaging, 135
arguments (CM32), 699-700
assemblers, 16, 16-17
DASM, 331, 664-668
386 instruction set, 682-688
addressing, 673-675
CALL instruction, 679-681
command line options, 667-668
DOT commands, 670-672
error codes, 693-696
EXTERN declarations, 677-678
JUMP instruction, 679-681

local declarations, 668-669
memory, 669
prefixes, 678-679
PUBLIC declarations, 677
storage, 675-676
DOS, 672-673
Assign Keyboard( ) function, 192
asynchronous system services, 146

B

Beep( ) call, 231
timer, 417-418
Beep_Work( ) function (timer), 416-417
Block mode (files), 203-204
block-oriented devices (DMA), 85-86
boot sector (File Allocation Table), 593
booting, 100-111
from different operating system, 111
ROM (Read Only Memory), 100
sectors, 101-110
buffers (keyboard), 557
buses, 83-84

C

C language (CM32), 699
character constants, 702
preprocessor commands, 702
statements, 701-702
storage classes, 701-702
CALL instruction (DASM assembler), 679-681
caller information (request blocks), 145-150
calling
device drivers, 149, 154-156
internal, systems programming, 120
operating systems, 131-132
stacks, 132-134
source code, 328
calling conventions (procedural interfaces), 117-118
calls
AddCallGate( ), 227-228
AddIDTGate( ), 228-229
Alarm( ), 229
AliasMem( ), 229-230
AllocDMAPage( ), 230
AllocExch( ), 230
AllocOSPage( ), 230-231
AllocPage( ), 231
Beep( ), 231
Chain( ), 231
CheckMsg( ), 232
ClrScr( ), 232
Compare( ), 233
CompareNCS( ), 233
CopyData( ), 233
CopyDataR( ), 233-234
DeAliasMem( ), 234
DeAllocExch( ), 234
DeAllocPage( ), 234
DeviceInit( ), 235
DeviceOp( ), 235-236
DeviceStat( ), 236
DMASetUp( ), 236-237
EditLine( ), 237
EndOfIRQ ( ), 237
ExitJob( ), 238
far calls (CM32), 700
FillData( ), 238
GetCmdLine( ), 238-239
GetCMOSDate( ), 239
GetCMOSTime( ), 239
GetDMACount( ), 240
GetExitJob( ), 240
GetIRQVector( ), 240-241
GetJobNum( ), 241
GetNormVid( ), 241
GetPath( ), 241-242
GetPhyAdd( ), 242
GetpJCB( ), 242
GetSysIn( ), 242, 243
GetSysOut( ), 243
GetTimerTick( ), 243
GetTSSExch( ), 243
GetVidChar( ), 244, 581
GetVidOwner( ), 244
GetXY( ), 244
HardXY( ), 584
InByte( ), 244
InDWord( ), 245
InitDevDr( ), 245-246
InWord( ), 245
InWords( ), 245
ISendMsg ( ), 246-247
KillAlarm ( ), 247
LoadNewJob( ), 247
MaskIRQ ( ), 248
MicroDelay( ), 248
MoveRequest( ), 248-249
NewTask( ), 249

OutByte( ), 249
OutDWord( ), 249
OutWord( ), 250
OutWords( ), 250
public, 222
device drivers, 225
I/O, 226
interrupts, 226-227
jobs, 224-225
keyboard, 227
memory, 225-226, 394-404
messaging, 224
moving data, 226
parameters, 222-223
speakers, 227
stacks, 224
system service, 225
tasks, 225
timers, 226
video, 227
PutVidAttrs( ), 250, 580
PutVidChars( ), 250-251, 580
QueryPages( ), 251
ReadCMOS( ), 251
ReadKbd ( ), 251-252
RegisterSvc( ), 252
Request( ), 252-253
Respond( ), 254
ScrollVid( ), 254
see also functions
SendMsg ( ), 254-255
SetCmdLine( ), 255
SetExitJob( ), 255-256
SetIRQVector( ), 256-257
SetJobName( ), 257
SetNormVid( ), 257-258
SetPath( ), 258
SetPriority( ), 258
SetSysIn( ), 258-259
SetSysOut( ), 259
SetUserName( ), 259
SetVidOwner( ), 259-260, 574
SetXY( ), 260
Sleep( ), 260
SpawnTask( ), 261
Tone( ), 261-262
TTYOut ( ), 262
UnMaskIQR ( ), 262-263
UnRegisterSvc( ), 263
WaitMsg( ), 263
see also functions
Chain( ) call, 231
Chain( ) function, 140, 440, 448
chaining applications, 140
character constants (C language), 702
CheckMsg( ) call, 232
CheckMsg( ) function, 46, 135, 363-366
ChkRdyQ function, 341-342
classes (C language), 701-702
CLI (Command-Line Interpreter), 266-288
external commands, 270-271
hot keys, 271
internal commands, 266-270
Cls (Clear Screen), 267
Copy, 267
Debug, 268
Del (Deleting files), 268
Dir (Directory Listing), 267-268
Exit, 268
Help, 268
MD (Make Directory), 269
Monitor, 268
Path (Set File-Access Path), 269
RD (Remove Directory), 269
Rename (files), 270
Run, 270
program listing, 271-288
client/server, 7
CloseFile( ) function, 207
closing files, 220
ClrScr( ) call, 232
ClrScr( ) function, 575
Cls (Clear Screen command), 267
CM32 (C-Minus), 698
addresses, 699
ANSI, 702-703
argument lists, 699-700
C language, 699
character constants, 702
preprocessor commands, 702
statements, 701-702
storage classes, 701-702
command line options, 707-708
compiler, 331
compiling, 707-718
far calls, 700

far external functions, 703-704
interrupts, 700
library functions, 708-709
library header file access, 702
memory, 699
pointers, 704
program listings
I/O file for MMURTL, 709-714
library functions, 708-709
standard string functions, 715-718
shift right function, 706
static variables
initializing, 704-706
string functions, 715
structures, 706
type conversion, 704
code
borrowing, 333
comments, 328
modular, 332
command lines
CM32, 707-708
DASM assembler, 667-668
device drivers, 668
DLLs (Dynamic Link Libraries), 668
error statements, 668
list files, 667-668
Command-Line Interpreter, ***see*** **CLI**
commands
C language (preprocessor), 702
CLI (Command-Line Interpreter)
external, 270-271
internal, 266-270
DOT (DASM assembler), 670-672
editor, 288-290
VIRTUAL, 422-429
COMMANDS.CLI file, 270-271
comments, 328
Compare( ) call, 233
CompareNCS( ) call, 233
compiling
CM32, 707-718
compiler, 331
DASM assembler, 331
MMURTL, 331-332
cooperative
multitasking, 31-32
queueing (scheduling tasks), 34
scheduling tasks, 34
Copy command, 267
CopyData( ) call, 233
CopyDataR( ) call, 233-234
CPU (Central Processing Unit), 82-83
bus structure, 83-84
tasks (time), 30-32
CreateDirectory( ) function, 215
CreateFile( ) function, 213
cursor
key codes, 198
positioning, 584-586

D

DASM assembler, 331, 664-668
386 instruction set, 682-688
addressing, 673-675
CALL instruction, 679-681
command line options, 667-668
device drivers, 668
DLLs (Dynamic Link Libraries), 668
error statements, 668
list files, 667-668
DOS assemblers, 672-673
DOT commands, 670-672
error codes, 693-696
EXTERN declarations, 677-678
JUMP instruction, 679-681
local declarations, 668-669
memory, 669
prefixes, 678-679
program listings
alternating data and code, 669-670
application template file (listing 28.1), 666
application template file (listing 28.2), 666-667
PUBLIC declarations, 677
storage, 675-676
DCB (Device Control Block), device drivers, 152-154

DD (Define Double Word), 676
DeAliasMem( ) call, 234, 400-401
deallocating linear memory, 76-77
DeAllocExch( ) call, 234
DeAllocExch( ) function, 135, 371-373
DeAllocMem( ) call, 401-403
DeAllocPage( ) call, 234
DeAllocPage( ) function, 134
Debug command, 268
debugger, 182-185
debugging applications, 185
display, 183
entering, 182
exiting, 182
function keys, 183-184
interrupt procedure, 472-473
keyboard, 565
processors, 185-188
program listings
debugger data and code, 474-496
exception handlers for entering debugger, 472-473
source code, 473-496
Debugger.ASM (MMURTL source file), 329
Del command (Deleting files), 268
DeleteDirectory( ) function, 216
DeleteFile( ) function, 214
demand-paged flat memory, 59
demand-paged memory, 61
demand-paged virtual memory, 61
deQueueMsg function, 338-339
deQueueRdy function, 340-341
deQueueTSS function, 339
DevDrvr.ASM (MMURTL source file), 330
device drivers
calling, 154-155
DASM assembler, 668
DCB (Device Control Block), 152-154
DeviceInit( ) function, 157
DeviceStat( ) function, 156
DMA, 151
errors, 158
IDE (Integrated Drive Electronics), 498-519
IDE device driver (listing 24.1), 498-499
IDE device driver (listing 24.2), 499-502
IDE device driver (listing 24.3), 502-504
IDE device driver code (listing 24.4), 504-508
IDE device driver code (listing 24.5), 508-515
IDE device driver code (listing 24.6), 515-519
initializing, 157-158
installing, 149-150
interrupts, 150
messaging features, 151
operating systems (functions), 158
public calls, 225
RS-232 asynchronous, 519-540
device driver (listing 24.10), 525
device driver (listing 24.11), 526
device driver (listing 24.12), 526-528
device driver (listing 24.13), 528-535
device driver (listing 24.14), 535-540
device driver (listing 24.8), 521-523
device driver (listing 24.9), 523-525
header file (listing 24.7), 519-520
system resources, 150-151
systems programming, 120-125
calling drivers, 124-125
calling operating system, 122-124
timers, 151
writing, 147-158
calling, 149
theory, 148
DeviceInit( ) call, 235
DeviceInit( ) function, 149, 154, 157
DeviceOp( ) call, 235-236

DeviceOp( ) function, 149, 154-155
devices, accessing with file system, 206
DeviceStat( ) call, 236
DeviceStat( ) function, 149, 154, 156
DF (Define Far Word), 676
Dir command (Directory Listing), 267-268
directories
MD command (Make Directory), 269
memory, paging 9PDs), 72-76
RD command (Remove Directory), 269
disks
FAT (File Allocation Table), 592-593
boot sector, 593
hidden, 592
initializing, 593
LBA (Logical Block Address), 592-593
physical sector numbers, 592
see also file system
display (debugger), 183
DLLs (Dynamic Link Libraries), 668
DMA (Direct Memory Access), 89-97
block-oriented devices, 85-86
device drivers, 151
DMACode.ASM (MMURTL source file), 330
DMASetUp( ) call, 236-237
DMASetUp( ) function, 151, 158
documenting API, 116
DOS assemblers, 672-673
DOT commands (DASM assembler), 670-672
drivers
device, writing, 147-158
systems programming
calling drivers, 124-125
calling operating system, 122-124
device, 120-125
see also device drivers
DumbTerm application, 324
DW (Define Word), 676
dynamic structures, 114
dynamic tables, 114

E

edit keys (key codes), 198
EditLine( ) call, 237
EditLine( ) function, 586
editor, 288-314
commands, 288-290
program listing, 290-314
screen, 288
EndOfIRQ( ) call, 237
EndOfIRQ( ) function, 150-151, 158
enQueueMsg function, 337-338
enQueueRdy function, 340
entering debugger, 182
entry point (operating system), 422-429
errors
DASM assemblers
codes, 693-696
command line options, 668
device drivers, 158
procedural interfaces (API), 119
system services, 146-147
Except.ASM (MMURTL source file), 331
executable files, 688-693
tagged fields, 689
Exit command, 268
exit( ) function, 140
exiting
applications, 140
debugger, 182
ExitJob( ) call, 238
ExitJob( ) function, 140, 440, 448
EXTERN declarations (DASM assembler), 677-678
external commands (CLI), 270-271

F

far calls (CM32), 700
far external functions (CM32), 703-704
FAT (File Allocation Table), 592-593
boot sector, 593
hidden, 592

initializing, 593
LBA (Logical Block Address), 592-593
physical sector numbers, 592
see also file system
FCB (File Control Blocks), 217-218
File Allocation Tables, *see* FAT
File Control Blocks (FCB), 217-218
file system, 593-660
MS-DOS FAT-compatible file system, 594-660
File User Blocks (FUB), 218
files
addresses (logical), 204
closing, 220
COMMANDS.CLI, 270-271
Del command (Delete), 268
devices, accessing, 206
executable, 688-693
tagged fields, 689
FAT (File Allocation Table), 592-593
boot sector, 593
hidden, 592
initializing, 593
LBA (Logical Block Address), 592-593
physical sector numbers, 592
see also file system
functions, 207-217
CloseFile( ), 207-208
CreateDirectory( ), 215
CreateFile( ), 213-214
DeleteDirectory( ), 216
DeleteFile( ), 214-215
GetDirSector( ), 216-217
GetFileLFA( ), 211
GetFileSize( ), 212
OpenFile( ), 207
ReadBlock( ), 208
ReadBytes( ), 209-210
RenameFile( ), 214
SetFileLFA( ), 211-212
SetFileSize( ), 213
WriteBlock( ), 209
WriteBytes( ), 210-211
handles, 203
MMURTL source code, 328-331
modes
Block, 203-204
open, 203
Stream, 204
names, 202
network specifications, 202-203
operations, 219-220
procedural interfaces, 206
reading, 220
system, 217-220
internal structure, 217-219
requests, 204-205
writing, 220
FillData( ) call, 238
FindHiPage function, 381
FindLoPage function, 382-383
FindRun function, 385-387
Floppy.C (MMURTL source file), 330
FUB (File User Blocks), 218
function keys
debugger, 183-184
key codes, 196-197
monitor, 161
functions
AddAliasRun, 388-390
AddOSPT, 392-394
AddRun, 387-388
AddUserPT, 390-392
Alarm( ), 151, 406
timer, 411-412
AllocDMAMem( ), 158
AllocDMAPage( ), 76
AllocExch( ), 45, 135
AllocOSPage( ), 76
AllocPage( ), 76, 134
Assign Keyboard( ), 192
Beep( ), 417-418
Beep_Work( ), 416-417
Chain( ), 140, 440, 448
CheckMsg( ), 46, 135
CloseFile( ), 207
ClrScr( ), 575
CM32
library, 708-709
string, 715
CreateDirectory( ), 215
CreateFile( ), 213
DeAllocExch( ), 135
DeAllocPage( ), 134
DeleteDirectory( ), 216
DeleteFile( ), 214
device drivers (operating systems), 158
DeviceInit( ), 149, 154, 157

DeviceOp( ), 149, 154-155
DeviceStat( ), 149, 154, 156
DMASetUp( ), 151, 158
EditLine( ), 586
EndOfIRQ( ), 150-151, 158
exit( ), 140
ExitJob( ), 140, 440, 448
far external (CM32), 703-704
files, 207-217
CloseFile( ), 207-208
CreateDirectory( ), 215
CreateFile( ), 213-214
DeleteDirectory( ), 216
DeleteFile( ), 214-215
GetDirSector( ), 216-217
GetFileLFA( ), 211
GetFileSize( ), 212
OpenFile( ), 207
ReadBlock( ), 208
ReadBytes( ), 209-210
RenameFile( ), 214
SetFileLFA( ), 211-212
SetFileSize( ), 213
WriteBlock( ), 209
WriteBytes( ), 210-211
FindHiPage, 381
FindLoPage, 382-383
FindRun, 385-387
GetCmdLine( ), 137, 448
GetCMOSDate( ), 415-416
GetCMOSTime( ), 414-415
GetDirSector( ), 216
GetDMACount( ), 89
GetExitJob( ), 137, 448
GetFileLFA( ), 211
GetFileSize( ), 212
GetJobNum( ), 136
GetMeSomeMem( ), 52
GetPath( ), 137, 448
GetpJCB( ), 136
GetTimerTick( ), 416
GetUserName( ), 137, 448
GetVidChar ( ), 138
GetXY( ), 586
hdisk_setup( ), 504
InitDevDr( ), 148, 157
InitFS( ), 594
InitMemMgmt( ), 379
InitOSPublics( ), 429
internal public helper, 342-347
GetExchOwner, 344-345
RemoveRdyJob, 342-344
SendAbort, 346-347
SetExchOwner( ), 345-346
ISendMsg( ), 151
job management, 448-469
KillAlarm( ), 151
timer, 412-413
LinToPhy, 385
LoadNewJob( ), 448
local kernel helper, 337-342
ChkRdyQ, 341-342
deQueueMsg, 338-339
deQueueRdy, 340-341
deQueueTSS, 339
enQueueMsg, 337-338
enQueueRdy, 340
MarkPage, 383-384
MaskIRQ( ), 150, 158
MicroDelay( ), 413-414
MoveRequest( ), 146
NewTask( ), 46
Notify on Global Keys( ), 191-192
OpenFile( ), 207
public kernel, 348-375
AllocExch( ), 370-371
CheckMsg( ), 363-366
DeAllocExch( ), 371-373
GetPriority( ), 374-375
GetTTSExch( ), 373
ISendMsg( ), 357-359
MoveRequest( ), 354-355
NewTask( ), 366-368
Request( ), 348-351
Respond( ), 351-353
SendMsg( ), 355-357
SetPriority( ), 374
SpawnTask( ), 368-370
WaitMsg( ), 359-363
PutVidChars ( ), 138
QueryPages( ), 134
Read Keyboard( ), 191
ReadBlock( ), 208
ReadBytes( ), 209
ReadKbd( ), 138
RenameFile( ), 214
Request( ), 49, 135, 144
Respond( ), 49, 144
ScrollVid( ), 575, 582
see also calls
SendMsg ( ), 135
SendMsg( ), 46
SetCmdLine( ), 448
SetExitJob( ), 137, 448
SetFileLFA( ), 211
SetFileSize( ), 213

SetIRQVector( ), 150, 158
SetJobName( ), 137
SetNormVid( ), 575
SetPath( ), 137
Setpath( ), 448
SetSysIn( ), 137
SetUpDMA( ), 89
SetUserName( ), 137, 448
SetXY( ), 585
Sleep( ), 151, 406
timer, 410-411
SpawnTask( ), 46, 136
Tone( ), 418
TTYOut( ), 577
UnMarkPage, 384
UnMaskIRQ( ), 150, 158
wait( ), 44
WaitMsg( ), 46, 135
WriteBlock( ), 209
WriteBytes( ), 210

G

GetCmdLine( ) call, 238-239
GetCmdLine( ) function, 137, 448
GetCMOSDate( ) call, 239, 415-416
GetCMOSTime( ) call, 239, 414-415
GetDirSector( ) function, 216
GetDMACount( ) call, 240
GetDMACount( ) function, 89
GetExchOwner function, 344-345
GetExitJob( ) call, 240
GetExitJob( ) function, 137, 448
GetFileLFA( ) function, 211
GetFileSize( ) function, 212
GetIRQVector( ) call, 240-241
GetJobNum( ) call, 241
GetJobNum( ) function, 136
GetMeSomeMem( ) function, 52
GetNormVid( ) call, 241
GetPath( ) call, 241-242
GetPath( ) function, 137, 448
GetPhyAdd( ) call, 242, 404
GetpJCB( ) call, 242
GetpJCB( ) function, 136
GetPriority( ) function, 374-375
GetSysIn( ) call, 242, 243
GetSysOut( ) call, 243
GetTimerTick( ) call, 243
GetTimerTick( ) function, 416
GetTSSExch( ) call, 243
GetTTSExch( ) function, 373
GetUserName( ) function, 137, 448
GetVidChar ( ) function, 138
GetVidChar( ) call, 244, 581
GetVidOwner( ) call, 244
GetXY( ) call, 244
GetXY( ) function, 586
graphics (video), 87-89

H

handles (files), 203
HardIDE.C (MMURTL source file), 330
hardware, 17-20
abstraction, 82
block-oriented devices, 85-86
buses, 83-84
DMA (Direct Memory Access), 89-97
initializing, 111
isolating, 82
keyboard, 86-87, 566
PICU (Priority Interrupt Controller Unit), 98
procedural interfaces (API), 118-119
tasks, 29-30
timers, 97-98
video, 87-89
hardware requirements (MMURTL), 10-11
HardXY( ) call, 584
hdisk_setup( ) function, 504
Help command, 268
hidden sectors (FAT), 592
hot keys (CLI), 271

I

I/O (public calls), 226
IDE disk device drivers, 498-519

program listings
IDE device driver (listing 24.1), 498-499
IDE device driver (listing 24.2), 499-502
IDE device driver (listing 24.3), 502-504
IDE device driver code (listing 24.4), 504-508
IDE device driver code (listing 24.5), 508-515
IDE device driver code (listing 24.6), 515-519
InByte( ) call, 244
InDWord( ) call, 245
InitCode.ASM (MMURTL source file), 331
InitDevDr( ) call, 245-246
InitDevDr( ) function, 148, 157
InitFS( ) function, 594
initial job (monitor), 160-161
initializing
device drivers, 157-158
FAT (File Allocation Table), 593
hardware, 111
memory, 69-70, 113-114, 379-381
operating system
end of initialization, 429
initialization subroutines, 429-432
initialize call gates and IDT, 433-438
main operating system data, 420-422
operating system initialization and entry point, 423-427
operating system initialization, 427
operating system initialization, 427-429
static variables (CM32), 704-706
system services, 142
tasks, 112
video (listing 26.2), 574
InitKBDService (keyboard), 567-568
InitMemMgmt( ) function, 379
InitOSPublics( ) function, 429
input/output
parallel I/O, 85
serial I/O, 84-85
installing device drivers, 149-150
IntCode.ASM (MMURTL source file), 330
interfaces (procedural), 116-120
calling, 117-118
error handling, 119
files, 206
hardware, 118-119
portability, 119
internal calls, systems programming, 120
internal commands (CLI), 266-270
Cls (Clear Screen), 267
Copy\, 267
Debug, 268
Del (Deleting files), 268
Dir (Directory Listing), 267-268
Exit, 268
Help, 268
MD (Make Directory), 269
Monitor, 268
Path (Set File-Access Path), 269
RD (Remove Directory), 269
Rename (files), 270
Run, 270
internal public helper functions, 342
GetExchOwner, 344-345
RemoveRdyJob, 342-344
SendAbort, 346-347
SetExchOwner( ), 345-346
internal structure (file system), 217-219
FAT (File Allocation Tables), 218-219
File Control Blocks (FCB), 217-218
File User Blocks (FUB), 218
interrupts, 39-40
CM32, 700
debugger, 472-473
device drivers, 150
messaging features, 151
latency, 53
PICU (Priority Interrupt Controller Unit), 98

public calls, 226-227
timer, 408-418
InWord( ) call, 245
InWords( ) call, 245
IPC (Interprocess Communications), 42-51
interrupt latency, 53
memory, 53-54
messages, 44-54
link blocks, 51
Request( ), 48-50
Respond( ), 50-51
multitasking (reetrancy), 51-53
pipes, 44
semaphores, 43-44
synchronization, 42
ISendMsg ( ) call, 246-247
ISendMsg( ) function, 151, 357-359
isolating hardware, 82
ISRs (Interrupt Service Routines), 39-40, 550-551

J

JCB (Job Control Block), 74, 136-137
job management program code, 448-469
subroutines, 440-447
JobC.C (MMURTL source file), 330
JobCode.ASM (MMURTL source file), 330
jobs
monitor
active, 160
initial, 160-161
public calls, 224-225
JUMP instruction (DASM assembler), 679-681

K

kernel
functions
internal public helper, 342-347
local helper, 337-342
public, 348-375
program listings
adding task to ready queue, 340
allocate exchange code, 371
changing owner of exchanges, 345-346
CheckMsg kernel primitive code, 363-366
data segment source, 336-337
de-queueing a message, 338-339
de-queueing a task, 339
de-queueing highest priority task, 341
deallocate exchange code, 372-373
find highest physical page code, 382
finding highest priority task, 341-342
finding owner of exchanges, 344-345
Get priority code, 374-375
Get TSS exchange code, 373
ISendMsg( ) kernel primitive code, 358-359
MoveRequest kernel primitive code, 354-355
NewTask kernel primitive code, 366-368
notifying services of a job's demise, 346-347
queueing for message exchange, 338
removing a terminated task from ready queue, 342-344
Request kernel primitive code, 348-351
Respond kernel primitive code, 351-353
SendMsg kernel primitive code, 356-357
Set priority code, 374
SpawnTask kernel primitive code, 369-370
WaitMsg kernel primitive code, 359-363
source file, 336-337
tasks, 47
Kernel.ASM (MMURTL source file), 331

kernels, 27
key codes (keyboard), 194-198
cursor, 198
edit keys, 198
function keys, 196-197
numeric key pad, 197-198
keyboard, 86-87, 138, 542-570
alphanumeric key values, 193
buffers, 557
debugger, 565
hardware setup, 566
hot keys (CLI), 271
InitKBDService, 567-568
ISR code, 550-551
KBDServiceTask, 557-563
key codes, 194-198
cursor, 198
edit keys, 198
function keys, 196-197
numeric key pad, 197-198
lock-state, 194
numeric keypad, 194
key codes, 197-198
procedural interface, 563-565
program listings
code for blocking read keyboard call, 563-565
debugger read keyboard code, 565
initializing keyboard hardware, 566
keyboard buffer code, 557
keyboard data and tables, 542-550
keyboard ISR code, 550-551
keyboard service initialization, 567-568
keyboard system service code, 558-563
keyboard translation code, 552-556
low-level keyboard code, 568-570
public calls, 227
routines, 568-570
shift-states, 193-194
system services, 190-192
Assign Keyboard( ) function, 192
Notify on Global Keys( ) function, 191-192
Read Keyboard function, 191
translation, 551-556
Keyboard.ASM (MMURTL source file), 329
KillAlarm ( ) call, 247
KillAlarm( ) function, 151
timer, 412-413

L

latency (interrupts), 53
LBA (Logical Block Address), 592-593
LFA (Logical File Addresses), 204
library functions (CM32), 708-709
library header file access (CM32), 702
linear memory, 56
allocating, 76
deallocating, 76-77
tracking, 63-68
link blocks (messages), 51
linked lists
program listings
linked list memory allocation, 65-66
linked list memory management, 64
memory, 63-68
linkers (DASM), 664-665
LinToPhy function, 385
list files (DASM assembler), 667-668
listings
5.1 linked list memory management, 64
5.2 linked list memory allocation, 65-66
5.3 returning memory to free pool, 67-68
10.1 system service, 143-144
11.1 initial.Job.file, 160-161
11.2 monitor program, 163-179
14.1 File System request, 204-205
16.1 CLI (Command Line Interpreter), 271-288
16.2 Editor source code, 290-314
16.3 DumbTerm, 324
16.4 Print program, 324
16.5 System Service, 322-323

16.6 TestSvc.C, 324
18.1 kernel data segment source, 336-337
18.2 queueing for message exchange, 338
18.3 de-queueing a message, 338-339
18.4 de-queueing a task, 339
18.5 adding task to ready queue, 340
18.6 de-queueing highest priority task, 341
18.7 finding highest priority task, 341-342
18.8 removing a terminated task from ready queue, 342-344
18.9 finding owner of exchanges, 344-345
18.10 changing owner of exchanges, 345-346
18.11 notifying services of a job's demise, 346-347
18.12 Request kernel primitive code, 348-351
18.13 Respond kernel primitive code, 351-353
18.14 MoveRequest kernel primitive code, 354-355
18.15 SendMsg kernel primitive code, 356-357
18.16 ISendMsg( ) kernel primitive code, 358-359
18.17 WaitMsg kernel primitive code, 359-363
18.18 CheckMsg kernel primitive code, 363-366
18.19 NewTask kernel primitive code, 366-368
18.20 SpawnTask kernel primitive code, 369-370
18.21 allocate exchange code, 371
18.22 deallocate exchange code, 372-373
18.23 Get TSS exchange code, 373
18.24 Set priority code, 374
18.25 get priority code, 374-375
19.1 memory management, 378
19.2 memory management initialization code, 379
19.3 memory management initialization code, 379-380
19.4 memory management initialization code, 380
19.5 paged memory management initialization code, 381
19.6 allocation of memory management exchange, 381
19.7 memory management initialization, 381
19.8 find highest physical page code, 382
19.9 find lowest physical page code, 383
19.10 code to mark a physical page in use, 384
19.11 code to free physical page for reuse, 384
19.12 converting physical addresses to linear, 385
19.13 finding free linear memory, 386-387
19.14 adding linear memory, 387-388
19.15 adding aliased linear memory, 389-390
19.16 adding page table for user memory, 390-392
19.17 operating system page tables, 392-394
19.18 adding call gates to the GDT, 394-395
19.19 adding entries to IDT, 395-396
19.20 allocating a page of linear memory, 396-397
19.21 code to alias memory, 398-400
19.22 code to remove alias PTEs, 400-401
19.23 deallocating linear memory, 401-403
19.24 code to find number of free pages, 403-404
19.25 public function converting linear to physic, 404
20.1 loop in timer ISR, 406-407
20.2 data and constants for timer, 407-408
20.3 external functions for timer code, 408
20.4 timer interrupts, 409-410
20.5 code for sleep function, 411
20.6 code for Alarm( ) function, 412
20.7 code for KillAlarm( ) function, 413
20.8 code for the MicroDelay( ) function, 414
20.9 code reading CMOS time, 414-415

20.10 code reading CMOS date, 415-416
20.11 code returning timer ticks, 416
20.12 helper function for Beep( ) and Tone( ), 417
20.13 code producing beeps, 418
20.14 code producing tones, 418
21.1 main operating system data, 420-422
21.2 operating system initialization and entry point, 423-427
21.3 operating system initialization, 427
21.4 operating system initialization, 427-429
21.5 end of initialization, 429
21.6 initialization subroutines, 429-432
21.7 initialize call gates and IDT, 433-438
22.1 job management subroutines, 440-447
22.2 job management code, 448-469
23.1 exception handlers for entering debugger, 472-473
23.2 debugger data and code, 474-496
24.1 IDE device driver, 498-499
24.2 IDE device driver, 499-502
24.3 IDE device driver, 502-504
24.4 IDE device driver code, 504-508
24.5 IDE device driver code, 508-515
24.6 IDE device driver code, 515-519
24.7 RS-232. header file, 519-520
24.8 RS-232 device driver, 521-523
24.9 RS-232 device driver, 523-525
24.10 RS-232 device driver, 525
24.11 RS-232 device driver, 526
24.12 RS-232 device driver, 526-528
24.13 RS-232 device driver code, 528-535
24.14 RS-232 device driver code, 535-540
25.1 keyboard data and tables, 542-550
25.2 keyboard ISR code, 550-551
25.3 keyboard translation code, 552-556
25.4 keyboard buffer code, 557
25.5 keyboard system service code, 558-563
25.6 code for blocking read keyboard call, 563-565
25.7 debugger read keyboard code, 565
25.8 initializing keyboard hardware, 566
25.9 keyboard service initialization, 567-568
25.10 low-level keyboard code, 568-570
26.1 video constants and data, 573
26.2 initializing video, 574
26.3 code to set video owner, 574-575
26.4 code to set normal video attribute, 576
26.5 code to set normal video attribute, 576
26.6 code to get current video owner, 576-577
26.7 code to clear screen, 577
26.8 code for TTY output to screen, 578-579
26.9 code to screen video attributes, 580
26.10 code to place characters, 581
26.11 code to get characters and attributes, 582
26.12 code to scroll area of screen, 582-584
26.13 code to set cursor position, 585
26.14 code to set cursor position, 585-586
26.15 code to return to current cursor position, 586
26.16 code to edit lines on screen, 587-589
27.1 MS-DOS FAT-compatible file system, 594-660
28.1 application template file, 666
28.2 application template file, 666-667
28.3 alternating data and code in template file, 669-670
29.1 library functions (CM32), 708-709

29.2 library code--I/O file for MMURTL (CM32), 709-714
29.3 standard string functions (CM32), 715-718
loading memory, 77-78
LoadNewJob( ) call, 247
LoadNewJob( ) function, 448
local declarations (DASM assembler), 668-669
local kernel helper functions, 337
ChkRdyQ, 341-342
deQueueMsg, 338-339
deQueueRdy, 340-341
deQueueTSS, 339
enQueueMsg, 337-338
enQueueRdy, 340
lock-state (keyboard), 194
Logical File Addresses (LFA), 204
logical memory, 56

M

MAIN.ASM (MMURTL source file), 329
mapping memory, 73-75
MarkPage function, 383-384
MaskIRQ ( ) call, 248
MaskIRQ( ) function, 150, 158
MD command (Make Directory), 269
MemCode.ASM (MMURTL source file), 330
memory
32-bit software, 131
address aliasing, 78
allocating, 63
CM32, 699
DASM assembler, 669
demand-paged, 61
demand-paged flat, 59
demand-paged virtual, 61
DMA (Direct Memory Access), 89-97
functions
AddAliasRun, 388-390
AddOSPT, 392-394
AddRun, 387-388
AddUserPT, 390-392
FindHiPage, 381-382
FindLoPage, 382-383
FindRun, 385-387
InitMemMgmt, 379-381
LinToPhy, 385
MarkPage, 383-384
UnMarkPage, 384
initializing, 69-70, 113-114, 379-381
IPC, 53-54
linear, 56
allocating, 76
deallocating, 76-77
tracking, 63-68
linked lists, 63-68
linked list memory allocation, 65-66
linked list memory management, 64
loading, 77-78
logical, 56
MMURTL, 132-135
paged flat, 58
paging, 72-80
allocating linear memory, 76
allocating physical memory, 77
deallocating linear memory, 76-77
directories (PDs), 72-76
mapping, 73-75
tables, 72
physical, 56
allocating, 77
tracking, 63-69
pointers, 78-79
program listings
adding aliased linear memory, 389-390
adding call gates with GDT, 394-395
adding entries to IDT, 395-396
adding linear memory, 387-388
adding page table for user memory, 390-392
allocating a page of linear memory, 396-397
allocation of memory management exchange, 381
code for KillAlarm( ) function, 413
code reading CMOS date, 415-416
code to alias memory, 398-400
code to find number of free pages, 403-404
code to free physical page for reuse, 384

code to mark a physical page in use, 384
code to remove alias PTEs, 400-401
converting physical addresses to linear, 385
deallocating linear memory, 401-403
external functions for timer code, 408
find lowest physical page code, 383
finding free linear memory, 386-387
memory management, 378
memory management initialization, 381
memory management initialization code, 379-380
operating system page tables, 392-394
paged memory management, 381
public function converting linear to physic, 404
returning memory to free pool, 67-68
protection, 70
public calls, 225-226, 394
AddGDTCallGate( ), 394-395
AddIDTGate( ), 395-396
AliasMem( ), 397-400
AllocOSPage( ), 396-397
DeAliasMem( ), 400-401
DeAllocMem( ), 401-403
GetPhyAdd( ), 404
QueryMemPages( ), 403-404
ROM (Read Only Memory), 100-110
segmented, 61-62, 71
simple flat, 57-58
structures, 112
tables, 68, 112
virtual, 8
virtual paged, 59-60
messages
applications, 135
IPC
Request( ), 48-50
Respond( ), 50-51
IPC (Interprocess Communications), 44-54
link blocks, 51
public calls, 224
system services, 142
asynchronous, 146
errors, 146-147
initializing, 142
listing 10.1, 143-144
request blocks, 144-145
MicroDelay( ) call, 248
MicroDelay( ) function, 413-414
microprocessors, 6
debugger, 185-188
MiscCode.ASM (MMURTL source file), 330
MMURTL, 9-11
compiling, 331-332
hardware requirements, 10-11
memory, 132-135
operating systems (protection), 134-135
source code
borrowing, 333
files, 328-331
modular, 332
models (scheduling tasks), 38-40
modes (files)
Block, 203-204
open, 203
Stream, 204
monitor
active job, 160
function keys, 161
initial job, 160-161
performance statistics, 162-163
program listing, 163-179
program theory, 161-162
Monitor command, 268
Monitor.C(MMURTL source file), 331
MOSDPR.ASM (MMURTL source file), 329
MOSGDT.ASM (MMURTL source file), 329
MOSIDT.ASM (MMURTL source file), 329
MoveRequest( ) call, 248-249
MoveRequest( ) function, 146, 354-355
Mpublics.ASM (MMURTL source file), 329
multiple users (tasks), 28-29

multitasking, 7, 31-36
cooperative, 31-32
preemptive, 27, 32
reentrancy, 51-53
starting tasks, 136

N

naming files, 202
network file specifications, 202-203
NewTask( ) call, 249
NewTask( ) function, 46, 366-368
Notify on Global Keys( ) function, 191-192
NumCnvrt.ASM (MMURTL source file), 330
numeric keypad
key codes, 197-198
keyboard, 194

O

open modes (files), 203
OpenFile( ) function, 207
operating system, 420-422
entry point, 422-429
initialization subroutines, 429-438
program listings
end of initialization, 429
initialization subroutines, 429-432
initialize call gates and IDT, 433-438
operating system initialization, 427-429
operating system initialization and entry point, 423-427
VIRTUAL command, 422-429
operating systems
calling stacks, 131-134
device drivers (functions), 158
program listings (main operating system data), 420-422
protection, 134-135
OutByte( ) call, 249
OutDWord( ) call, 249
OutWord( ) call, 250

P

Page Allocation Map (PAM), 77
paged flat memory, 58
paging memory
allocating
linear, 76
physical, 77
deallocating linear, 76-77
directories (PDs), 72-76
mapping, 73-75
tables, 72
PAM (Page Allocation Map), 77
parallel I/O, 85
Parallel.C (MMURTL source file), 330
parameters (public calls), 222-223
Path command (Set File-Access Path), 269
PDs (page directories), 72-76
Pentium microprocessors, 6
performance monitoring, 162-163
physical memory, 56
allocating, 77
tracking, 63-69
physical sector numbers (FAT), 592
PICU (Priority Interrupt Controller Unit), 98
pipes (IPC), 44
pointers
CM32, 704
memory, 78-79
portability, procedural interfaces (API), 119
preemptive multitasking, 27, 32
prefixes (DASM assembler), 678-679
preprocessor commands (C language), 702
Print application, 324
procedural interfaces
API, 116-120
calling, 117-118
error handling, 119
hardware, 118-119
portability, 119
files, 206
keyboard, 563-565

processors
386, 6
486, 6
debugger, 185-188
Pentium, 6
protecting
memory, 70
operating systems, 134-135
PTs (page tables), 72
public calls, 222
device drivers, 225
I/O, 226
interrupts, 226-227
jobs, 224-225
keyboard, 227
memory, 225-226, 394-404
AddGDTCallGate( ), 394-395
AddIDTGate( ), 395-396
AliasMem( ), 397-400
AllocOSPage( ), 396-397
DeAliasMem( ), 400-401
DeAllocMem( ), 401-403
GetPhyAdd( ), 404
QueryMemPages( ), 403-404
messaging, 224
moving data, 226
parameters, 222-223
speakers, 227
stacks, 224
system service, 225
tasks, 225
timers, 226
video, 227
PUBLIC declarations (DASM assembler), 677
public kernel functions, 348-375
AllocExch( ), 370-371
CheckMsg( ), 363-366
DeAllocExch( ), 371-373
GetPriority( ), 374-375
GetTTSExch( ), 373
ISendMsg( ), 357-359
MoveRequest( ), 354-355
NewTask( ), 366-368
Request( ), 348-351
Respond( ), 351-353
SendMsg( ), 355-357
SetPriority( ), 374
SpawnTask( ), 368-370
WaitMsg( ), 359-363
PutVidAttrs( ) call, 250, 580
PutVidChars ( ) function, 138
PutVidChars( ) call, 250-251, 580

Q

QueryMemPages( ) call, 403-404
QueryPages( ) call, 251
QueryPages( ) function, 134

R

RD command (Remove Directory), 269
Read Keyboard( ) function, 191
ReadBlock( ) function, 208
ReadBytes( ) function, 209
ReadCMOS( ) call, 251
reading files, 220
ReadKbd ( ) call, 251-252
ReadKbd( ) function, 138
real-time, 7
scheduling tasks, 35-36
reclaiming resources, 440-469
reentrancy (multitasking), 51-53
RegisterSvc( ) call, 252
RemoveRdyJob function, 342-344
Rename command (files), 270
RenameFile( ) function, 214
request blocks
system services, 144-145
caller information, 145-150
service code, 145
Request( ) call, 252-253
Request( ) function, 49, 135, 144, 348-351
requests (file system), 204-205
resources (reclamation), 440-469
Respond( ) call, 254
Respond( ) function, 49, 144, 351-353
ROM (Read Only Memory), 100-110
routines (keyboard), 568-570

RQBCode.ASM (MMURTL source file), 330
RS-232 asynchronous device driver, 519-540
program listings
device driver (listing 24.10), 525
device driver (listing 24.11), 526
device driver (listing 24.12), 526-528
device driver (listing 24.13), 528-535
device driver (listing 24.14), 535-540
device driver (listing 24.8), 521-523
device driver (listing 24.9), 523-525
header file, 519-520
RS232.C (MMURTL source file), 330
Run command, 270

S

scheduling tasks, 32-39
combining techniques, 36-38
cooperative queueing, 34
models, 38-40
prioritized cooperative, 34
real-time, 35-36
time slicing, 33-35
screen editor, 288
ScrollVid( ) call, 254
ScrollVid( ) function, 575, 582
sectors
boot ROM, 101-110
FAT (File Allocation Table)
boot, 593
hidden, 592
initializing, 593
LBA (Logical Block Address), 592-593
physical sector numbers, 592
segmented memory, 61-62, 71
semaphores (IPC), 43-44
send( ) function, 44
SendAbort function, 346-347
SendMsg ( ) call, 254-255
SendMsg( ) function, 46, 135, 355-357
serial I/O, 84-85
service code (request blocks), 145
SetCmdLine( ) call, 255
SetCmdLine( ) function, 448
SetExchOwner( ) function, 345-346
SetExitJob( ) call, 255-256
SetExitJob( ) function, 137, 448
SetFileLFA( ) function, 211
SetFileSize( ) function, 213
SetIRQVector( ) call, 256-257
SetIRQVector( ) function, 150, 158
SetJobName( ) call, 257
SetJobName( ) function, 137
SetNormVid( ) call, 257-258
SetNormVid( ) function, 575
SetPath( ) call, 258
SetPath( ) function, 137, 448
SetPriority( ) call, 258
SetPriority( ) function, 374
SetSysIn( ) call, 258-259
SetSysIn( ) function, 137
SetSysOut( ) call, 259
SetUpDMA( ) function, 89
SetUserName( ) call, 259
SetUserName( ) function, 137, 448
SetVidOwner( ) call, 259-260, 574
SetXY( ) call, 260
SetXY( ) function, 585
shift right function (CM32), 706-707
shift-states (keyboard), 193-194
simple flat memory, 57-58
single tasking, 30
single user tasks, 28-29
Sleep( ) call, 260
Sleep( ) function, 151, 406
timer, 410-411
software
32-bit, 130-131
tasks, 29
source code
borrowing, 333

calling conventions, 328
comments, 328
debugger, 473-496
kernel, 336-337
MMURTL, 328-331
modular, 332
SpawnTask( ) call, 261
SpawnTask( ) function, 46, 136, 368-370
speakers (public calls), 227
stacks
calling operating systems, 132-134
public calls, 224
standard interval (timer), 406-407
starting tasks (multitasking), 136
statements (C language), 701-702
static variables (CM32), 704-706
statistics, 162-163
storage
classes (C language), 701-702
DASM assembler, 675-676
Stream mode (files), 204
structures, 112
CM32, 706
dynamic, 114
subroutines
initialization, 429-438
job management, 440-447
SVCCode.ASM (MMURTL source file), 330
synchronization (IPC), 42
system resources (device drivers), 150-151
system services
asynchronous, 146
errors, 146-147
keyboard, 190-192, 557-563
NAssign Keyboard( ) function, 192
Notify on Global Keys( ) function, 191-192
Read Keyboard function, 191
messages, 142-147
initialzing, 142
listing 10.1, 143-144
request blocks, 144-145
program listings, 322-324
Simple Service source code, 322-324
TestSvc.C, 324
public calls, 225
systems programming (API), 120-125
device drivers, 120-125
calling drivers, 124-125
calling operating system, 122-124
internal calls, 120

T

tables
dynamic, 114
memory, 68
paging, 72
static, 112
tagged fields (executable files), 689
Task State Segment (TSS), 74
tasks, 26-30
CPU time, 30-32
hardware, 29-30
initializing, 112
IPC, 42-51
messages, 44-54
pipes, 44
semaphores, 43-44
synchronization, 42
kernel, 47
multitasking, 31-36
cooperative, 31-32
preemptive, 32
public calls, 225
scheduling, 32-39
combining techniques, 36-38
cooperative queueing, 34
models, 38-40
prioritized cooperative, 34
real-time, 35-36
time slicing, 33-35
single, 30
software, 29
starting, 136
structures, 112
tables (static), 112
users
multiple, 28-29
single, 28-29
TestSvc.C (program listing), 324
threads, *see* tasks
thunking, 19
time slicing (scheduling tasks), 33-35

timer
device drivers, 151
functions
Alarm( ), 411-412
Beep( ), 417-418
Beep_Work( ), 416-417
GetCMOSDate( ), 415-416
GetCMOSTime( ), 414-415
GetTimerTick( ), 416
KillAlarm( ), 412-413
MicroDelay( ), 413-414
Sleep( ), 410-411
Tone( ), 418
interrupts, 408-418
hardware, 97-98
program listings
code for Alarm( ) function, 412
code for sleep function, 411
code for the Micro Delay( ) function, 414
code producing beeps, 418
code producing tones, 418
code reading CMOS time, 414-415
code returning timer ticks, 416
data and constants for timer, 407-408
helper function for Beep( ) and Tone( ), 417
loop in timer ISR, 406-407
timer interrupts, 409-410
standard interval, 406-407
public calls, 226
TmrCode.ASM (MMURTL source file), 330
Tone( ) call, 261-262
Tone( ) function, 418
tracking memory
linear, 63-68
linked lists, 63-68
physical, 63-69
transferring data (block-oriented devices), 85-86
translating keyboard information, 551-556
TSS (Task State Segment), 74
TTYOut ( ) call, 262
TTYOut( ) function, 577
type conversion (CM32), 704

U

UARTs (Universal Asynchronous Receiver Transmitters), 84
UASM.ASM (MMURTL source file), 329
UnMarkPage function, 384
UnMaskIQR ( ) call, 262-263
UnMaskIRQ( ) function, 150, 158
UnRegisterSvc( ) call, 263
user tasks, 28-29

V

VGA text mode (video), 572
video, 138-140, 573-589
hardware, 87-89
program listings
code for TTY stream output to screen, 578-579
code to clear screen, 577
code to edit lines on screen, 587-589
code to get characters and attributes, 582
code to get current video owner, 576-577
code to place characters, 581
code to return to cursor position, 586
code to screen video attributes, 580
code to scroll area of screen, 582-584
code to set cursor position, 585
code to set cursor position, 585-586
code to set normal video attribute, 576
code to set normal video attribute, 576
code to set video owner, 574-575
initializing video, 574
video constants and data, 573
public calls, 227
VGA text mode, 572
virtual, 572

Video.ASM (MMURTL source file), 329
VIRTUAL command, 422-429
virtual memory, 8
virtual paged memory, 59-60
virtual video, 572

W-Z

wait( ) function, 44
WaitMsg( ) call, 263
WaitMsg( ) function, 46, 135, 359-363
WriteBlock( ) function, 209
WriteBytes( ) function, 210
writing
device drivers, 147-158
calling, 149
DMA, 151
installing, 149-150
interrupts, 150-151
system resources, 150-151
theory, 148
timers, 151
files, 220

Add to Your Sams Library Today with the Best Books for Programming, Operating Systems, and New Technologies

The easiest way to order is to pick up the phone and call

1-800-428-5331

between 9:00 a.m. and 5:00 p.m. EST.

For faster service please have your credit card available.

ISBN	Quantity	Description of Item	Unit Cost	Total Cost
0-672-48470-6		Assembly Language: For Real Programmers Only! (Book/Disk)	$44.95	
0-672-30286-1		C Programmer's Guide to Serial Communications, Second Edition	$39.95	
0-672-30050-8		C Programmer's Guide to NetBIOS, IPX, and SPX (Book/Disk)	$49.95	
0-672-30475-9		PC Programmer's Guide to Low-Level Functions and Interrupts (Book/Disk)	$45.00	
0-672-30364-7		Win32 API Desktop Reference (Book/CD-ROM)	$49.95	
0-672-30471-6		Teach Yourself Advanced C in 21 Days (Book/Disk)	$34.95	
0-56686-127-6		The Winn L. Rosch Hardware Bible, Third Edition	$35.00	
0-672-30624-7		Peter Norton's Inside the PC, Sixth Edition	$35.00	
0-672-30291-8		DOS 6 Developer's Guide (Book/Disk)	$39.95	
0-672-22789-4		The Waite Group's MS-DOS Bible, 4th Edition	$29.95	
0-672-30501-1		Understanding Data Communications, 4th Edition	$29.99	
0-672-30160-1		Multimedia Developer's Guide (Book/Disk)	$49.95	
0-672-30309-4		Programming Sound for DOS and Windows (Book/Disk)	$39.95	
❑ 3 ½" Disk		Shipping and Handling: See information below.		
❑ 5 ¼" Disk		TOTAL		

Shipping and Handling: $4.00 for the first book, and $1.75 for each additional book. Floppy disk: add $1.75 for shipping and handling. If you need to have it NOW, we can ship product to you in 24 hours for an additional charge of approximately $18.00, and you will receive your item overnight or in two days. Overseas shipping and handling adds $2.00 per book and $8.00 for up to three disks. Prices subject to change. Call for availability and pricing information on latest editions.

201 W. 103rd Street, Indianapolis, Indiana 46290

1-800-428-5331 — Orders 1-800-835-3202 — FAX 1-800-858-7674 — Customer Service

Book ISBN 0-672-30655-7

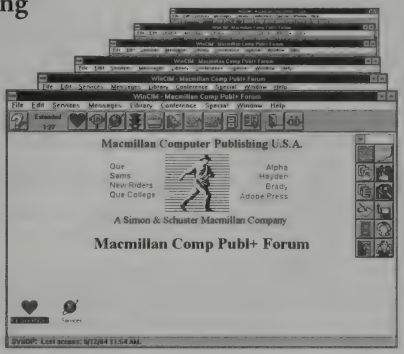
GET CONNECTED
to the ultimate source of computer information!

The MCP Forum on CompuServe

Go online with the world's leading computer book publisher! Macmillan Computer Publishing offers everything you need for computer success!

Find the books that are right for you!
A complete online catalog, plus sample chapters and tables of contents give you an in-depth look at all our books. The best way to shop or browse!

- Get fast answers and technical support for MCP books and software
- Join discussion groups on major computer subjects
- Interact with our expert authors via e-mail and conferences
- Download software from our immense library:
  - Source code from books
  - Demos of hot software
  - The best shareware and freeware
  - Graphics files



Join now and get a free CompuServe Starter Kit!

To receive your free CompuServe Introductory Membership, call **1-800-848-8199** and ask for representative #597.

The Starter Kit includes:
- Personal ID number and password
- $15 credit on the system
- Subscription to *CompuServe Magazine*

Once on the CompuServe System, type:

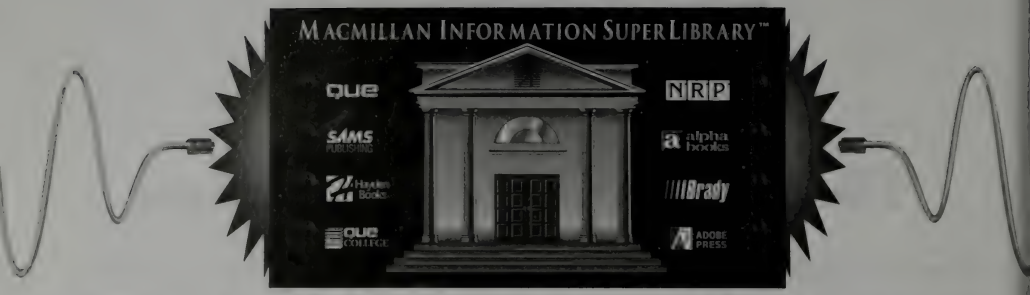
GO MACMILLAN

for the most computer information anywhere!

MACMILLAN COMPUTER PUBLISHING

CompuServe

PLUG YOURSELF INTO...



THE MACMILLAN INFORMATION SUPERLIBRARY™

Free information and vast computer resources from the world's leading computer book publisher—online!

FIND THE BOOKS THAT ARE RIGHT FOR YOU!

A complete online catalog, plus sample chapters and tables of contents give you an in-depth look at ***all*** of our books, including hard-to-find titles. It's the best way to find the books you need!

- STAY INFORMED with the latest computer industry news through our online newsletter, press releases, and customized Information SuperLibrary Reports.
- GET FAST ANSWERS to your questions about MCP books and software.
- VISIT our online bookstore for the latest information and editions!
- COMMUNICATE with our expert authors through e-mail and conferences.
- DOWNLOAD SOFTWARE from the immense MCP library:
  - Source code and files from MCP books
  - The best shareware, freeware, and demos
- DISCOVER HOT SPOTS on other parts of the Internet.
- WIN BOOKS in ongoing contests and giveaways!

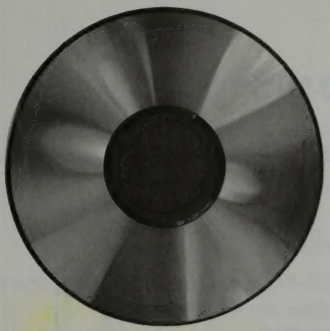
TO PLUG INTO MCP: ➔

GOPHER: gopher.mcp.com

FTP: ftp.mcp.com

WORLD WIDE WEB: **http://www.mcp.com**





Installing Your CD-ROM

What's on the CD-ROM

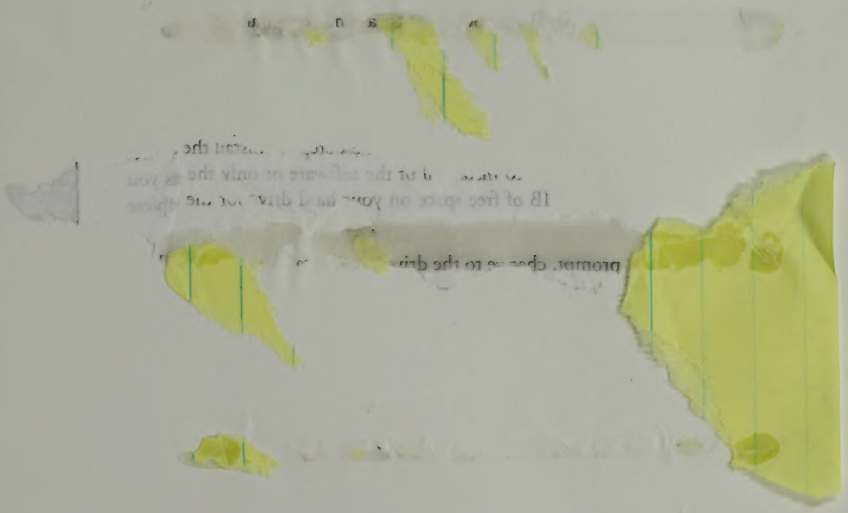
The CD-ROM contains the complete MMURTL operating system source code; sample applications, including source code; a 32-bit assembler, including source code; and a 32-bit C compiler, including source code.

Installing the CD-ROM

Insert the disc in your CD-ROM drive and follow these steps to install the software. The install program enables you to install all of the software or only the parts you need. You'll need at least 20M of free disk space for the installation.

1. From the DOS prompt, change to the drive that contains the CD-ROM. For example, if the disk is in drive D:, type **D:** and press Enter.
2. Type **INSTALL** and press Enter.
3. The install program gives you the option to install all the software on the CD-ROM or only the parts you need. Use the up arrow and down arrow keys to highlight your choice on the menu and press Enter.
4. After installing the software, press the Esc key to exit the install program.

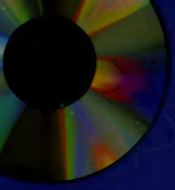
Read the file README.TXT on the CD-ROM for more information on the files that are installed. Chapter 1, "Overview," explains how to start using the software.



By opening this package, you are agreeing to be bound by the following agreement:

This software product is copyrighted, and all rights are reserved by the publisher and author. You are licensed to use this software on a single computer. You may copy and/or modify the software as needed to facilitate your use of it on a single computer. Making copies of the software for any other purpose is a violation of United States copyright laws.

This software is sold *as is* without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither the publisher nor its dealers or distributors assumes any liability for any alleged or actual damages arising from the use of this program. (Some states do not allow for the exclusion of implied warranties, so the exclusion may not apply to you.)



The companion CD-ROM contains

- Complete source code for a 32-bit operating system (MMURTL)
- 32-bit C compiler
- 32-bit assembler
- Sample/test programs for MMURTL

With *Developing Your Own 32-Bit Operating System*, you'll

- Learn the intricacies of 32-bit programming
- Gain an understanding of 32-bit operating systems
- Learn about real-time, multitasking operating systems
- Master MMURTL, the example 32-bit operating system included on the CD-ROM

User Level

New | Casual | Accomplished | Expert

Category: Programming

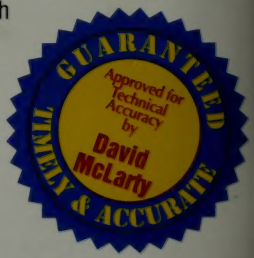
Covers: 32-Bit Programming

Developing Your Own 32-BIT OPERATING SYSTEM

This tutorial builds upon an intermediate programmer's knowledge and explains how to design and develop a feature-rich operating system. With *Developing Your Own 32-Bit Operating System*, you'll not only get the theory behind basic operating system design, but also learn how to build your own operating system from scratch.

Meet MMURTL, a full-featured, 32-bit, message-based, multitasking, real-time operating system that you can modify and use. In addition to learning how to program an operating system, you'll gain a general understanding of 32-bit programming and how other 32-bit operating systems work. *Developing Your Own 32-Bit Operating System* prepares you for the future in 32-bit systems programming.

Richard A. Burgess is a retired U.S. Coast Guard commissioned officer. He taught computer electronics technology at the Computer Learning Center in Washington D.C. for two years, and he consulted in and around Washington with The D-Group. Currently, he is the Chief Information Officer with Optipat, Inc., a rapidly growing, high-technology, patent, trademark, and copyright imaging firm.



$49.99 USA
$67.99 CAN
£39.50 Net UK (inc of VAT)

ISBN 0-672-30655-7



SAMS PUBLISHING